ECON 6140

Problem Set 4

Gabe Sekeres

March 4, 2025

Income Fluctuations Problem

Remark. Code is below, after the solutions to the Credit Markets problem.

Before writing the code, observe that the households solve the problem

$$\max_{\{a_{t+1} c_t\}} \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t \frac{c^{1-\gamma}}{1-\gamma}\right]$$

subject to

$$\begin{aligned} c_t + a_{t+1} &= (1+r)a_t + y_t && \text{(Budget Constraint)} \\ a_{t+1} &\geq -b && \text{(Borrowing Constraint)} \\ \left[\mathbb{P}\{y_t = y_j \mid y_{t-1} = y_i\}\right]_{ij} &\equiv \begin{bmatrix} \pi_{HH} & \pi_{HL} \\ \pi_{LH} & \pi_{LL} \end{bmatrix}_{ij} && \text{(Markov Process)} \end{aligned}$$

Following the strategy in QuantEcon, we will solve this using Euler Equation Time Iteration rather than Value Function Iteration. Formally, we have that the Euler Equation is

$$c_t^{-\gamma} \leq \beta(1+r)\mathbb{E}\left[c_{t+1}^{-\gamma} \mid y_t\right]$$

where this will hold with equality if and only if the borrowing constraint does not bind. Since income follows a (memoryless) Markov process, we look for time-invariant policy functions $a'(y,a)$ and $c(y,a)$ for savings and consumption respectively. Since utility is strictly increasing in consumption and CRRA utility function satisfy Inada, we can write $c(a,y) = (1+r)a + y - a'(a,y)$. Additionally, since the Euler Equation holding with equality (implying an interior solution) and the borrowing constraint binding are mutually exclusive, we can solve for the (functional) solution to the Euler Equation:

$$[c(a,y)]^{-\gamma} = \beta(1+r) \sum_{y' \in \{y_H, y_L\}} \pi(y' \mid y)\left[c(a'(a,y), y')\right]^{-\gamma}$$

and if the solution satisfies $a'(a,y) > -b$, we are done. Otherwise, we set $a'(a,y) = -b$ and solve for optimal consumption. With the structure defined, we can solve the problem computationally:

1. I solved for the optimal consumption and savings policies when $r = 0.01$ and $b = 0$, using a tolerance of $1 \cdot 10^{-6}$. I achieved convergence in 145 iterations using a bisection algorithm.

2. The optimal savings policy function is plotted in Figure 1, and the optimal consumption policy function is plotted in Figure 2.

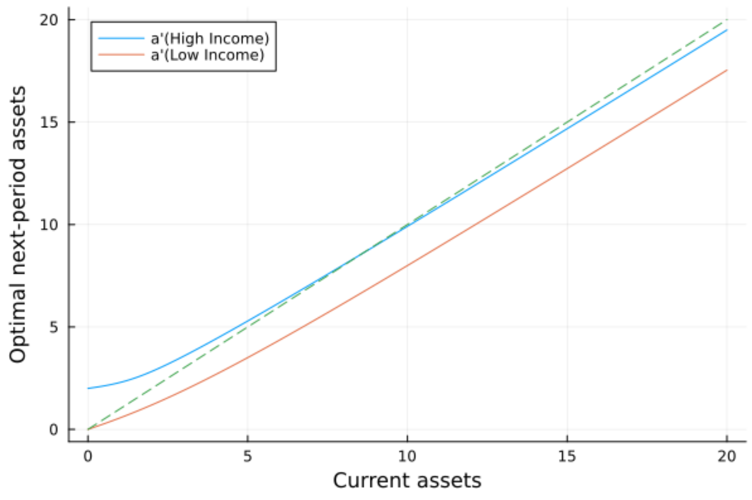

Figure 1: Optimal Savings Policy, $b = 0$

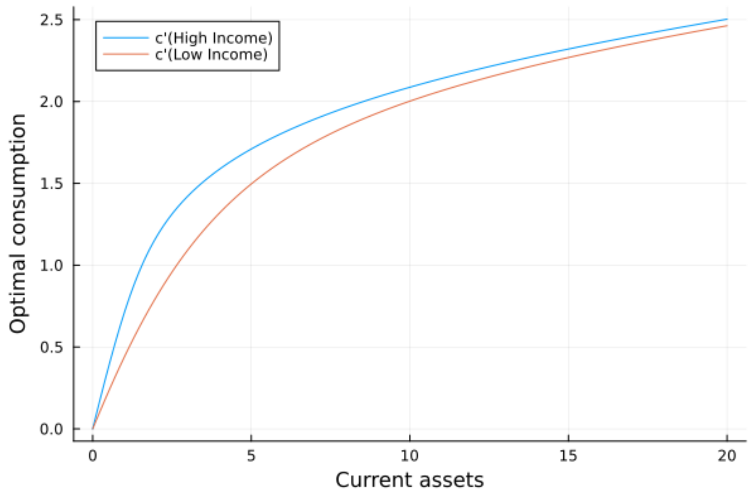

Figure 2: Optimal Consumption Policy, $b = 0$

3. I increased the coefficient of relative risk aversion to 3, and plotted the new policies. I achieved convergence in 196 iterations. The results are Figure 3 and Figure 4. As we can see, compared to the earlier plots the agent will consume less and save more when they are in the high-income state. As they are more risk-averse, they want to prevent the bad outcome – less consumption when they have low-income – more than in the previous question.

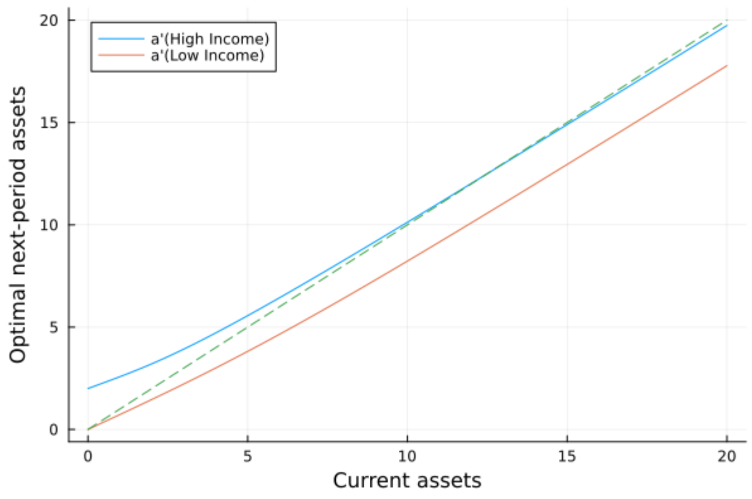

Figure 3: Optimal Savings Policy, $\gamma = 3$

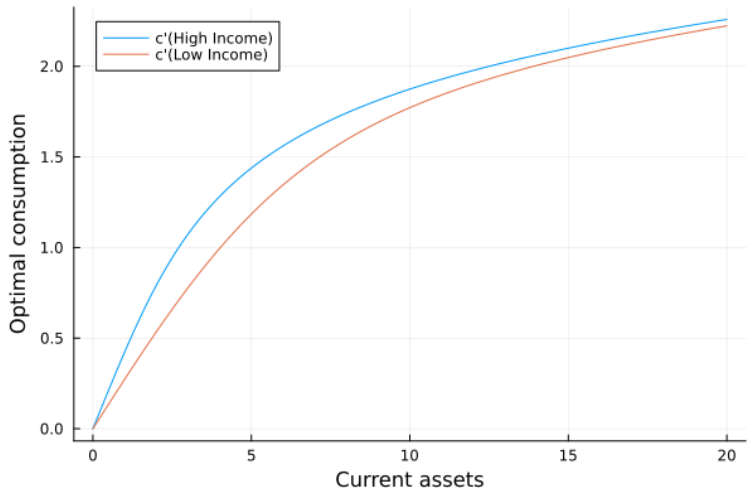

Figure 4: Optimal Consumption Policy, $\gamma = 3$

4. I set the borrowing constraint to $b = 4$, and solved for the new policies. I achieved convergence in 145 iterations. The results are Figure 5 and Figure 6. As we can see, consumption increases and savings decrease in this model. This is because the households know that they can borrow if they end up in the bad state, and so are more willing to consume in the current period, especially when they have higher income.

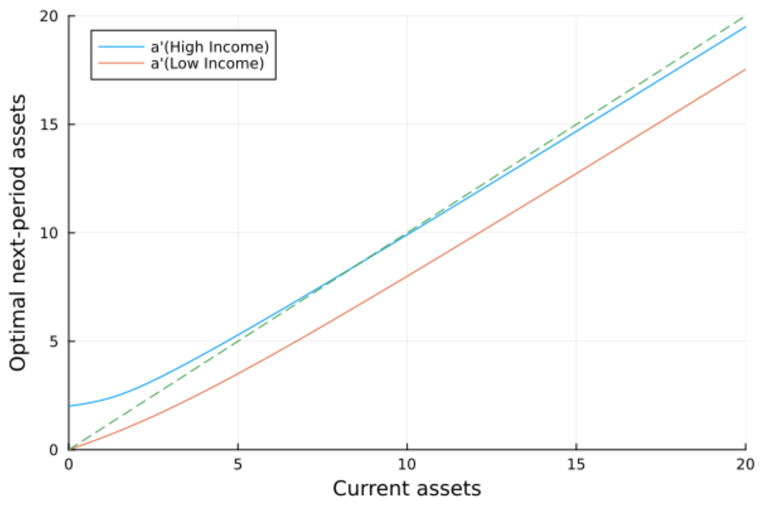

Figure 5: Optimal Savings Policy, $b = 4$

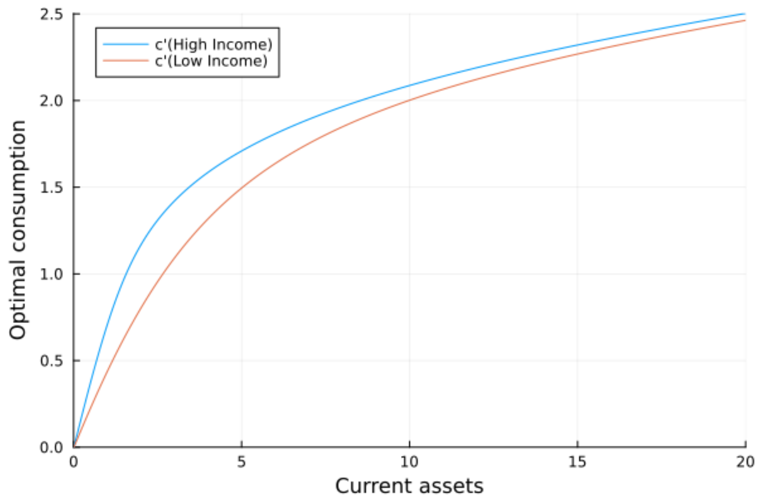

Figure 6: Optimal Consumption Policy, $b = 4$

5. I constructed two helper functions – one to find the stationary distribution of the Markov chain (this is standard, and I copied older code), and one to simulate the series of asset choices over time (for 250,000 periods). I solved the model for 100 values of r between 0.00 and $1/\beta - 1 \approx 0.0204$, the maximum interest rate that would ensure that $(1+r)\beta \leq 1$, and attained the aggregate (mean) assets for each of those interest rates. They are plotted in Figure 7. As we would expect, savings are directly increasing in the interest rates. Annoyingly, we have divergence as we get close to $r = 0.02$ (specifically, at the 89th iteration, when $r = 0.01814058956916102$). This doesn't make a ton of sense, but it appears to be a consistent pattern.

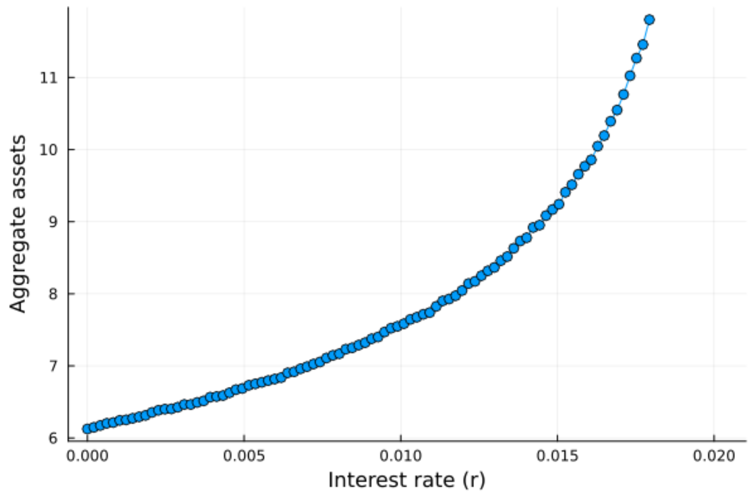

Figure 7: Aggregate Assets and Interest Rate

As a reminder, the entirety of the code I used is below, in the Code Section.

Credit Markets and Economic Growth

1. Observe that if a worker with entrepreneurial ability $e_L = 0$ tried to operate their technology, they would attain output $f(e_L, \cdot, \cdot) = 0$, and attain 0 profit for any rental price. Thus, all of the low-ability individuals will solely supply labor in the competitive equilibrium. Additionally, we have directly that there are enough low-ability individuals that the high-ability individuals are incentivized to operate their technology rather than work – so the profit from the rental price is strictly higher than the equilibrium wage. Conclusion will follow if we can show that the social planner will also prefer to assign all high-ability individuals to technology and all low-ability individuals to labor. We will use an argument similar to proving that a certain strategy is optimal in Game Theory – by assuming that the social planner is following the same allocation, and proving that there are no profits (in fact, there are losses!) from deviation.

   Assume that the social planner assigns all high-ability individuals to technology and all low-ability individuals to labor. First, it is plain to see that they will not improve by moving a low-ability individual to technology – instead of producing marginal output of $f_l'(\cdot) > 0$, they will produce 0 as $e_L = 0$. Next, moving a high-ability individual to labor would have them produce marginal output $f_l'(\cdot)$ instead of $f_k'(\cdot)$. However, recall that in the competitive equilibrium we had that the high-ability individuals were incentivized to always operate their technology, meaning that $r > w$ (at their levels of ability and technology). Since competitive prices are the marginal outputs of each action, this implies that $f_l'(\cdot) < f_k'(\cdot)$, so moving a high-ability individual to labor would reduce output.

   Since we have shown that the social planner will choose the same production decisions as in the competitive equilibrium, they will attain the same output and the same consumption and investment decisions as well. Thus, we can compute the competitive equilibrium by solving the planner's problem.

2. Observe that the production function is homogeneous of degree $1-\nu$ in (k, l):

$$f(e_i, \lambda k, \lambda l) = \lambda^{1-\nu} f(e_i, k, l) \; \forall \; \lambda > 0$$

   Since the production function is homogeneous, we can use a representative consumer for this economy. Formally, if we have a measure 1 of identical firms who each use k_i, l_i, where $\int_0^1 k_i di = K_i$ and $\int_0^1 l_i di = \pi$, then we have that

$$\int_0^1 f(e_i, k_i, l_i) di = e^{\nu} \pi^{(1-\nu)(1-\alpha)} K_t^{\alpha(1-\nu)} = F(K_t)$$

3. The social planner cares equally for the two agents. This means that total output will be divided so as to give the agents the same utility in each period. Thus, maximizing c_H and c_L is the same as maximizing aggregate consumption C_t. Formally, the planner maximizes

$$\int_0^{\infty} e^{-\rho t} \left[\frac{c_H(t)^{1-\sigma}}{1-\sigma} + \pi \frac{c_L(t)^{1-\sigma}}{1-\sigma} \right] dt$$

   subject to

$$\dot{K}(t) = F(K_t) - \delta K_t - (c_H - \pi c_L)$$

   so, since CRRA utility is increasing and monotonic, this is equivalent to just maximizing the sum $c_H + \pi c_L = C_t$.

4. The social planner's (present-value) Hamiltonian is:

$$\mathcal{H} = e^{-\rho t} \frac{C(t)^{1-\sigma}}{1-\sigma} + \lambda(t) \left[e^{\nu} \pi^{(1-\nu)(1-\alpha)} K(t)^{\alpha(1-\nu)} - \delta K(t) - C(t) \right]$$

which admits first order conditions

$$\begin{aligned}
0 &= e^{-\rho t} C(t)^{-\sigma} - \lambda(t) && (C) \\
-\dot{\lambda}(t) &= \lambda(t) \left(\alpha(1-\nu) e^{\nu} \pi^{(1-\nu)(1-\alpha)} K(t)^{\alpha(1-\nu)-1} - \delta \right) && (K) \\
\dot{K}(t) &= e^{\nu} \pi^{(1-\nu)(1-\alpha)} K(t)^{\alpha(1-\nu)} - \delta K(t) - C(t) && (\lambda)
\end{aligned}$$

So differentiating the C condition with respect to t, we get that

$$\frac{\partial \lambda(t)}{\partial t} = \dot{\lambda}(t) = -\rho e^{-\rho t} C(t)^{-\sigma} - \sigma e^{-\rho t} C(t)^{-\sigma-1} \dot{C}(t) \Longrightarrow \dot{\lambda}(t) = e^{-\rho t} C(t)^{-\sigma} \left[-\rho - \sigma \frac{\dot{C}(t)}{C(t)} \right]$$

So, substituting into the K condition, it becomes

$$-\rho + \sigma \frac{\dot{C}(t)}{C(t)} = \alpha(1-\nu) e^{\nu} \pi^{(1-\nu)(1-\alpha)} K(t)^{\alpha(1-\nu)-1} - \delta$$

and rearranging, we get

$$\frac{\dot{C}(t)}{C(t)} = \frac{1}{\sigma} \left[\alpha(1-\nu) e^{\nu} \pi^{(1-\nu)(1-\alpha)} K(t)^{\alpha(1-\nu)-1} + \rho - \delta \right]$$

which we can combine with the first order condition on λ

$$\dot{K}(t) = F(K(t)) - \delta K(t) - C(t)$$

These two ODEs, combined with the initial condition that $K(0) = K_0$, entirely characterize the paths for C and K.

5. If the low-ability agents start with the capital, the equilibrium dynamics will not change. To see why, observe that since using their capital is never profitable for the low-ability agents, they will simply choose labor and then rent their capital to the high-ability agents at a certain rate. To the social planner, this only affects the distribution of consumption between the types of agents, not the production or capital levels. Since we saw in part (3) that the social planner can solve only for aggregate consumption and attain the equilibrium, the distribution within that aggregated consumption won't affect anything. Thus, everything (except for the ratio of c_H to c_L) will remain the same, including the dynamics of $C(t)$ and $K(t)$.

Code

This code takes a long time to run, entirely because of the number of times the model is simulated for Question 5 in Part 1. If I had more time or ability, I could convergence time by using estimates from the previous interest rate. However, I am lazy. The code is:

```julia
using Roots, Plots, StatsBase, LinearAlgebra

# Define parameters
mutable struct Parameters
    R::Float64                # Interest rate (1+r)
    beta::Float64             # Discount factor
    gamma::Float64            # Risk aversion
    P::Matrix{Float64}        # Markov matrix
    y::Vector{Float64}        # Income levels
    assets::Vector{Float64}   # Asset levels
    b::Float64                # Borrowing limit

    """
        Parameters(;
            R::Float64=1.01,
            beta::Float64=0.98,
            gamma::Float64=1.5,
            P::Matrix{Float64}=[0.95 0.05; 0.6 0.4],
            y::Vector{Float64}=[2.0, 0.0],
            assets::Vector{Float64}=collect(range(-0.0, 20.0, length=200)),
            b::Float64=0.0) -> Parameters

    Constructor for the Parameters struct.
    """
    function Parameters(;
            R::Float64=1.01,
            beta::Float64=0.98,
            gamma::Float64=1.5,
            P::Matrix{Float64}=[0.95 0.05; 0.6 0.4],
            y::Vector{Float64}=[2.0, 0.0],
            assets::Vector{Float64}=collect(range(-0.0, 20.0, length=200)),
            b::Float64=0.0)
        return new(R, beta, gamma, P, y, assets, b)
    end
end

"""
    linear_interp(x, grid::Vector{Float64}, values::Vector{Float64}) ->
       Float64

    Linear interpolation helper function.
"""
function linear_interp(x, grid::Vector{Float64}, values::Vector{Float64})
    if x <= grid[1]
        return values[1]
    elseif x >= grid[end]
```

```
        return values[end]
    else
        i = searchsortedfirst(grid, x)
        x_low, x_high = grid[i-1], grid[i]
        y_low, y_high = values[i-1], values[i]
        return y_low + (y_high - y_low) * (x - x_low) / (x_high - x_low)
    end
end

"""
    consumption_interp(m::Float64, z::Int, params, c_policy) -> Float64

    Consumption interpolation helper function.
"""
function consumption_interp(a::Float64, z::Int, params, c_policy)
    # Here, params.assets is the grid for cash-on-hand.
    return linear_interp(a, params.assets, c_policy[:, z])
end

"""
    u_prime(c::Float64, params) -> Float64

    Marginal utility function.
"""
function u_prime(c::Float64, params)
    return c^(-params.gamma)
end

"""
    euler_diff(c::Float64, m::Float64, z::Int, c_policy, params) -> Float64

    Euler equation difference function.
"""
function euler_diff(c::Float64, a::Float64, z::Int, c_policy, params)
    # Prevent nonpositive consumption.
    if c <= 0.0
        return 1e10
    end

    # Compute expected marginal utility next period.
    expect = 0.0
    nY = length(params.y)
    for z_next in 1:nY
        a_next = params.R * (a - c) + params.y[z_next]
        c_next = consumption_interp(a_next, z_next, params, c_policy)
        # Ensure c_next is positive.
        if c_next <= 0.0
            c_next = 1e-8
        end
        expect += params.P[z, z_next] * u_prime(c_next, params)
    end
    # The right-hand side: if the constraint binds, we use u'(a).
    rhs = max(params.beta * params.R * expect, u_prime(a, params))
    return u_prime(c, params) - rhs
end

"""
    K_operator(c_policy, params) -> Vector{Float64}

    K operator
"""
function K_operator(c_policy, params)
    nA = length(params.assets)
    nY = length(params.y)
    c_policy_new = similar(c_policy)
    for i in 1:nA
        a = params.assets[i]
        for z in 1:nY
            if a <= 1e-8
                c_policy_new[i, z] = a
            else
                # Solve for c in the interval [1e-8, m] such that euler_diff =
                    0.
                # We use the bisection (Bisection()) method.
                try
                    sol = find_zero(c -> euler_diff(c, a, z, c_policy, params)
                        , (1e-8, a), Bisection(), atol=1e-8)
                    c_policy_new[i, z] = sol
                catch e
                    # If the solver fails (e.g. due to a sign issue), default
                        to an interior guess.
                    c_policy_new[i, z] = 0.5 * a
                end
            end
        end
    end
    return c_policy_new
end

"""
    solve_model_consumption(params; maxiter::Int=1000, tol::Float64=1e-6) ->
        Vector{Float64}

    Solve the model for consumption.
"""
function solve_model_consumption(params; maxiter::Int=1000, tol::Float64=1e-6,
     verbose::Bool=true)
    nA = length(params.assets)
    nY = length(params.y)

    # Initial guess for consumption: choose an interior guess
    c_policy_old = zeros(nA, nY)
    for i in 1:nA
        a = params.assets[i]
        for z in 1:nY
            c_policy_old[i, z] = a > 1e-8 ? 0.5 * a : a
        end
    end

    diff = 1.0
    iter = 0
    while iter < maxiter && diff > tol
        c_policy_new = K_operator(c_policy_old, params)
        diff = maximum(abs.(c_policy_new .- c_policy_old))
        c_policy_old = c_policy_new
        iter += 1
        if verbose && iter % 25 == 0
            println("Iteration $iter, diff = $diff")
        end
    end

    if verbose && diff > tol
        println("Failed to converge after $maxiter iterations!")
    elseif verbose
        println("Converged in $iter iterations.")
    end

    return c_policy_old
end
"""
    stationary_income(P::Matrix{Float64}) -> Vector{Float64}

    Get stationary distribution for income levels.
"""
function stationary_income(P::Matrix{Float64})
    vals, vecs = eigen(P')
    idx = findall(x -> isapprox(x,1.0; atol=1e-8), vals)
    pi = vecs[:, idx[1]]
    pi = abs.(pi)  # ensure nonnegative
    pi = pi / sum(pi)
    return pi
end

"""
    compute_asset_series(params, c_policy; T)

Simulates a time series of assets of length T (plus the initial asset)
given optimal savings behavior.
"""
function compute_asset_series(params, c_policy; T::Int=500_000)
    nY = length(params.y)
    pi = stationary_income(params.P)
    z_seq = Vector{Int}(undef, T)
    z_seq[1] = sample(1:nY, Weights(pi))
    for t in 2:T
         current_state = z_seq[t-1]
         z_seq[t] = sample(1:nY, Weights(params.P[current_state, :]))
    end

    # Initialize asset series with a[1] = 0.
    a = zeros(T+1)
    for t in 1:T
         # Get the current income state.
         z = z_seq[t]
         # Compute consumption via linear interpolation on the asset grid.
         c = linear_interp(a[t], params.assets, c_policy[:, z])
         # Update assets: a[t+1] = R*(a[t] - c) + income.
         a[t+1] = params.R * (a[t] - c) + params.y[z]
    end
    return a
end


# Question 1, solve the model when r = 0.01 and b = 0.0
params1 = Parameters(R=1.01, b=0.0, assets=collect(range(-0.0, 20.0, length
    =1000)))
c_policy1 = solve_model_consumption(params1)
a_policy1 = params1.assets .- c_policy1

# Plot the results
plot(params1.assets, a_policy1[:,1] .+ params1.y[1], label="a'(High Income)")
plot!(params1.assets, a_policy1[:,2] .+ params1.y[2], label="a'(Low Income)")
plot!(params1.assets, params1.assets, linestyle=:dash, label=false)
xlabel!("Current assets")
ylabel!("Optimal next-period assets")
savefig("macro_hw4_code/savings_policy_function1.png")

plot(params1.assets, c_policy1[:,1], label="c'(High Income)")
plot!(params1.assets, c_policy1[:,2], label="c'(Low Income)")
xlabel!("Current assets")
ylabel!("Optimal consumption")
savefig("macro_hw4_code/consumption_policy_function1.png")

# Question 3, solve the model when gamma = 3
params2 = Parameters(gamma=3.0, assets=collect(range(-0.0, 20.0, length=1000))
    )
c_policy2 = solve_model_consumption(params2)
a_policy2 = params2.assets .- c_policy2

# Plot the results
plot(params2.assets, a_policy2[:,1] .+ params2.y[1], label="a'(High Income)")
plot!(params2.assets, a_policy2[:,2] .+ params2.y[2], label="a'(Low Income)")
```

```julia
plot!(params2.assets, params2.assets, linestyle=:dash, label=false)
xlabel!("Current assets")
ylabel!("Optimal next-period assets")
savefig("macro_hw4_code/savings_policy_function2.png")

plot(params2.assets, c_policy2[:,1], label="c'(High Income)")
plot!(params2.assets, c_policy2[:,2], label="c'(Low Income)")
xlabel!("Current assets")
ylabel!("Optimal consumption")
savefig("macro_hw4_code/consumption_policy_function2.png")

# Question 4, solve the model when b = 4
params3 = Parameters(b=4.0, assets=collect(range(-4.0, 20.0, length=1000)))
c_policy3 = solve_model_consumption(params3)
a_policy3 = params3.assets .- c_policy3

# Plot the results
plot(params3.assets, a_policy3[:,1] .+ params3.y[1], label="a'(High Income)",
   xlims=(0,20), ylims=(0,Inf))
plot!(params3.assets, a_policy3[:,2] .+ params3.y[2], label="a'(Low Income)",
   xlims=(0,20), ylims=(0,Inf))
plot!(params3.assets, params3.assets, linestyle=:dash, label=false)
xlabel!("Current assets")
ylabel!("Optimal next-period assets")
savefig("macro_hw4_code/savings_policy_function3.png")

plot(params3.assets, c_policy3[:,1], label="c'(High Income)", xlims=(0,20),
   ylims=(0,Inf))
plot!(params3.assets, c_policy3[:,2], label="c'(Low Income)", xlims=(0,20),
   ylims=(0,Inf))
xlabel!("Current assets")
ylabel!("Optimal consumption")
savefig("macro_hw4_code/consumption_policy_function3.png")

# Question 5, get aggregate assets in the stationary distribution
max_interest_rate = (1 / params1.beta) - 1.0
rs = collect(range(0.0, max_interest_rate, length=100))
agg_assets = zeros(length(rs))


for (i, r) in enumerate(rs)
    params_temp = Parameters(R=1+r, b=0.0, assets=collect(range(0.0, 20.0,
       length=1000)))
    c_policy_temp = solve_model_consumption(params_temp; verbose=false, tol=1e
       -4)
    a_series = compute_asset_series(params_temp, c_policy_temp, T=250_000)
    agg_assets[i] = mean(a_series)
    println("i = $i, r = $r, agg_assets = $(agg_assets[i])")
end

# Plot aggregate assets vs interest rate.
```

```
plot(rs, agg_assets, marker=:o, xlabel="Interest rate (r)", ylabel="Aggregate
    assets", legend=false)
savefig("macro_hw4_code/aggregate_assets_vs_interest_rate.png")
```