ECON 6130

Problem Set 4

Gabe Sekeres

November 1, 2024

Worked with Omar Andujar on Problem 1.

n.b. Each problem is on a new page (Problem 2 begins on pg. 15, and Problem 3 begins on pg. 18), and each part of Problem 1 is on a new page. Specifically Problem 1 includes a large number of code blocks, which are inserted inline for readability. All code is Julia, and is my own. I've written each function for future use, so it is likely they are not optimized. The totality of the code runs in 7.53 seconds, over 260 value function iterations and 2,000 periods of simulation, which is quick enough for my purposes.

Problem 1. Consider the following problem:

$$\max_{\{c_t\}_{t=0}^{\infty}} \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right]$$

subject to

$$c_t + k_{t+1} = e^{y_t} k_t^{\alpha} + (1-\delta)k_t$$

where y_t is a random process.

1. We have that, defining $\pi(y'|y)$ as the probability of state y' given current state y, and Y as the state space of the Markov chain, preferences are

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(c_t)\right] = \sum_{t=0}^{\infty} \sum_{y_t \in Y} \beta^t \pi(y_t) u(c_t(y^t))$$

where $c_t(y_t) = e^{y_t} k_t^{\alpha} + (1-\delta)k_t - k_{t+1}$. With state variables y and k, as well as control variable k', we get the Bellman equation

$$v(k,y) = \max_{0 \leq k' \leq e^y k^{\alpha} + (1-\delta)k} \left\{ u(e^y k^{\alpha} + (1-\delta)k - k') + \beta \sum_{y'} \pi(y' \mid y) v(k', y') \right\}$$

A sufficient condition that the value function is continuous is that Berge's theorem holds, meaning that the feasible set is upper hemi-continuous and compact-valued, each conditions met trivially, and that the function u is continuous. To ensure that the value function is monotone, we need that the feasible set is further nonempty, which is implied by $\delta \in [0,1]$, and continuous, which is also met by this formulation. Additionally, we need that u is bounded and that $\beta \in (0,1)$, as well as that $e^y k^{\alpha} + (1-\delta)k$ is strictly increasing in k, for which we need $\alpha > 0$. From that, we get that the unique fixed point v is strictly increasing, from the Monotonicity of the Value Function Theorem (SLP Theorem 4.7). Finally, to ensure that v is strictly concave, we need that the value function is strictly concave, which we can meet by fixing $\alpha \in (0,1)$, and noting that the feasible set is already convex. Then, we will have that by the Strict Concavity Theorem (SLP Theorem 4.8) that v is strictly concave.

In summary, in addition to the formulation above, we need that u is continuous and bounded, that $\delta \in [0,1]$, that $\beta \in (0,1)$, and that $\alpha \in (0,1)$. If those conditions are met, the value function will be continuous, monotonically increasing, and concave.

2. To construct the Markov chain, we need the parameters for the persistence of the process, the long-run variance of y_t, the long-run mean of y_t, the distribution of ε_t, and the size of the sample space. All are given except the long-run mean of y_t, but since ε_t has mean 0 and the persistence is less than 1, the long-run mean of y_t will be 0.

   I simulated the Markov chain over 2,000 periods using `Julia`, and found that the long-run mean was 0.059, the serial correlation was 0.992, and the volatility was 0.277. Repeated simulations found similar numbers in that range.

   The Markov chain was:

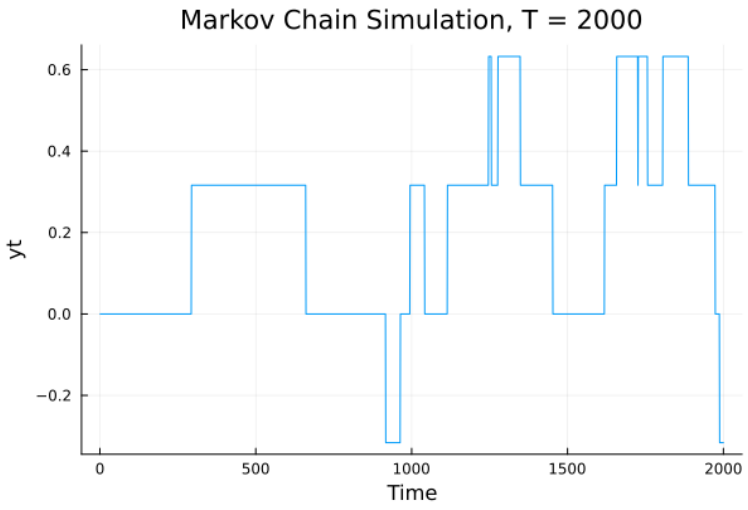

Figure 1: Markov Chain

In order to find these values, I defined the following functions:

```
"""
    tauchen(persistence, lr_var, lr_mean, e_mean, size_space)

Implements Tauchen's method for a size_space-point Markov chain
   approximation
of a persistent process with a given long-run mean and variance, and
normally-distributed noise.

Returns the state space and the transition matrix.

Needs: Distributions

"""
function tauchen(persistence, lr_var, lr_mean, e_mean, size_space)
    @assert size_space % 2 == 1
    # Compute variance of shock
    e_var = (1 - persistence ^ 2) * lr_var

    # Make the state space
    z = collect(range(lr_mean - ((size_space - 1) / 2) * sqrt(lr_var),
       lr_mean + ((size_space - 1) / 2) * sqrt(lr_var), length =
```

```
        size_space))

    # Initialize transition matrix
    P = zeros(size_space, size_space)

    # Compute transition probabilities
    # Note: Currently O(n^2). Doesn't matter for small sizes, but could
       likely be improved to O(n) if more fine grid.
    for i in 1:size_space
        for j in 1:size_space
            if j == 1
                P[i,j] = cdf(Normal(e_mean, sqrt(e_var)), (z[j] -
                   persistence * z[i] + 0.5 * sqrt(lr_var)))
            elseif j == size_space
                P[i,j] = 1 - cdf(Normal(e_mean, sqrt(e_var)), (z[j] -
                   persistence * z[i] - 0.5 * sqrt(lr_var)))
            else
                P[i,j] = cdf(Normal(e_mean, sqrt(e_var)), (z[j] -
                   persistence * z[i] + 0.5 * sqrt(lr_var))) - cdf(Normal
                   (e_mean, sqrt(e_var)),  (z[j] - persistence * z[i] -
                   0.5 * sqrt(lr_var)))
            end
        end
    end

    return z, P
end

"""
    stationary_distribution(P)

Given a transition matrix, return the stationary distribution.

Needs: LinearAlgebra
"""
function stationary_distribution(P)
    # Find the eigenvector corresponding to eigenvalue 1
    vals, vecs = eigen(P')
    stat_dist = vecs[:, argmax(vals)]

    # Normalize the eigenvector
    stat_dist = stat_dist ./ sum(stat_dist)

    # Return the real part
    return real(stat_dist)
end


"""
    simulate_markov_chain(P, z, N, stationary_dist)
```

```
Given a state space and a transition matrix, simulate a Markov chain for N
    periods, starting
from a stationary distribution.

Returns the simulated chain.

Needs: StatsBase

"""
function simulate_markov_chain(P, z, N, stationary_dist)
    # Initialize state vector
    y_state = zeros(Int, N)  # Index of the realization of the variable
    y_val = zeros(N)         # Value of the realization of the variable

    # First period: use stationary distribution
    num = rand()
    cumulative_sum = 0.0
    for j in 1:eachindex(stationary_dist)
        cumulative_sum += stationary_dist[j]
        if num <= cumulative_sum
            y_state[1] = j
            y_val[1] = z[j]
            break
        end
    end

    # Following periods: use transition matrix
    for i in 2:N
        num = rand()  # Random variable drawn from uniform distribution
        cumulative_sum = 0.0
        for j in 1:eachindex(z)
            cumulative_sum += P[y_state[i-1], j]
            if num <= cumulative_sum
                y_state[i] = j
                y_val[i] = z[j]
                break
            end
        end
    end

    return y_val
end
```

And ran the following code, using the packages `Distributions`, `LinearAlgebra`, `StatsBase`, and `Plots`:

```
# Set parameters
persistence = 0.98
lr_var = 0.1
lr_mean = 0.0
e_mean = 0.0
size_space = 7
```

```
N = 2000

# Compute the state space and the transition matrix
z,P = tauchen(persistence, lr_var, lr_mean, e_mean, size_space)

# Compute the stationary distribution
stationary_dist = stationary_distribution(P)

# Simulate the Markov chain
simulated_chain = simulate_markov_chain(P, z, N, stationary_dist)

# Compute long-run mean, serial correeiation, and volatility
long_run_mean = mean(simulated_chain)
serial_correlation = cor(simulated_chain[1:end-1], simulated_chain[2:end])
volatility = std(simulated_chain)

# Output the results:
println("Long-run Mean: $long_run_mean")
println("Serial Correlation: $serial_correlation")
println("Volatility: $volatility")

p = plot(simulated_chain, title="Markov Chain Simulation, T = 2000",
   xlabel="Time", ylabel="yt", legend=false)
display(p)
savefig(p,"/Users/gabe/Dropbox/Notes/Cornell_Notes/Fall_2024/Macro/Julia/
   pset4_markov.png")
```

3. We have that $u(c_t) = \log c_t$, that $\beta = 0.95$, that $\delta = 0.1$, and that $\alpha = 0.35$. First, to appropriately bound the grid, we need to find the steady state capital level. We get the Euler equation:

$$u'(f(k_t) - k_{t+1}) = \beta u'(f(k_{t+1}) - k_{t+2}) f'(k_{t+1})$$

Which becomes

$$\beta(e^{y_{t+1}} \alpha k_{t+1}^{\alpha-1} + (1-\delta))(e^{y_t} k_t^{\alpha} + (1-\delta)k_t - k_{t+1}) = e^{y_{t+1}} k_{t+1}^{\alpha} + (1-\delta)k_{t+1} - k_{t+2}$$

and using the fact that at a steady state $k_{t+1} = k_t$ for all t and that the long-run mean of y_t is 0, we get that

$$\beta(\alpha k^{\alpha-1} + (1-\delta))(k^{\alpha} + (1-\delta)k - k) = k^{\alpha} + (1-\delta)k - k$$

This simplifies to

$$\beta(\alpha k^{\alpha-1} + (1-\delta)) = 1 \implies k^{\star} = \left(\frac{1 - \beta(1-\delta)}{\alpha\beta}\right)^{\frac{1}{\alpha-1}}$$

and in our terms, we have that $k^{\star} \approx 3.585$.

I discretized k across a grid from $0.25k^{\star}$ to $2k^{\star}$, and using a similar value function iteration method as in Problem Set 3, I found the following value and policy functions:

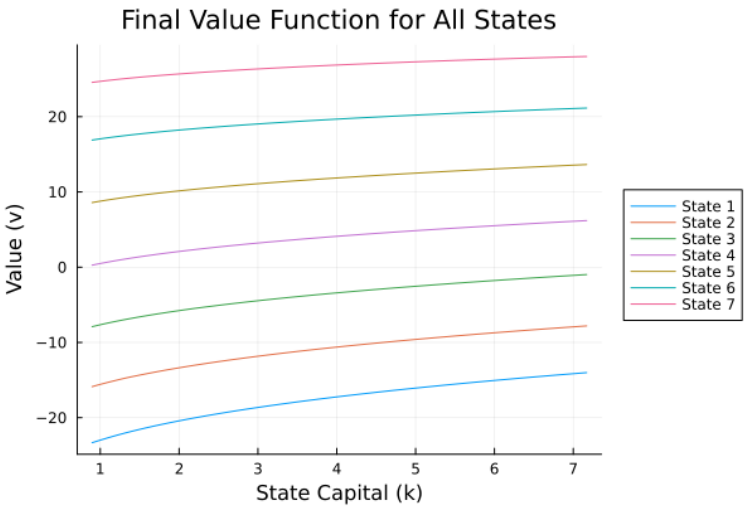

Figure 2: Value Functions

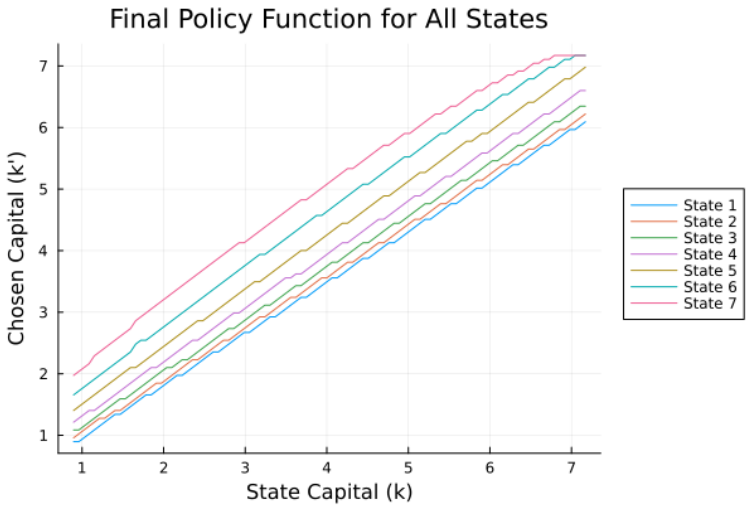

Figure 3: Policy Functions

I defined the following functions in Julia:

```
"""
    log_markov_utility(k_t, y_t, k_next, alpha, beta, v_next, P)

Computes utility for a given level of capital, choice of capital, and
   value function.

Returns calculated utility.
"""
function log_markov_utility(k_t::Float64, y_t::Float64, k_next::Float64,
   alpha::Float64, beta::Float64,
     v_next::Vector{Float64}, P::Vector{Float64})
    # Check feasibility of consumption
    consumption = exp(y_t) * k_t^alpha + (1 - delta) * k_t - k_next

    if consumption <= 0 || k_next <= 0
        return -Inf  # Large penalty for infeasible consumption
    else
        return log(consumption) + beta * sum(P .* v_next)
    end
end

"""
    function value_function_iteration(grid_size, size_space, k_grid, z, P,
        alpha, beta, tol, utility_form)

Takes in parameters, a capital grid, and a functional form for utility,
   and performs the value function iteration.

Returns the stream of value functions, and the indices of the optimal
   policy choices.
"""
```

```
function value_function_iteration(grid_size::Int, size_space::Int, k_grid
    ::Vector{Float64}, z::Vector{Float64},
     P::Matrix{Float64}, alpha::Float64, beta::Float64, tol::Float64,
         utility_form::Function)
    # Initialize value function (2 layers for old and new iteration)
    value = zeros(2, grid_size, size_space)
    sup = 1.0  # To track convergence

    # Initialize streams for value and policy functions
    value_stream = []
    policy_indices_stream = []  # To store the policy indices

    # Objective value for each (i, k, j)
    value_iter = zeros(grid_size, size_space, grid_size)

    while sup >= tol
        # Update the previous value function with the current one
        value[1, :, :] .= value[2, :, :]

        # Compute the new value function and policy extraction
            simultaneously
        policy_indices = zeros(Int, grid_size, size_space)  # To store
            policy (index of k_next)

        for k in 1:size_space
            for i in 1:grid_size
                for j in 1:grid_size
                    # Compute utility and future value
                    value_iter[i, k, j] = utility_form(k_grid[i], z[k],
                        k_grid[j], alpha, beta, value[1, j, :], P[k, :])
                end
                # Maximize over next-period capital (k_grid)
                max_value, max_index = findmax(value_iter[i, k, :])
                value[2, i, k] = max_value  # Store the maximum value
                policy_indices[i, k] = max_index  # Store the index of the
                    maximizing capital choice
            end
        end

        # Push the current value function and policy indices into their
            respective streams
        push!(value_stream, copy(value[2, :, :]))
        push!(policy_indices_stream, copy(policy_indices))

        # Update the sup norm to track convergence
        sup = maximum(abs.(value[2, :, :] - value[1, :, :]))
    end

    return value_stream, policy_indices_stream
end
```

```
"""
    extract_policy(grid_size, size_space, value_stream, k_grid)

Computes the policy that optimized every value function iteration.

Returns the stream of policy functions.
"""
function extract_policy(grid_size::Int, size_space::Int,
   policy_indices_stream::Array{Any,1}, k_grid::Vector{Float64})
    # Store the policy functions (chosen capital values)
    policy_stream = []

    # Iterate over each iteration to extract policy functions based on
       indices
    for iter in 1:length(policy_indices_stream)
        policy = zeros(grid_size, size_space)

        for i in 1:grid_size
            for j in 1:size_space
                # Use the policy index to get the chosen capital from
                   k_grid
                policy[i, j] = k_grid[policy_indices_stream[iter][i, j]]
            end
        end

        # Append the current policy function to the stream
        push!(policy_stream, copy(policy))
    end

    return policy_stream
end


"""
    solve_value_function(persistence, lr_var, lr_mean, e_mean, size_space,
        grid_size, grid_min, grid_max, alpha, beta, tol, utility_form)

Take in parameters for the Tauchen Markov process and the capital grid, as
    well as the problem parameters and a utility functional form.

Return a set of value function and policy function iterations, as well as
   the capital grid.
"""
function solve_value_function(persistence::Float64, lr_var::Float64,
   lr_mean::Float64, e_mean::Float64,
    size_space::Int, grid_size::Int, grid_min::Float64, grid_max::Float64,
        alpha::Float64, beta::Float64,
    tol::Float64, utility_form::Function)
    # Get sample space and transition matrix
    z, P = tauchen(persistence, lr_var, lr_mean, e_mean, size_space)
    # Get capital grid
```

```
    k_grid = collect(LinRange(grid_min, grid_max, grid_size))
    # Perform value function iteration
    value_stream, policy_indices_stream = value_function_iteration(
        grid_size, size_space, k_grid, z, P, alpha, beta, tol,
        utility_form)
    # Extract policy function
    policy_stream = extract_policy(grid_size, size_space,
        policy_indices_stream, k_grid)

    return value_stream, policy_stream, k_grid
end
```

and ran the following code:

```
# Problem Parameters:
N = 100
alpha = 0.35
beta = 0.95
delta = 0.1
tol = 1e-6

# Uncertainty Parameters:
persistence = 0.98
lr_var = 0.1
lr_mean = 0.0
e_mean = 0.0
size_space = 7

# Capital Grid Parameters:
k_ss = 3.585
grid_size = 100
grid_min = 0.25 * k_ss
grid_max = 2 * k_ss

# Utility function:
utility_form = log_markov_utility

value_stream, policy_stream, k_grid = solve_value_function(persistence,
    lr_var, lr_mean, e_mean, size_space, grid_size, grid_min, grid_max,
    alpha, beta, tol, utility_form)




p_final_value = plot()
for state in 1:size_space
    plot!(p_final_value, k_grid, value_stream[end][:, state], label = "
        State $state",
            title = "Final Value Function for All States", xlabel = "State
                Capital (k)", ylabel = "Value (v)")
end
```

```
plot!(p_final_value, legend=:outerright)
display(p_final_value)
savefig(p_final_value, "/Users/gabe/Dropbox/Notes/Cornell_Notes/Fall_2024/
    Macro/Julia/pset4_final_value_function.png")


p_final_policy = plot()
for state in 1:size_space
    plot!(p_final_policy, k_grid, policy_stream[end][:, state], label = "
        State $state",
          title = "Final Policy Function for All States", xlabel = "State
              Capital (k)", ylabel = "Chosen Capital (k')")
end

plot!(p_final_policy, legend=:outerright)
display(p_final_policy)
savefig(p_final_policy, "/Users/gabe/Dropbox/Notes/Cornell_Notes/Fall_2024
    /Macro/Julia/pset4_final_policy_function.png")


# Number of iterations:
iters = length(value_stream)
println("Number of Iterations: $iters")
```

4. Using the final-period value and policy functions calculated in part (3), as well as the simulated random process calculated in part (2), I simulated the process over $N = 2,000$ periods. I sampled the given 0th period capital k_0 from $U(0.25k^\star, 2k^\star)$.

   I found that the standard deviations of logged consumption, investment, and output were 0.342, 0.316, and 0.333 respectively. I found that the correlation between consumption and investment was 0.939, that the correlation between consumption and output was 0.998, and that the correlation between investment and output was 0.961. I also plotted the values for the 2,000 periods:

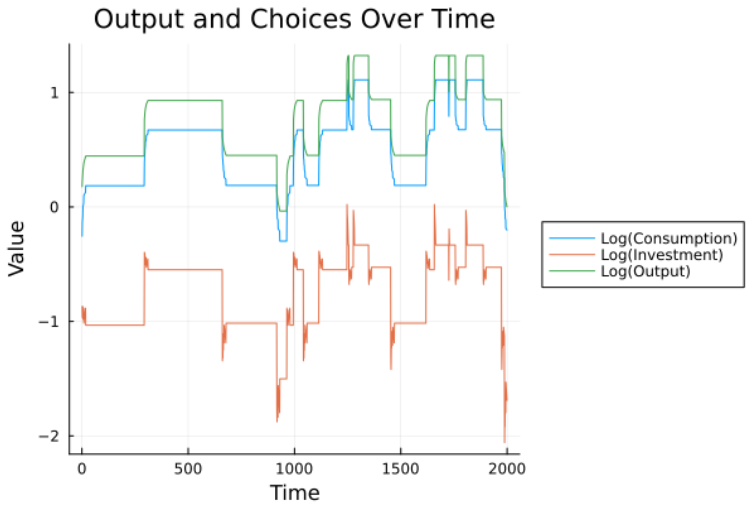

Figure 4: Simulated Economy

Looking at the data on FRED, I combined series for real GDP, real investment, and real consumption in the United States over the last sixteen years. They are presented here:

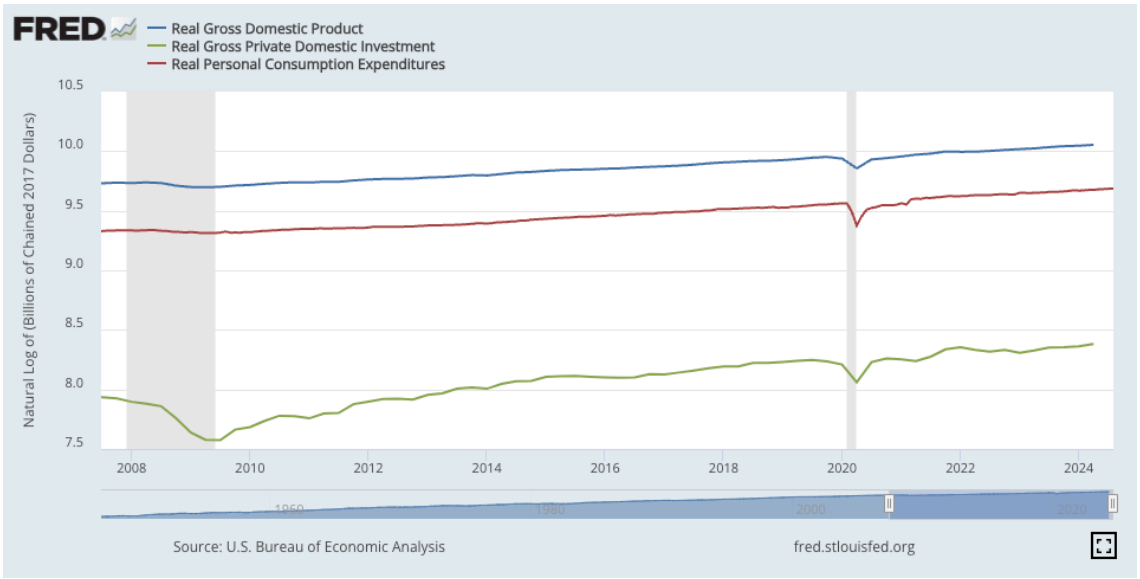

Figure 5: U.S. Economy

The relationships between the lines in the two plots are very similar – consumption and output are extremely correlated, and investment is also correlated but tends to move more.

I used the following code to simulate the economy:

```
# Simulate the converged value function over a large number of periods
N = length(simulated_chain)

# Initialize vectors for storing values:
output = zeros(N-1)
consumption = zeros(N-1)
investment = zeros(N-1)
capital = zeros(N)

# Randomize first-period capital:
capital[1] = rand(k_grid)

for i = 1:(N-1)
    # Set income state
    income_state = argmin(abs.(z .- simulated_chain[i]))

    # Use policy function to determine next-period capital
    capital[i + 1] = policy_stream[end][argmin(abs.(k_grid .- capital[i]))
        , income_state]

    # Compute output, consumption, and investment
    output[i] = exp(simulated_chain[i]) * capital[i] ^ alpha
    consumption[i] = output[i] + (1 - delta) * capital[i] - capital[i + 1]
    investment[i] = capital[i+1] - (1 - delta) * capital[i]
end

# Compute standard deviations of consumption, investment, and log(output)
std_log_consumption = std(log.(consumption))
std_log_investment = std(log.(investment))
std_log_output = std(log.(output))

# Compute correlations
corr_consumption_investment = cor(log.(consumption), log.(investment))
corr_consumption_output = cor(log.(consumption), log.(output))
corr_investment_output = cor(log.(investment), log.(output))

println("Standard deviation of consumption: $std_log_consumption")
println("Standard deviation of investment: $std_log_investment")
println("Standard deviation of output: $std_log_output")

println("Correlation between consumption and investment: $
    corr_consumption_investment")
println("Correlation between consumption and output: $
    corr_consumption_output")
println("Correlation between investment and output: $
    corr_investment_output")

# Plot consumption, investment, and log(output)
```

```
sim_plot = plot()
plot!(sim_plot,log.(consumption), label="Log(Consumption)", title="Output
    and Choices Over Time", xlabel="Time", ylabel="Value")
plot!(sim_plot,log.(investment), label="Log(Investment)", title="Output
    and Choices Over Time", xlabel="Time", ylabel="Value")
plot!(sim_plot,log.(output), label="Log(Output)", title="Output and
    Choices Over Time", xlabel="Time", ylabel="Value")
plot!(sim_plot, legend=:outerright)
display(sim_plot)

savefig(sim_plot,"/Users/gabe/Dropbox/Notes/Cornell_Notes/Fall_2024/Macro/
    Julia/pset4_simulated_economy.png")
```

I ran all of the code for this problem using the following shell file:

```
using Distributions, LinearAlgebra, StatsBase, Plots

@time begin
# Import relevant functions
include("pset4_markov_functions_julia.jl")
include("pset4_value_functions_julia.jl")

# Simulate Markov process:
include("pset4_markov_julia.jl")

# Value Function Iteration:
include("pset4_value_julia.jl")

# Simulate model
include("pset4_growth_sim.jl")

end
```

Problem 2. Consider a neoclassical growth model with two sectors, one producing consumption goods and one producing investment goods.

1. The planner's problem is as follows:

$$\max_{\{K_{C,t},K_{I,t}\}_{t=0}^{\infty}\in\mathbb{R}_+^{\infty}} \sum_{t=0}^{\infty} \beta^t u(C_t)$$

subject to

$$\begin{aligned} K_{C,t+1} &= (1-\delta)K_{C,t} + I_{C,t} \\ K_{I,t+1} &= (1-\delta)K_{I,t} + I_{I,t} \\ L &= L_{C,t} + L_{I,t} \\ C_t &= F(K_{C,t}, L_{C,t}) \\ I_t &= G(K_{I,t}, L_{I,t}) \\ I_t &= I_{C,t} + I_{I,t} \end{aligned}$$

We can reformulate this problem recursively, and get the Bellman equation

$$v(K_{C,t}, K_{I,t}) = \max_{L_{C,t}, I_{C,t}} \{u(C_t) + \beta v(K_{C,t+1}, K_{I,t+1})\}$$

where $K_{C,t}, K_{I,t}$ are the state variables and $L_{C,t}, I_{C,t}$ are the control variables. Using the constraints, we can reformulate this as

$$v(K_{C,t}, K_{I,t}) = \max_{L_{C,t}, I_{C,t}} \{u(F(K_{C,t}, L_{C,t})) + \beta v\left((1-\delta)K_{C,t} + I_{C,t}, (1-\delta)K_{I,t} + G(K_{I,t}, L - L_{C,t}) - I_{C,t}\right)\}$$

2. Taking first order conditions with respect to the control variables $L_{C,t}$ and $I_{C,t}$, we get

$$\frac{\partial v(K_{C,t}, K_{I,t})}{\partial L_{C,t}} = u'(F(K_{C,t}, L_{C,t}))F_2(K_{C,t}, L_{C,t}) - \beta v_2(K_{C,t+1}, K_{I,t+1})G_2(K_{I,t}, L - L_{C,t})$$

and

$$\frac{\partial v(K_{C,t}, K_{I,t})}{\partial I_{C,t}} = \beta v_1(K_{C,t+1}, K_{I,t+1}) - \beta v_2(K_{C,t+1}, K_{I,t+1})$$

Setting them equal to 0, we get the relationships

$$u'(F(\cdot))F_2(\cdot) = \beta v_2(\cdot)G_2(\cdot) \quad \text{and} \quad v_1(\cdot) = v_2(\cdot)$$

Benveniste-Scheinkman gives us that

$$\begin{aligned} v_1(K_{C,t}, K_{I,t}) &= u'(F(K_{C,t}, L_{C,t}))F_1(K_{C,t}, L_{C,t}) + \beta(1-\delta)v_1(K_{C,t+1}, K_{I,t+1}) \\ v_2(K_{C,t}, K_{I,t}) &= \beta(1-\delta+G_1(K_{I,t}, L_{I,t}))v_2(K_{C,t+1}, K_{I,t+1}) \end{aligned}$$

Combining with the first order conditions (w/r/t $I_{C,t}$), we get that

$$u'(F(K_{C,t}, K_{I,t}))F_1(K_{C,t}, L_{C,t}) + \beta(1-\delta)v_1(K_{C,t+1}, K_{I,t+1}) = \beta(1-\delta+G_1(K_{I,t}, L_{I,t}))v_2(K_{C,t+1}, K_{I,t+1})$$

and since we have that $v_1(\cdot) = v_2(\cdot)$ for all t, this becomes

$$u'(F(K_{C,t}, L_{C,t}))F_1(K_{C,t}, L_{C,t}) = v_2(K_{C,t+1}, K_{I,t+1})(\beta G_1(K_{I,t}, L_{I,t}))$$

which implies that

$$v_2(K_{C,t+1}, K_{I,t+1}) = \frac{u'(F(K_{C,t}, L_{C,t}))F_1(K_{C,t}, L_{C,t})}{\beta G_1(K_{I,t}, L_{I,t})}$$

Using the second Benveniste-Scheinkman condition, we get also that

$$\frac{v_2(K_{C,t}, K_{L,t})}{1-\delta+G_1(K_{I,t}, L_{I,t})} = \frac{u'(F(K_{C,t}, L_{C,t}))F_1(K_{C,t}, L_{C,t})}{G_1(K_{I,t}, L_{I,t})}$$

and again using the fact that $v_1(\cdot) = v_2(\cdot)$, we have

$$\frac{v_1(K_{C,t}, K_{L,t})}{1-\delta+G_1(K_{I,t}, L_{I,t})} = \frac{u'(F(K_{C,t}, L_{C,t}))F_1(K_{C,t}, L_{C,t})}{G_1(K_{I,t}, L_{I,t})}$$

Using the first order condition w/r/t $L_{C,t}$, and the fact that this has all not depended on the time period so also holds for $t+1$, this becomes

$$\frac{u'(F(K_{C,t}, L_{C,t}))F_2(K_{C,t}, L_{C,t})}{\beta G_2(K_{I,t}, L_{I,t})(1-\delta+G_1(K_{I,t+1}, L_{I,t+1}))} = \frac{u'(F(K_{C,t+1}, L_{C,t+1}))F_1(K_{C,t+1}, L_{C,t+1})}{G_1(K_{I,t+1}, L_{I,t+1})}$$

So finally, we have the Euler equation:

$$\frac{u'(C_t)F_2(K_{C,t}, L_{C,t})}{G_2(K_{I,t}, L_{I,t})} = \beta(1-\delta+G_1(K_{I,t+1}, L_{I,t+1}))\frac{u'(C_{t+1})F_1(K_{C,t+1}, L_{C,t+1})}{G_1(K_{I,t+1}, L_{I,t+1})}$$

The optimal solution to the planning problem, per Benveniste-Scheinkman, must satisfy thie Euler equation for all levels of t.

3. Recall that from the second Benveniste-Scheinkman condition that

$$v_2(K_{C,t}, K_{I,t}) = \beta(1-\delta+G_1(K_{I,t}, L_{I,t}))v_2(K_{C,t+1}, K_{I,t+1})$$

which implies that, at the steady state, $\beta(1-\delta+G_1(\cdot)) = 1$ for all t. Thus, recalling that at the steady state $U'(C_t) = U'(C_{t+1})$, our Euler equation becomes

$$\frac{F_2(K_C, L_C)}{G_2(K_I, L_I)} = \frac{F_1(K_C, L_C)}{G_1(K_I, L_I)}$$

Additionally, we need that $K = K_I + K_C$, $L = L_I + L_C$, and that

$$K = (1-\delta)K + G(K_I, L_I) \Longrightarrow K = \frac{G(K_I, L_I)}{\delta}$$

These conditions entirely characterize every steady state.

4. We have that $F(K_C, L_C) = K_C^\alpha L_C^{1-\alpha}$ and $G(K_I, L_I) = K_I^\gamma L_I^{1-\gamma}$. This means that $G_1(K_I, L_I) = \gamma\left(\frac{K_I}{L_I}\right)^{\gamma-1}$, so we have that

$$\gamma\left(\frac{K_I}{L_I}\right)^{\gamma-1} = \frac{1}{\beta} - (1-\delta) \Longleftrightarrow \frac{K_I}{L_I} = \left(\frac{1}{\beta\gamma} - \frac{1-\delta}{\gamma}\right)^{\frac{1}{\gamma-1}}$$

We also have that $\frac{F_2(K_C, L_C)}{G_2(K_I, L_I)} = \frac{F_1(K_C, L_C)}{G_1(K_I, L_I)}$, which becomes

$$\frac{(1-\alpha)\left(\frac{K_C}{L_C}\right)^\alpha}{(1-\gamma)\left(\frac{K_I}{L_I}\right)^\gamma} = \frac{\alpha\left(\frac{K_C}{L_C}\right)^{\alpha-1}}{\gamma\left(\frac{K_I}{L_I}\right)^{\gamma-1}} \Longleftrightarrow \frac{\alpha}{1-\alpha}\frac{K_C}{L_C} = \frac{\gamma}{1-\gamma}\frac{K_I}{L_I}$$

Thus, we have that

$$K_C = \frac{1-\alpha}{\alpha}\frac{\gamma}{1-\gamma}\left(\frac{1}{\beta\gamma} - \frac{1-\delta}{\gamma}\right)^{\frac{1}{\gamma-1}} L_C$$

and that

$$K_I = \left(\frac{1}{\beta\gamma} - \frac{1-\delta}{\gamma} \right)^{\frac{1}{\gamma-1}} L_I$$

Since we also have that

$$K = \frac{G(K_I, L_I)}{\delta} \iff K = \frac{K_I^{\gamma} L_I^{1-\gamma}}{\delta} = K_C + K_I$$

We can substitute in the above expressions for K_I and K_C, and since $L_I + L_C = L$ is given, we have two expressions for two unknowns (L_I and L_C), so the rest follows.

Problem 3. Neoclassical growth with externality.

1. Note that since the collective households have unit mass and are identical, the social planner will solve the problem assuming that $c^i = c^j = c$ for all households i, j. Additionally, since they have unit mass, we may assume that $c_t = C_t$ for all t, and that $k_t = K_t$ for all t. Finally, since households do not face any disutility for working, we will assume that $N_t = 1$ for all t. The planner's problem is

$$\max_{\{c_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t U(c_t, C_t)$$

subject to

$$F(k_t, 1) = c_t + k_{t+1} + (1-\delta)k_t \quad \text{and} \quad C_t = \int_0^1 c_t \quad \text{and} \quad k_0 \geq 0 \text{ given}$$

We can represent the social planner's problem recursively, using a Bellman equation where the state variable is k and the control variable is k'. The problem is:

$$v(k_t) = \max_{k_{t+1}} \{U(c_t, C_t) + \beta v(k_{t+1})\}$$

subject to

$$F(k_t) = c_t + k_{t+1} - (1-\delta)k_t$$

where $F(k_t) = F(k_t, 1)$.

2. Taking the first order condition with respect to k_{t+1}, we get that

$$\frac{\partial v(k_t)}{\partial k_{t+1}} = U_1(c_t, C_t) + U_2(c_t, C_t) - \beta v'(k_{t+1}) = 0 \implies \beta v'(k_{t+1}) = U_1(c_t, C_t) + U_2(c_t, C_t)$$

From Benveniste-Scheinkman, we get

$$v'(k_t) = ((1-\delta) + F_1(k_t))(U_1(c_t, C_t) + U_2(c_t, C_t))$$

By combining these two, we get the following Euler equation:

$$U_1(c_t, C_t) + U_2(c_t, C_t) = \beta((1-\delta) + F_1(k_{t+1}))(U_1(c_{t+1}, C_{t+1}) + U_2(c_{t+1}, C_{t+1}))$$

3. A recursive competitive equilibrium is a value function v and policy functions C, G, prices w and r, and the law of motion $\mathcal{H}$ such that the following hold:
   (a) Given w, r, and $\mathcal{H}$, v solves the Bellman equation and C and G are the associated policy functions.
   (b) The pricing functions w and r solve the firm's first order conditions.
   (c) Consistency holds, meaning that $\mathcal{H}(K_t) = G(K_t, K_t)$.
   (d) For all K_t, $C(K_t, K_t) + G(K_t, K_t) - (1-\delta)K_t = F(K_t, 1)$.

4. We have that the household's problem is

$$\max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t U(c_t, C_t)$$

subject to

$$c_t + k_{t+1} - (1-\delta)k_t = w_t + r_t k_t$$

with k_0 given. The Bellman equation is

$$v(k, K) = \max_{c,k'} \{U(c, C) + \beta v(k', K')\}$$

subject to

$$c + k' = w(K) + (1 + r(K) - \delta)k \quad \text{and} \quad F(K, 1) = C + K' - (1 - \delta)K$$

where the state variables are k and K, and the control variables are c and k'. We can take the first order conditions, recalling that the household takes C is given. We get that

$$\frac{\partial v(k, K)}{\partial c} = U_1(c, C) - \beta v'(k') = 0 \Longrightarrow U_1(c, C) = \beta v'(k')$$

From Benveniste-Scheinkman, we get that

$$v'(k) = (1 + r - \delta)U_1(c, C)$$

Combining the two, as in part (2), we get the Euler equation

$$U_1(c, C) = \beta(1 - \delta + r')U_1(c', C')$$

Recalling that in equilibrium, firms take the price as given and demand capital such that the price of capital is equal to the marginal cost, we can say that $r' = F_1(k)$. Thus, our Euler equation is (going back to c_t notation for later comparisons):

$$U_1(c_t, C_t) = \beta(1 - \delta + F_1(k_{t+1}))U_1(c_{t+1}, C_{t+1})$$

5. The unique competitive equilibrium is not Pareto efficient. To see why, recall that the social planner's solution is necessarily Pareto efficient.[1] We can see that the household's Euler equation is different than the social planner's Euler equation, as the social planner internalizes the externality. Since they have Euler equations, they have different solutions and it must therefore be the case that the competitive equilibrium is not Pareto efficient.

[1] If it were not, it would not be a maximum.