Computational Methods HW2

Omer Lurie

May 2025

Exercise 8.1

Show that the solution to the maximization problem

$$\max_{\{c_\nu\}_{\nu=t}^{\infty}} U_t \;=\; \sum_{\nu=t}^{\infty} \beta^{\nu-t}\, u(c_t) \quad \text{s.t.} \quad \sum_{\nu=t}^{\infty} c_\nu = a_t$$

is equal to the solution of the sequential problem

$$\max_{c_t}\Big\{\, u(c_t) \;+\; \beta \max_{\{c_\nu\}_{\nu=t+1}^{\infty}} U_{t+1}\Big\}$$

subject to

$$a_{t+1} = a_t - c_t, \qquad \sum_{\nu=t+1}^{\infty} c_\nu = a_{t+1}.$$

We consider an optimal consumption/saving problem in a model with perfect information. To show that this problem is identical to the sequential solution, we will show that applying Bellman's principle of optimality leads to it. The Lagrangian for the problem is

$$\mathcal{L} = \sum_{\nu=t}^{\infty} \beta^{\nu-t} u(c_\nu) + \lambda \left[a_t - \sum_{\nu=t}^{\infty} c_\nu \right] \tag{I}$$

Leading to the FOCs

$$\frac{\partial \mathcal{L}}{\partial c_\nu} = \beta^{\nu-t}\, u'(c_\nu) \;-\; \lambda \;\; = 0 \qquad \forall \nu \geq t, \tag{II}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = a_t \;-\; \sum_{\nu=t}^{\infty} c_\nu \;=\; 0. \tag{III}$$

by comparing the FOC w.r.t c_t for the periods t and $t+\nu$ and equating λ we obtain

$$u'(c_t) = \beta^{\nu-t} u'(c_\nu) \tag{IV}$$

Which shows that the marginal utility of consumption at t must equal the time-discounted marginal utility of consumption at $t+\nu$.

We now turn to the problem $\max_{\{c_\nu\}_{\nu=t+1}^{\infty}} U_{t+1} \quad s.t. \quad \sum_{\nu=t+1}^{\infty} c_\nu = a_{t+1}$

The Euler equation is given by

$$\beta^{\nu-(t+1)} u'(c_\nu^*) = u'(c_t + 1^*) \quad \text{where} \quad a_{t+1} = \sum_{\nu=t+1}^{\infty} c_\nu^* \tag{V}$$

Giving us the FOC

$$\max_{a_{t+1}} u(a_t - a_{t+1}) + \beta \sum_{\nu=t+1}^{\infty} \beta^{\nu-(t+1)} u(c_\nu^*) \tag{VI}$$

Which balances the loss of utility from consuming less in t against the discounted marginal utility of consumption in future periods.

Using the equality in equation 4 we simplify

$$\begin{aligned} & -u'(c_t^*) \; + \; \beta \sum_{\nu=t+1}^{\infty} \beta^{\nu-(t+1)} \, u'(c_\nu^*) \, \frac{\mathrm{d}c_\nu^*}{\mathrm{d}a_{t+1}} = 0, \\ \Leftrightarrow \quad & -u'(c_t^*) \; + \; \beta \, u'(c_{t+1}^*) \sum_{\nu=t+1}^{\infty} \frac{\mathrm{d}c_\nu^*}{\mathrm{d}a_{t+1}} = 0 \end{aligned} \tag{VII}$$

Since the model is a closed system it holds that

$$da_{t+1} = \sum_{\nu=t+1}^{\infty} dc_\nu^* \quad \Rightarrow \quad 1 = \sum_{\nu=t+1}^{\infty} \frac{dc_\nu^*}{da_{t+1}} \tag{VIII}$$

$$u'(c_t^*) = \beta \, u'(c_{t+1}^*) \tag{IX}$$

Exercise 8.2

Consider an agent who lives for an infinite number of periods indexed by

$$t = 0, 1, \ldots, \infty.$$

In each of the periods he receives a constant income stream w. At each point in time the agent has to decide how much to consume c_t and how much to save for the next period a_t. Negative savings, i.e. debt, are also allowed. Having saved an amount of a_t in period t, the agent receives an amount of $(1+r)a_t$ in the next period. The interest rate therefore is r. His utility should be representable by a discounted additively separable utility specification of the form

$$U_0 \;=\; \sum_{t=0}^{\infty} \beta^t \, \frac{c_t^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}.$$

The agent's optimization problem consequently reads

$$\max_{\{c_t\}_{t=0}^{\infty}} \; U_0 \;=\; \sum_{t=0}^{\infty} \beta^t \, \frac{c_t^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \quad \text{s.t.} \quad \sum_{t=0}^{\infty} \frac{c_t}{(1+r)^t} \;=\; \sum_{t=0}^{\infty} \frac{w}{(1+r)^t}.$$

Calculate the all-in-one solution of this problem.

The Lagrangian is

$$\mathcal{L} = \sum_{t=0}^{\infty} \beta^t \, \frac{c_t^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} - \lambda \left[\sum_{t=0}^{\infty} \frac{c_t}{(1+r)^t} \;-\; \sum_{t=0}^{\infty} \frac{w}{(1+r)^t} \right] \tag{X}$$

Taking FOC w.r.t c

$$\frac{\partial \mathcal{L}}{\partial c_t} = \beta^t c_t^{-\frac{1}{\gamma}} - \frac{\lambda}{(1+r)^t} = 0 \quad \Leftrightarrow \quad \lambda = \beta^t c_t^{-\frac{1}{\gamma}} (1+r)^t \tag{XI}$$

Since λ is constant across time:

$$\beta^t c_t^{-\frac{1}{\gamma}} (1+r)^t = \beta^0 c_0^{\frac{-1}{\gamma}} + (r+1)^0 = c_t^{-\frac{1}{\gamma}} \tag{XII}$$

Yielding

$$c_t^{-\frac{1}{\gamma}} \beta^t (1+r)^t = c_0^{-\frac{1}{\gamma}} \quad \Leftrightarrow \quad c_t = \left[\beta(1+r)\right]^{t\gamma} c_0 \tag{XIII}$$

Which describes any c_t as some function $f(c_0)$. The consumption path is therefore governed by the discount factor β as well as the interest rate $1+r$. Substituting into the budget constraint yields

$$\sum_{t=0}^{\infty} \frac{w}{(1+r)^t} = \sum_{t=0}^{\infty} \frac{\left[\beta(1+r)\right]^{t\gamma} c_0}{(1+r)^t} = c_0 \sum_{t=0}^{\infty} \left[\beta^{\gamma}(1+r)^{\gamma-1}\right]^t \tag{XIV}$$

Solving for c_0

$$c_0 = \frac{w\left[1 + r - \left[\beta(1+r)\right]^{\gamma}\right]}{r} \tag{XV}$$

From the Euler equation we know that

$$\frac{c_t}{c_{t+1}} = (\beta(r+1))^t \Leftrightarrow c_t = c_0(\beta(1+r))^{t\gamma} \tag{XVI}$$

We obtain the all in one solution by substituting c_0 so that

$$c_t = [\beta(1+r)]^{t\gamma} \frac{w\,[1+r-[\beta(1+r)]^{\gamma}]}{r} \tag{XVII}$$

Exercise 8.3

Now write the problem in 2 as a dynamic programming problem. Can you derive a closed-form solution for the policy and value function?

Technical note: Assume that the value function takes the form:

$$V(a) \;=\; B \cdot \frac{\left(a + \frac{w}{r}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}.$$

An analytical solution can be obtained by using a reasonable initial guess iteratively until the function converges. We use the general form

$$V(a) = \max_{c\in(0,a)} \left\{ u(c) + \beta V(a^+) \right\}$$

Where a^+ is obtained from the budget constraint:

$$(1+r)a + w = c + a^+ \leftrightarrow a^+ = (1+r)a + w - c \tag{XVIII}$$

Inserting our earlier results and exploiting constant risk aversion to guess the form of $V(\cdot)$ we get:

$$V(a) = \max_{c\in(0,a)} \left\{ \frac{c^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta B \frac{\left((1+r)a + w - c + \frac{w}{r}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \right\} \tag{XIX}$$

Taking derivatives and solving for c we get

$$\frac{\partial V}{\partial C} = c^{-\frac{1}{\gamma}} - ((1+r)a + w - c + B\beta\frac{w}{r})^{1-\frac{1}{\gamma}} \leftrightarrow c = \frac{(1+r)a + w + \frac{w}{r}}{1 + (\beta B)^{\gamma}} \tag{XX}$$

Inserting the maximum c and solving the value function for B

$$V(a) = \frac{\left(\frac{(1+r)a+w+\frac{w}{r}}{1+(\beta B)^{\gamma}}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta B \cdot \frac{\left((1+r)a + w - \left(\frac{(1+r)a+w+\frac{w}{r}}{1+(\beta B)^{\gamma}}\right) + \frac{w}{r}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

Define the shorthand:

$$R = 1 + r, \quad A = a + \frac{w}{r}$$

to simplify notation:

$$V(a) = \frac{\left(\frac{RA}{1+(\beta B)^{\gamma}}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta B \cdot \frac{\left(RA - \frac{RA}{1+(\beta B)^{\gamma}}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

$$
\begin{aligned}
&= \frac{\left(\frac{RA}{1+(\beta B)^\gamma}\right)^{1-\frac{1}{\gamma}} + \beta B \cdot \left(\frac{(\beta B)^\gamma RA}{1+(\beta B)^\gamma}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \\
&= (RA)^{1-\frac{1}{\gamma}} \cdot \frac{\left(\frac{1}{1+(\beta B)^\gamma}\right)^{1-\frac{1}{\gamma}} + \beta B \cdot \left(\frac{(\beta B)^\gamma}{1+(\beta B)^\gamma}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \\
&= (RA)^{1-\frac{1}{\gamma}} \cdot \frac{(1+(\beta B)^\gamma)^{\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \cdot R^{1-\frac{1}{\gamma}}
\end{aligned}
$$

Next, we solve for B. Using:

$$B = R^{1-\frac{1}{\gamma}} \cdot (1+(\beta B)^\gamma)^{\frac{1}{\gamma}}$$

Raise both sides to γ:

$$
\begin{aligned}
B^\gamma &= R^{\gamma-1} \cdot (1+(\beta B)^\gamma) \\
B^\gamma - R^{\gamma-1} &= \beta^\gamma \cdot B^\gamma \\
B^\gamma(1-\beta^\gamma) &= R^{\gamma-1} \\
B^\gamma &= \frac{R^{\gamma-1}}{1-\beta\gamma} \\
B &= \left(R^{\gamma-1} - \beta^\gamma\right)^{-\frac{1}{\gamma}}
\end{aligned}
$$

Final Value Function:

$$V(a) = B \cdot \frac{A^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} = \left((1+r)^{1-\gamma} - \beta^\gamma\right)^{-\frac{1}{\gamma}} \cdot \frac{\left(a+\frac{w}{r}\right)^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

Inserting B into equation 19 yields the policy function

$$c(a) = \frac{(1+r)(a+\frac{w}{r}}{1+(B\beta)^\gamma} = \frac{(1+r)(a+\frac{w}{r})}{1+(\beta((1+r)^{1-\gamma}-\beta^\gamma)^{-\frac{1}{\gamma}})^\gamma)} = (1+r+(\beta(1+r))^\gamma)\left(a+\frac{w}{r}\right) \quad \text{(XXI)}$$

Now that we have a c as a function of a we insert into the definition of a^+ in equation 18 gives us

$$
\begin{aligned}
a^+(a) &= (1+r)a + w - c(a) \\
&= (\beta(1+r))^\gamma(a+\frac{w}{r}) - \frac{w}{r}
\end{aligned}
\quad \text{(XXII)}
$$

Exercise 8.4

Implement the solution to the problem in 2 using minimization and interpolation. Plot the policy and value function for the parameter combination $\gamma = 0.5$, $\beta = 0.975$, $r = 0.02$, and $w = 1$. In addition, simulate a consumption path for 200 periods. Now change r to 0.03. How does the solution change?

The numerical approach solves the optimization problem faced by an agent who chooses consumption and savings to maximize discounted lifetime utility, subject to an intertemporal budget constraint.

1 The Optimization Problem

The optimization problem is given by:

$$\max_{c_t} \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

subject to :

$$\sum_{t=0}^{\infty} \frac{c_t}{(1+r)^t} = \sum_{t=0}^{\infty} \frac{w}{(1+r)^t}$$

where:

- β: Discount factor
- γ: Coefficient of relative risk aversion (CRRA)
- r: Interest rate on savings
- w: Constant labor income

2 Code Explanation

2.1 Initialization

The arrays for consumption c and value function V are initialized over a grid of asset levels a:

```
gamma, beta, r, w = 0.5, 0.975, 0.02, 1
c = (r * grid_a + w) / 2
V = np.zeros_like(grid_a)
```

Snippet 1: declaring parameters

2.2 Utility Function

The CRRA utility is implemented with a penalty for infeasible (negative) consumption. A similar practice is used to prevent division by 0 in other parts of the code, but I would like to refrain from mentioning it every time it is used:

```
def utility(x_in, i, spline_V):
# Utility function
def utility(x_in, i, spline_V):
    consumption = (1 + r) * grid_a[i] + w - x_in

    if consumption < 1e-10:
        penalty = -1e-10 ** egam / egam * (1 + abs(consumption))
        return penalty

    vplus = spline_eval(x_in, spline_V)
    vplus = max(vplus, 1e-10) ** egam / egam

    return - (consumption ** egam / egam + beta * vplus)
```

Snippet 2: Utility function

2.3 Value Function Iteration

The value function iteration solves for the optimal policy by minimizing the negative utility plus continuation value:

```
res = minimize_scalar(obj, bounds=(lower, upper), method='bounded')
```

Boundary conditions are imposed to ensure numerical stability at the lower grid point:

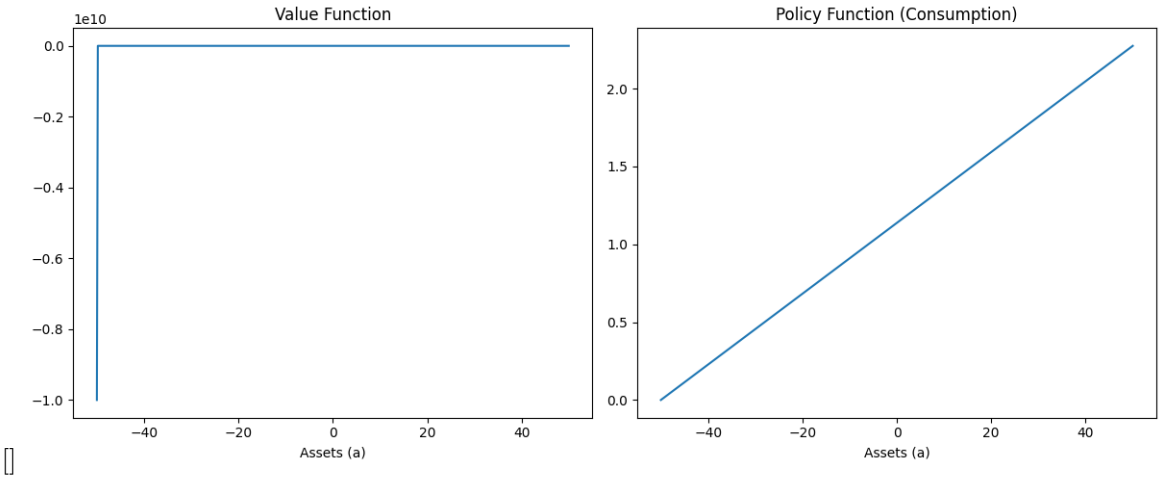
```
c[0] = 0.0
V_new[0] = 1e-10 ** egam / egam
```

Convergence is checked by:

```
con_lev = np.max(np.abs(V_new - V) / np.maximum(np.abs(V), 1e-10))
```

2.4 Policy and Value Function Plots

The resulting policy (optimal consumption) and value function are



[]

Figure 1: Value and policy functions

2.5 Simulation of Consumption Path

Starting with zero initial assets, a consumption path is simulated forward in time:

```
for i in range(200):
a_function[i + 1] = (1 + r) * a_function[i] + w - c_function[i]
```

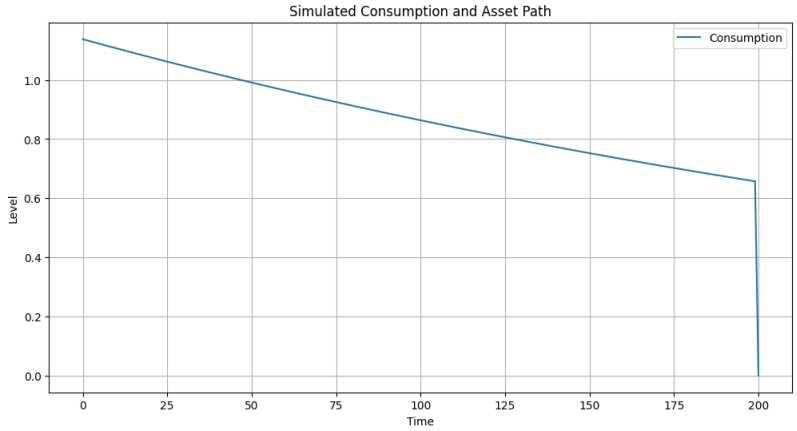


Figure 2: Caption

2.6 Effect of Interest Rate Change

By increasing the interest rate r from 0.02 to 0.03, the agent finds it optimal to save more due to the higher return on assets. This shifts the consumption path downward initially, as more resources are allocated toward future consumption.

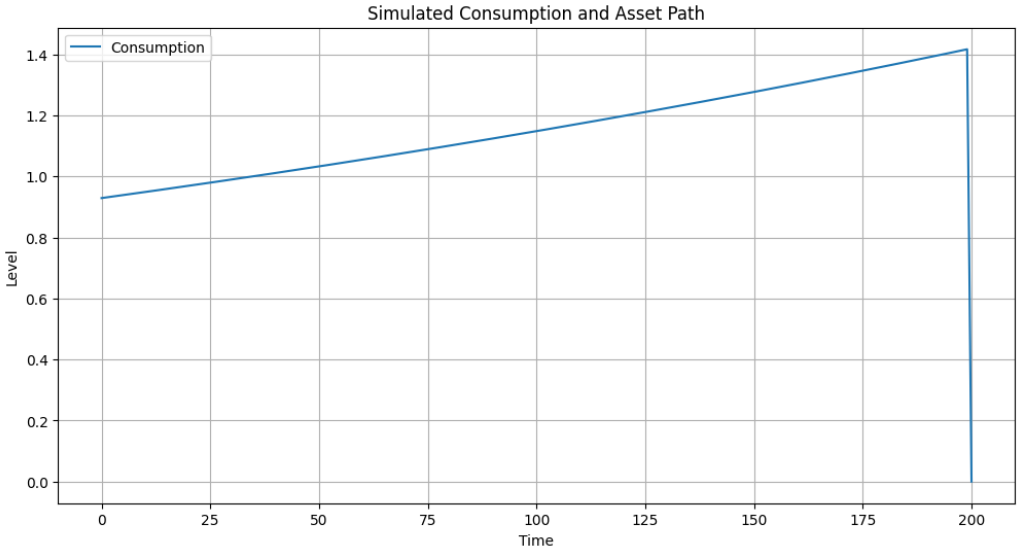


Figure 3: Simulated consumption path with r=.03

Exercise 8.5

2.7 The Optimization Problem

The optimization problem is given by:

$$\max_{c_t} \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

subject to the budget constraint:

$$\sum_{t=0}^{\infty} \frac{c_t}{(1+r)^t} = \sum_{t=0}^{\infty} \frac{w}{(1+r)^t}$$

2.8 First-Order Condition

The utility is the same as discussed in question 8.4, yielding the FOC for optimal consumption is solved using root-finding methods:

```
def foc(x_in, i, spline_c):
    """
    First-order condition for optimal consumption.
    """
    c_plus = spline_c(min(x_in, a_u))

    # Linear extrapolation if x_in > a_u
    if x_in > a_u:
        c_plus += (x_in - a_u) * (c[-1] - c[-2]) / (grid_a[-1] - grid_a
    [-2])

    return (1 + r) * grid_a[i] + w - x_in - (beta * (1 + r)) ** (-gamma) *
    c_plus
```

Snippet 3: First-Order Condition

2.9 Policy Function Iteration

The main iteration loop updates the consumption policy using the FOC:

```
for iteration in range(max_iterations):

    # Boundary condition at a = a_l
    c[0] = 0.0

    # Interpolate consumption function for spline evaluation
    spline_c = InterpolatedUnivariateSpline(grid_a, c, k=3)

    c_new = np.zeros_like(c)

    for i in range(1, len(grid_a)):

        # Initial guess for root finding
        x_guess = (1 + r) * grid_a[i] + w - c[i]

        # Bounds for root finding
        lower = a_l
        upper = min((1 + r) * grid_a[i] + w, a_u + 1e-6)

        # Root finding for FOC
        sol = root_scalar(foc, args=(i, spline_c), bracket=(lower, upper),
   method='brentq')

        if not sol.converged:
            print(f"Root finding failed at iteration {iteration}, i = {i}")

        x_in = sol.root
        c_new[i] = (1 + r) * grid_a[i] + w - x_in

    con_lev = np.max(np.abs(c_new - c) / np.maximum(np.abs(c), 1e-10))
    #print(f"Iteration {iteration+1:4d}, Convergence level: {con_lev:12.8f
   }")

    if con_lev < convergance_tolerance:
        print(f"Converged after {iteration+1} iterations")
        break

    c = c_new.copy()
```

Snippet 4: Policy Function Iteration

2.10 Policy and Value Function Plots

The resulting policy (optimal consumption) and value function are identical to those obtained by minimization:

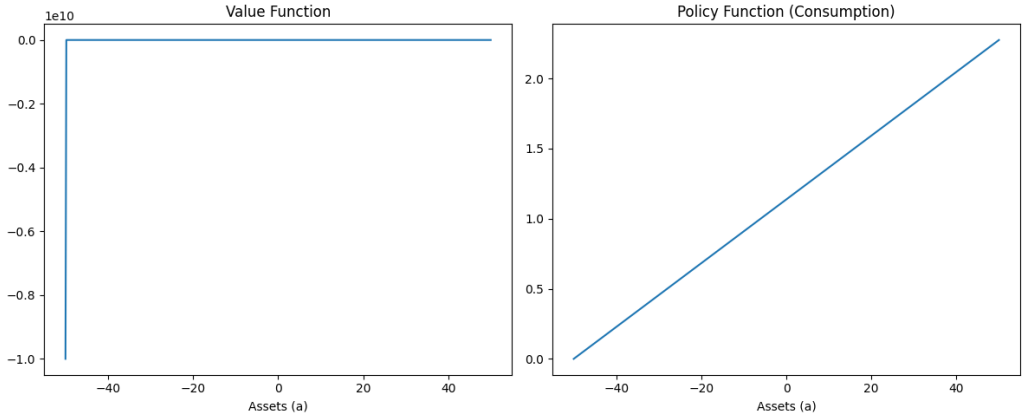


Figure 4: Value and Policy Functions

Exercise 8.6

In the Endogenous Grid Method, we solve the Euler equation to find the optimal consumption policy. As demonstrated in previous sections, solving for c_t is possible but computationally expensive, as $u'(c_t)$ is nonlinear in c_t. However, by inverting $u'(c_t)$ we can solve or c_t once if $u'(c_{t+1})$ is known. Doing so we obtain

$$c_t = (\beta(r+1)u'(c_{t+1}))^{-\frac{1}{\gamma}} \tag{XXIII}$$

This allows us to compute optimal consumption for any period t as long as we know the marginal utility of future consumption without needing to iteratively minimize the function. The inverse marginal utility function derived from the CRRA utility takes the closed form $u'^{-1}(x) = x^{-\frac{1}{\gamma}}$ and is implemented as:

```
def inv_marg_utility(mp):
    return mp ** (-1 / gamma)
```

Snippet 5: Inverse Marginal Utility Function

2.11 Policy Function Iteration with Endogenous Grid Method

The core algorithm is the Endogenous Grid Method. At each iteration, marginal utility for next period consumption is computed, and the Euler equation is inverted to find current consumption. Then the asset grid is adjusted accordingly:

```
for it in range(max_iterations):
    # Interpolated consumption function for next period
    spline_c = np.interp  # using numpy.interp is sufficient

    # Exogenous next-period grid and implied consumption
    grid_aplus = grid_a
    c_plus = np.maximum(spline_c(grid_aplus, grid_a, c, left=c[0], right=c
    [-1]), 1e-12)

    # Compute marginal utility next period
    mu_plus = c_plus ** (-gamma)

    # Invert Euler equation: c_current = (beta * (1+r) * mu_plus)^(-1/gamma
    )
    c_current = inv_marg_utility(beta * (1 + r) * mu_plus)

    # Endogenous current asset grid
    a_endog = (c_current + grid_aplus - w) / (1 + r)

    # Impose borrowing constraint: ensure monotonicity
    a_endog[0] = a_l
    c_current[0] = 0.0

    # Reinterpolate onto exogenous grid
    c_new = np.interp(grid_a, a_endog, c_current, left=0.0, right=c_current
    [-1])
```

```
    # Check for convergence
    diff = np.max(np.abs(c_new - c) / np.maximum(np.abs(c), 1e-10))
    print(f"Iter {it+1:3d}, diff = {diff:.2e}")
    if diff < convergance_tolerance:
        print(f"Converged after {it+1} iterations.")
        c = c_new
        break

    c = c_new
```

Snippet 6: Endogenous Grid Method

Exercise 8.7

The extension allowing borrowing can be introduced into the model by changing the lower limit of the asset grid to $\max\{-\frac{w}{r}, -BorrowingLimit\}$ And iterating over the new grid

```
for iteration in range(max_iterations):

    # Interpolate value function for use in utility evaluation
    spline_V = InterpolatedUnivariateSpline(grid_a, np.maximum(egam * V, 1e
    -12) ** (1 / egam), k=3)

    V_new = np.empty_like(V)

    for i in range(len(grid_a)):

        bounds = (a_l, min((1 + r) * grid_a[i] + w, a_u))

        res = minimize_scalar(lambda x: utility(x, i, spline_V), bounds=
    bounds, method='bounded')

        x_in = res.x
        fret = res.fun

        c[i] = (1 + r) * grid_a[i] + w - x_in
        V_new[i] = -fret

    # Convergence check
    con_lev = np.max(np.abs(V_new - V) / np.maximum(np.abs(V), 1e-10))
    #print(f"Iteration {iteration+1:4d}, Convergence level: {con_lev:12.8f
    }")

    if con_lev < convergance_tolerance:
        print(f"Converged after {iteration+1} iterations")
        break

    V = V_new.copy()
```

Snippet 7: Loop for minimization

2.12 Policy and Value Function Plots

The resulting consumption policy function and corresponding value function are :

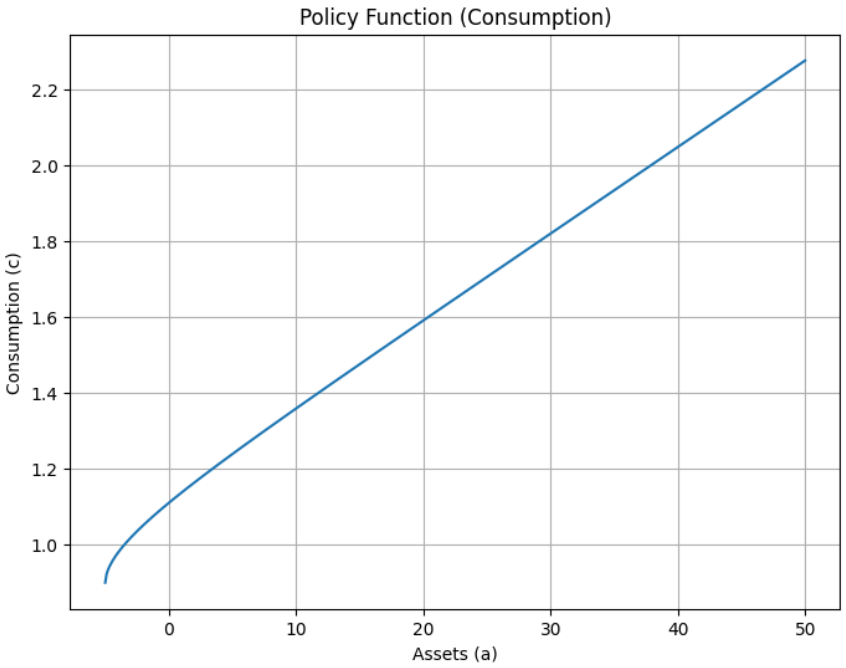


Figure 5: Policy Function with borrowing limit 5

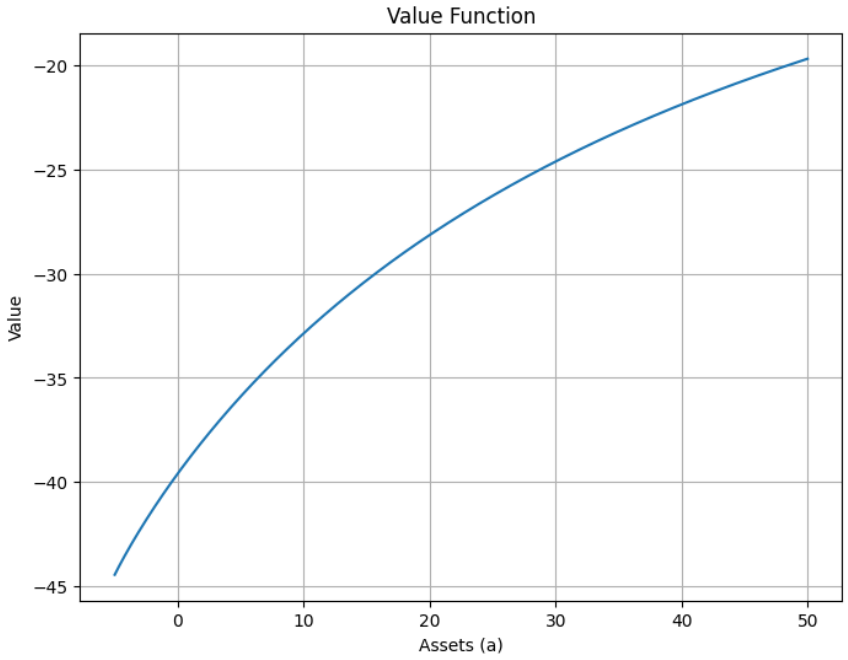


Figure 6: Value Function with borrowing limit 5

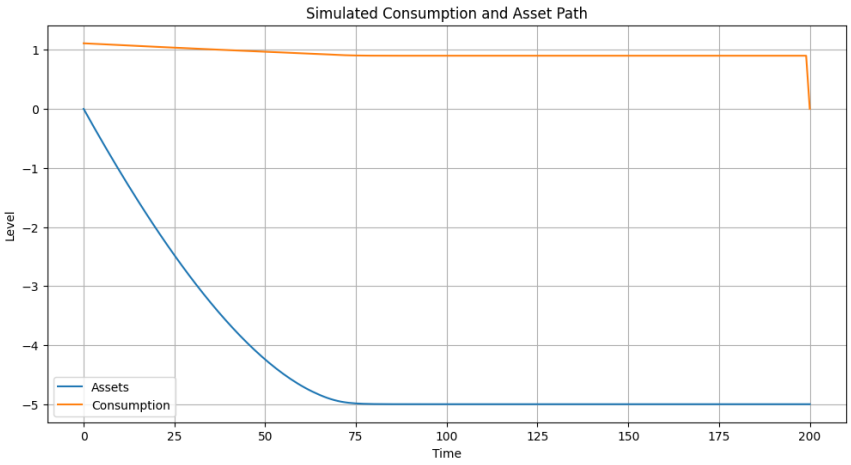


Figure 7: Consumption and Assets with borrowing limit 5

2.13 Effects of Changing Interest Rate and Borrowing Limit

The interest rate r and the borrowing limit govern consumption policy and asset dynamics:

- **Higher Interest Rate** r: An increase in the interest rate makes saving more attractive, encouraging individuals to consume less today in favor of future consumption. This shifts the consumption policy downward for a given asset level and results in faster accumulation of assets in the simulation.
- **Borrowing Limit**: Imposing a borrowing limit restricts how negative the asset position can be. A tighter borrowing constraint (lower borrowing limit) forces the agent to consume less when asset holdings are low and prevents smoothing consumption through borrowing. Conversely, relaxing the borrowing limit allows the agent to borrow more against future income, which flattens the consumption function near lower asset levels.

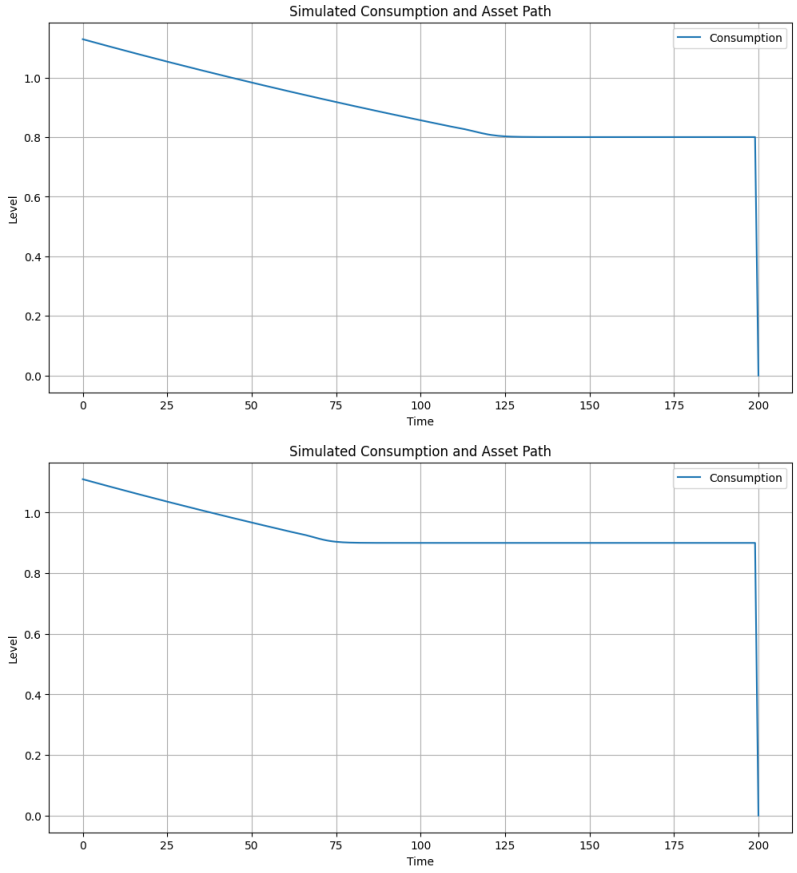


Figure 8: Consumption across borrowing limit levels (10 on upper, 4 on lower)

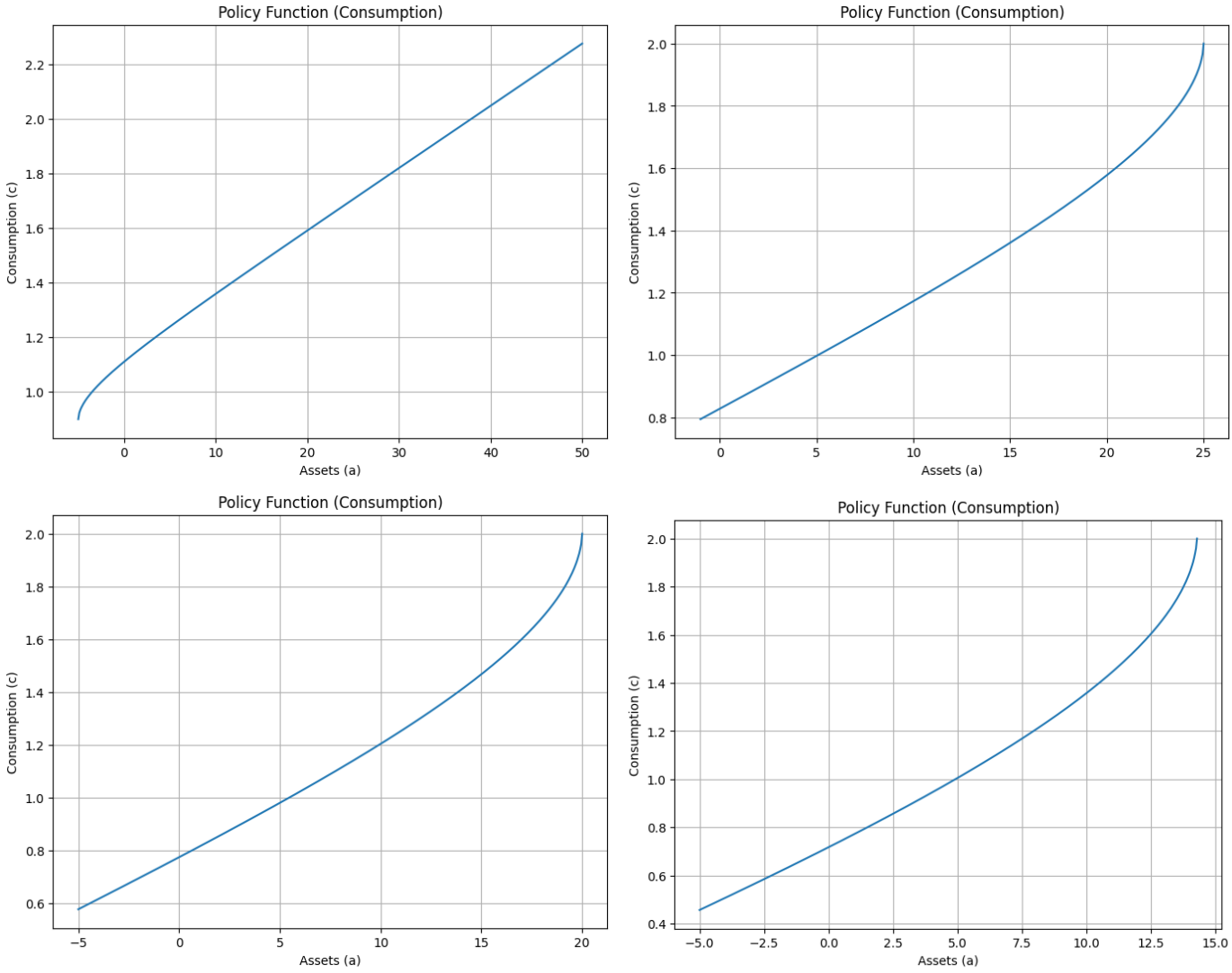


Figure 9: Policy function across r levels (r = 0.02 top left, r = 0.03 top right, r = 0.05 bottom left,r = 0.07 bottom right)

Exercise 8.8

Can you think of a way of implementing the constraint $\bar{a}$ in the root-finding and interpolation solution?

3 Introducing Borrowing Constraint Using Root Finding

To support a root finding approach we solve the Euler equation with the borrowing constraint enforced:

$$u'(c_t) = \beta(1+r)V'(a_{t+1})$$

subject to:

$$a_{t+1} = (1+r)a_t + w - c_t \geq -\bar{a}.$$

Implementation outline:

```
def foc(x_in, i, spline_c):
    """
    First-order condition (FOC) for optimal consumption.
    """
    if x_in <= a_u:
        c_plus = spline_c(x_in)
    else:
        # Linear extrapolation for c_plus beyond the grid
        slope = (c[-1] - c[-2]) / (grid_a[-1] - grid_a[-2])
        c_plus = c[-1] + slope * (x_in - a_u)

    mu_plus = c_plus ** (-gamma)
    return (1 + r) * grid_a[i] + w - x_in - (beta * (1 + r)) ** (-gamma) *
   mu_plus

# Main iteration
for iteration in range(max_iterations):

    # Boundary condition at a = a_l
    c[0] = 0.0

    # Interpolate consumption function
    spline_c = InterpolatedUnivariateSpline(grid_a, c, k=3)

    c_new = np.zeros_like(c)

    for i in range(1, len(grid_a)):

        # Bounds for root finding, respecting borrowing constraint
        lower = max(-abar, a_l)
        upper = min((1 + r) * grid_a[i] + w, a_u + 1e-8)

        # Root finding for FOC
```

```
        sol = root_scalar(foc, args=(i, spline_c), bracket=(lower, upper),
    method='brentq')

        if not sol.converged:
            print(f"Root finding failed at iteration {iteration}, i = {i}")

        x_in = sol.root
        c_new[i] = (1 + r) * grid_a[i] + w - x_in

    con_lev = np.max(np.abs(c_new - c) / np.maximum(np.abs(c), 1e-10))
    #print(f"Iteration {iteration+1:4d}, Convergence level: {con_lev:12.8f
    }")

    if con_lev < convergance_tolerance:
        print(f"Converged after {iteration+1} iterations")
        break

    c = c_new.copy()
```

Snippet 8: Root Finding with Borrowing Limit

When the optimal a_{t+1} implied by the Euler equation is below $-\bar{a}$, the borrowing limit binds, and consumption is adjusted accordingly:

$$c_t = (1+r)a_t + w + \bar{a}.$$

4 Comparison: Minimization vs. Root-Finding with Borrowing Constraints

4.1 Minimization Approach

To use Scipy's minimize we define the objective function as :

$$\min_{a_{t+1}} \left\{ -\left(\frac{[(1+r)a_t + w - a_{t+1}]^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta V(a_{t+1}) \right) \right\}$$

And evaluate it over a bounded interval:

$$a_{t+1} \in [-\bar{a}, a_{\max}]$$

where the lower bound directly enforces the borrowing limit. Minimization is stable and robust, but it tends to be computationally intensive.

4.2 Root-Finding Approach

Unlike minimization, the root-finding method targets the Euler equation directly:

$$u'(c_t) = \beta(1+r)V'(a_{t+1})$$

By defining a function in terms of the consumption c_t:

$$f(c_t) = c_t^{-\gamma} - \beta(1+r)V'(a_{t+1}(c_t)),$$

we solve $f(c_t) = 0$ where $a_{t+1}(c_t) = (1+r)a_t + w - c_t$. The borrowing constraint is enforced by restricting the root-finding interval:

$$c_t \leq (1+r)a_t + w + \bar{a}$$

This method's main advantage is that it does not require repeated function evaluation, as it exploits the structure of the Euler equation for optimization. As a result, it will fail with ill behaved objective functions, or if its constraints bind it before it is able to find roots.

4.3 Summary of Differences

Feature	**Minimization**	**Root-Finding**
What is solved	value function	Euler equation
constraint	binds the minimizer domain	binds the root-finding interval
Numerical robustness	Robust	sensitive to binding constraints
Convergence behavior	Slower, requires good interpolation of $V(a')$	Faster, but may fail near bounds
When constraint binds	Returns optimal constrained consumption via objective minimum	May not find root, may require a hardcoded failsafe

Table 1: Comparison of Minimization and Root-Finding Approaches

Exercise 8.9

Can you think of a way of implementing the constraint $\bar{a}$ in the solution that uses the method of endogenous gridpoints? The original article by Carroll (2006) might help you here.

As discussed in Carroll (2006), a possible way to solve the model with a borrowing limit using the method of endogenous gridpoints would be to explicitly add the point $a = -\bar{a}$ to the grid. at this point the agent would consume $(1+r)(-\bar{a}) + w + \bar{a} = w - r\bar{a}$, corresponding to the agent consuming all of their income after paying interest on their borrowing. During interpolation, all $a < \bar{a}$ are simply replaced by the explicitly defined bounded value so that

$$c(a) = \begin{cases} w + (1+r)a, & \text{if } a \geq -\bar{a} \text{ and the constraint does not bind,} \\ w - r\bar{a}, & \text{if } a = -\bar{a} \text{ (constraint binds).} \end{cases}$$

The code implementation for this approach would be

```
for iteration in range(max_iterations):
# Step 1: compute c_plus from current policy guess
spline_c = interp1d(grid_a, c, fill_value="extrapolate")
c_plus = spline_c(grid_a)

# Step 2: compute marginal utility and invert Euler equation
mu_plus = np.maximum(c_plus, 1e-12) ** (-gamma)
c_current = (beta * (1 + r) * mu_plus) ** (-1 / gamma)

# Step 3: endogenous asset grid implied by budget constraint
a_endog = (c_current + grid_a - w) / (1 + r)

# Step 4: Handle constrained region explicitly
# Where endogenous asset < borrowing limit     use corner solution
constrained = a_endog < a_l
unconstrained = ~constrained

# Consumption at the borrowing constraint
c_constraint = (1 + r) * grid_a[constrained] + w - a_l

# Combine constrained and unconstrained parts for interpolation
if np.any(unconstrained):
# Normal case: mix constrained and unconstrained
    a_combined = np.concatenate(([a_l], a_endog[unconstrained]))
    c_combined = np.concatenate(([ (1 + r) * a_l + w - a_l ], c_current
[unconstrained]))
else:
# Extreme case: everything is constrained     use only constrained
consumption
    a_combined = np.array([a_l, a_u])
    c_val = (1 + r) * a_l + w - a_l
    c_combined = np.array([c_val, c_val])  # constant consumption at
borrowing limit
```

```
    # Step 5: interpolate consumption back to original grid
    c_new = np.interp(grid_a, a_combined, c_combined, left=c_combined[0],
    right=c_combined[-1])

    # Step 6: check convergence
    diff = np.max(np.abs(c_new - c) / np.maximum(c, 1e-8))
    if diff < convergance_tolerance:
        print(f"Converged after {iteration+1} iterations.")
        break

    c = c_new.copy()
else:
    print("Did not converge.")
```

Snippet 9: endogenous gridpoints Borrowing Limit

Where step 4 distinguishes between the cases of the constraints binding or not binding

```
    # Step 4: Handle constrained region explicitly
    # Where endogenous asset < borrowing limit, use corner solution
    constrained = a_endog < a_l
    unconstrained = ~constrained
```

Snippet 10: Separating cases

defines consumption at the boundaries

```
    # Consumption at the borrowing constraint
    c_constraint = (1 + r) * grid_a[constrained] + w - a_l
```

Snippet 11: Binding consumption

and interpolates both cases back into the original grid.

```
    # Combine constrained and unconstrained parts for interpolation
    if np.any(unconstrained):
    # Normal case: mix constrained and unconstrained
        a_combined = np.concatenate(([a_l], a_endog[unconstrained]))
        c_combined = np.concatenate(([ (1 + r) * a_l + w - a_l ], c_current
    [unconstrained]))
    else:
    # Extreme case: everything is constrained, use only constrained
    consumption
        a_combined = np.array([a_l, a_u])
        c_val = (1 + r) * a_l + w - a_l
        c_combined = np.array([c_val, c_val])  # constant consumption at
    borrowing limit
```

Snippet 12: endogenous gridpoints Borrowing Limit