A Simple Representative Growth Model

Lecturer: Bo Li TA: Chen Gao

2026-03-17

Table of contents

1 A Simple Representative Growth Model

1.1 Brock and Mirman (1972)

- The social planner solves the following problem:

$$\max_{\{c_t, k_{t+1}\}} \sum_{t=0}^{\infty} \beta^t \log(c_t)$$
$$c_t + k_{t+1} \leq A k_t^{\alpha}$$
$$k_0 \text{ given}, c_t, k_t \geq 0 \text{ for all } t$$

- The controls are c_t and k_{t+1}, and the only state is k_t. We can reformulate the previous problem as a dynamic programming problem as follows:

$$V(k) = \max_{k'} \{\log(Ak^{\alpha} - k') + \beta V(k')\}$$

- If we solve for capital stock policy rule $k' = g_k(k)$, we can obtain the policy function for consumption $c = g_c(k)$ directly from the resource constraint:

$$g_c(k) = Ak^{\alpha} - g_k(k)$$

1.2 Step 1: Initialization of the parameters

```
import numpy as np
import matplotlib.pyplot as plt
import time

# Model Parameters
alpha = 0.4
A = 5
beta = 0.9888

# Algorithm Parameters
T = 20  # periods to simulate
tolv = 1e-8  # tolerance
```

1.3 Steady State

To compute the steady state, we use the equilibrium conditions:

$$\frac{1}{c} = \beta \frac{1}{c'} \left(\alpha A k'^{\alpha - 1} \right)$$
$$c + k' = A k^{\alpha}$$

Imposing $x = x'$ for all variables, we get:

$$k_{ss} = \left(\frac{1}{\beta \alpha A} \right)^{\frac{1}{\alpha - 1}}$$
$$c_{ss} = A k_{ss}^{\alpha} - k_{ss}$$
$$y_{ss} = A k_{ss}^{\alpha}$$

The steady state is given by:

```
k_ss = (1 / (beta * alpha * A)) ** (1 / (alpha - 1))
y_ss = A * k_ss**alpha
c_ss = y_ss - k_ss
ku = (1 / A) ** (1 / (alpha - 1))  # if c = 0

print("k_ss=", k_ss)
print("k_u=", ku)
```

```
k_ss= 3.1157606561253433
k_u= 14.62008869106433
```

1.4 Step 2: Creation of a grid for the states

- Create a vector for k, with l_k values i.e. $k \in [k_1 < k_2 < ... < k_{l_k}]$.
- The larger l_k, the better the approximation, but this increases the computing time.
- We can have the grid from 0 to k_{l_k}, or around the steady state, e.g. $\pm 10\%$ of k_{ss}.

. . .

```
lk = 500
k_min = 0.5 * k_ss
k_max = 1.5 * k_ss
k = np.linspace(k_min, k_max, num=lk)

print(k[:5])
```

```
[1.55788033 1.56412434 1.57036835 1.57661236 1.58285637]
```

1.5 Step 3: Computation of the return function

- For each value of k and k' in the grid we need:

$$c = Ak^{\alpha} - k'$$
$$U(c) = \log(c)$$

- U is of dimension $g_k \times 1$, where $g_k = l_k \times l_k$.
- If $l_k = 2$, U is of dimension 4×1:

$$\begin{array}{c} \\ k \end{array} \begin{array}{c} \leftarrow k' \longrightarrow \\ \begin{bmatrix} c_{11} \\ c_{12} \\ c_{21} \\ c_{22} \end{bmatrix} \end{array}$$

$$c_{ij} = Ak_i^{\alpha} - k_j \quad \text{for } i = 1 : l_k,\ j = 1 : l_k$$

```
gk = lk * lk  # dimension of U

c = np.zeros((gk))
for i in range(lk):
    for j in range(lk):
        c[i * lk + j] = A * (k[i] ** alpha) - k[j]
```

```
        if c[i * lk + j] < 0:
            c[i * lk + j] = tolv
print(c)
```

```
[4.41224766 4.40600365 4.39975964 ... 4.60356373 4.59731972 4.59107571]
```

> **Note**
>
> Here we use **index packing**: the combination `i * lk + j` converts 2D indices `(i, j)` into a unique 1D index.
>
> - This technique allows us to map a 2D grid (all possible pairs (k_i, k_j)) into a 1D array
> - which is often needed for vectorized operations or storage efficiency in numerical computing

. . .

Finally, to calculate the return function, we simply take the log of the consumption vector:

```
u = np.log(c)
print(u)
```

```
[1.48438423 1.48296808 1.48154991 ... 1.52683073 1.52547346 1.52411436]
```

1.6 Step 4: Initialization and computation of the optimal value function

- Our candidate value function is an $l_k \times 1$ vector:

$$V(k) = \begin{bmatrix} V(k_1) \\ V(k_2) \\ \vdots \\ V(k_{l_k}) \end{bmatrix}$$

- We can initialize V_0 with a vector of zeros of dimension l_k.
- We also need to initialize the contraction mapping operator $V_1 = T(V_0)$.
- In iteration n, it returns:

$$V_n(k_i) = \max_{k'} \left\{ \begin{bmatrix} U(Ak_i - k'_1) \\ \vdots \\ U(Ak_i - k'_{l_k}) \end{bmatrix} + \beta \begin{bmatrix} V_{n-1}(k'_1) \\ \vdots \\ V_{n-1}(k'_{l_k}) \end{bmatrix} \right\}$$

1.7 Value Function Iteration

In each iteration:

- Set V_0 to be the V_1 computed from the previous iteration.
- For each k_i, compute $U(c) + \beta V_0(k')$ and choose the maximum over k' and call it V_1 (Use the max operator).
- Compare V_0 with V_1 and, if the difference is sufficiently small, stop (Use the norm of $V_1 - V_0$).

. . .

```
# Initialize value function and operator and iterate
V0 = np.ones((lk))
V1 = np.zeros((lk))
start_time = time.time()
while abs(np.linalg.norm(V1 - V0)) > tolv:
    V0 = np.copy(V1)  # Use deepcopy here to avoid shallow copy.
    for i in range(lk):
        vtemp = u[i * lk : (i + 1) * lk] + beta * V0
        V1[i] = np.max(vtemp)

print(V1[:5])
print(f"Time taken: {time.time() - start_time:.2f} seconds")
```

```
[138.88421524 138.88686227 138.88949887 138.89212395 138.89474037]
Time taken: 1.66 seconds
```

1.8 Convergence Criteria: Norm Choices

When checking convergence of value function iteration, we often monitor

$$\|V_{n+1} - V_n\| \leq \texttt{tol}.$$

Different norms imply different (sometimes stronger/weaker) stopping rules.

1.8.1 1) Sup-norm (ℓ_∞): the standard for contraction mappings

$$\|x\|_\infty = \max_i |x_i|.$$

- **Interpretation:** the largest pointwise change in the value function.
- **Why common:** Bellman operators are contractions in $\|\cdot\|_\infty$ under standard conditions.

```
diff = V1 - V0
err_inf = np.max(np.abs(diff))  # ||V1 - V0||_inf
# or: err_inf = np.linalg.norm(diff, ord=np.inf)
print(err_inf)
```

```
4.4568082557816524e-10
```

1.8.2 2) Euclidean norm (ℓ_2): average-magnitude change

$$|x|_2 = \left(\sum_i x_i^2\right)^{1/2}.$$

- **Interpretation:** overall magnitude of the change.
- **Caution:** small ℓ_2 does not rule out a large error at a single grid point.

```
diff = V1 - V0
err_2 = np.linalg.norm(diff, ord=2)  # ||V1 - V0||_2
# same as: err_2 = np.sqrt(np.sum(diff**2))
print(err_2)
```

```
9.96425562765096e-09
```

1.8.3 3) ℓ_1 norm: total absolute change

$$|x|_1 = \sum_i |x_i|.$$

- **Interpretation:** aggregate absolute change across all grid points.
- **Caution:** depends on grid size (more points ⟶ larger sum), so tolerances are not comparable across lk.

```
diff = V1 - V0
err_1 = np.linalg.norm(diff, ord=1)  # ||V1 - V0||_1
# same as: err_1 = np.sum(np.abs(diff))
print(err_1)
```

```
2.2280752887127164e-07
```

1.8.4 4) Relative norms: scale-free stopping rule

Absolute errors depend on the scale of V. A relative rule is

$$\frac{|V_{n+1} - V_n|}{\max(1, |V_n|)} \leq \texttt{tol}.$$

```
diff = V1 - V0
den = max(1.0, np.linalg.norm(V0, ord=np.inf))
rel_err_inf = np.max(np.abs(diff)) / den
print(rel_err_inf)
```

```
3.192300058330576e-12
```

1.8.5 5) Weighted norm (optional): emphasize important regions

Given weights $w_i > 0$,

$$|x|_{w,\infty} = \max_i \frac{|x_i|}{w_i}.$$

Useful if some grid points matter more (e.g., near the ergodic distribution).

```
diff = V1 - V0
w = np.ones_like(diff)  # replace with your weights (must be > 0)
err_winf = np.max(np.abs(diff) / w)
print(err_winf)
```

```
4.4568082557816524e-10
```

> **Tip**
>
> **Practical default:** use the sup-norm (`np.max(np.abs(V1 - V0))`) for VFI convergence checks, because it matches the standard contraction-mapping theory.

1.9 Step 5: Computation of the optimal policy functions

- To calculate $g_k(k)$, for each point in the grid, find the index where $U(c) + \beta V^*(k')$ takes its maximum value when V^* is used.
- The element `optim[i]` gives the index of the k' maximizer when the initial capital is k_i.

. . .

```
optim = np.zeros(lk, dtype="int")  # "int" type for index.
for i in range(lk):
    vtemp = u[i * lk : (i + 1) * lk] + beta * V0
    optim[i] = np.argmax(vtemp)
print(optim[:5])
```

```
[129 129 130 130 131]
```

- The policy functions are given by:

$$\text{polk} = \begin{bmatrix} g_k(k_1) \\ g_k(k_2) \\ \vdots \\ g_k(k_{l_k}) \end{bmatrix}, \text{polc} = \begin{bmatrix} g_c(k_1) \\ g_c(k_2) \\ \vdots \\ g_c(k_{l_k}) \end{bmatrix} \quad \text{and} \quad \text{polc} = A \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_{l_k} \end{bmatrix}^{\alpha} - \begin{bmatrix} g_k(k_1) \\ g_k(k_2) \\ \vdots \\ g_k(k_{l_k}) \end{bmatrix}$$

```
polk = k[optim]
polc = A * k**alpha - polk

print(polk[:5])
print(polc[:5])
```

```
[2.36335753 2.36335753 2.36960154 2.36960154 2.37584555]
[3.60677046 3.61633032 3.6196233  3.62913757 3.63238525]
```

1.10 Testing our approximation

The analytical solution to the model is:

$$v(k) = (1-\beta)^{-1}\Big[\ln(A(1-\alpha\beta)) + \frac{\alpha\beta}{1-\alpha\beta}\ln(A\alpha\beta)\Big] + \frac{\alpha}{1-\alpha\beta}\ln(k)$$
$$k' = g_k(k) = \alpha\beta A k^{\alpha}$$
$$c = g_c(k) = (1-\alpha\beta)Ak^{\alpha}$$

To test our approximation, we compare the differences between the analytical and numerical solution.

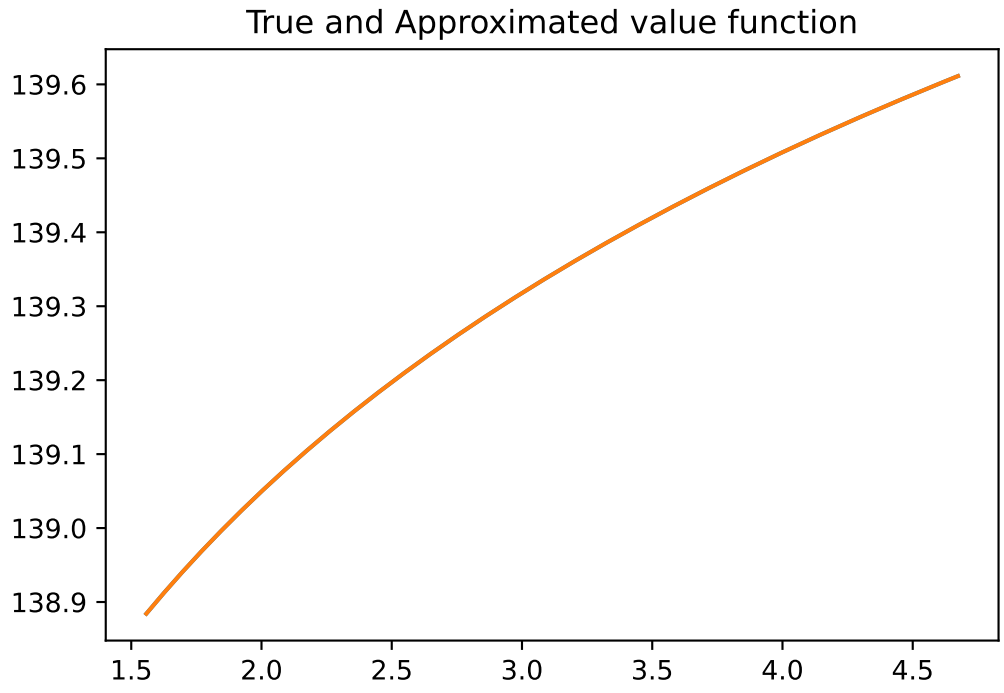
```
# True value functions
a = (1 / (1 - beta)) * (
    np.log(A * (1 - alpha * beta))
    + alpha * beta * np.log(A * alpha * beta) / (1 - alpha * beta)
)
b = alpha / (1 - alpha * beta)
Vk = a + b * np.log(k)
dist = np.linalg.norm(Vk - V0)
print(f"Distance between true and approx. value function: {dist:.8f}")
```

```
Distance between true and approx. value function: 0.00038753
```

> **Note**
>
> **Why is the distance between the true and approximated value function not zero?**
> The main reason is that the optimal policy function is not exact on the grid. If we increase the grid size, the distance will decrease.
> Also, if we use a smaller tolerance, the distance will decrease.

Plot the true and approximated value functions:

```
plt.figure(figsize=(6, 4))
plt.plot(k, V0)
plt.plot(k, Vk)
plt.title("True and Approximated value function")
plt.show()
```



1.11 Simulating the model

1. Choose an initial value for k_0. To do this, select a number between 1 and I_k, indicating the position in the initial capital in the grid. Given this position, we use the capital policy function to extract the optimal first period capital.

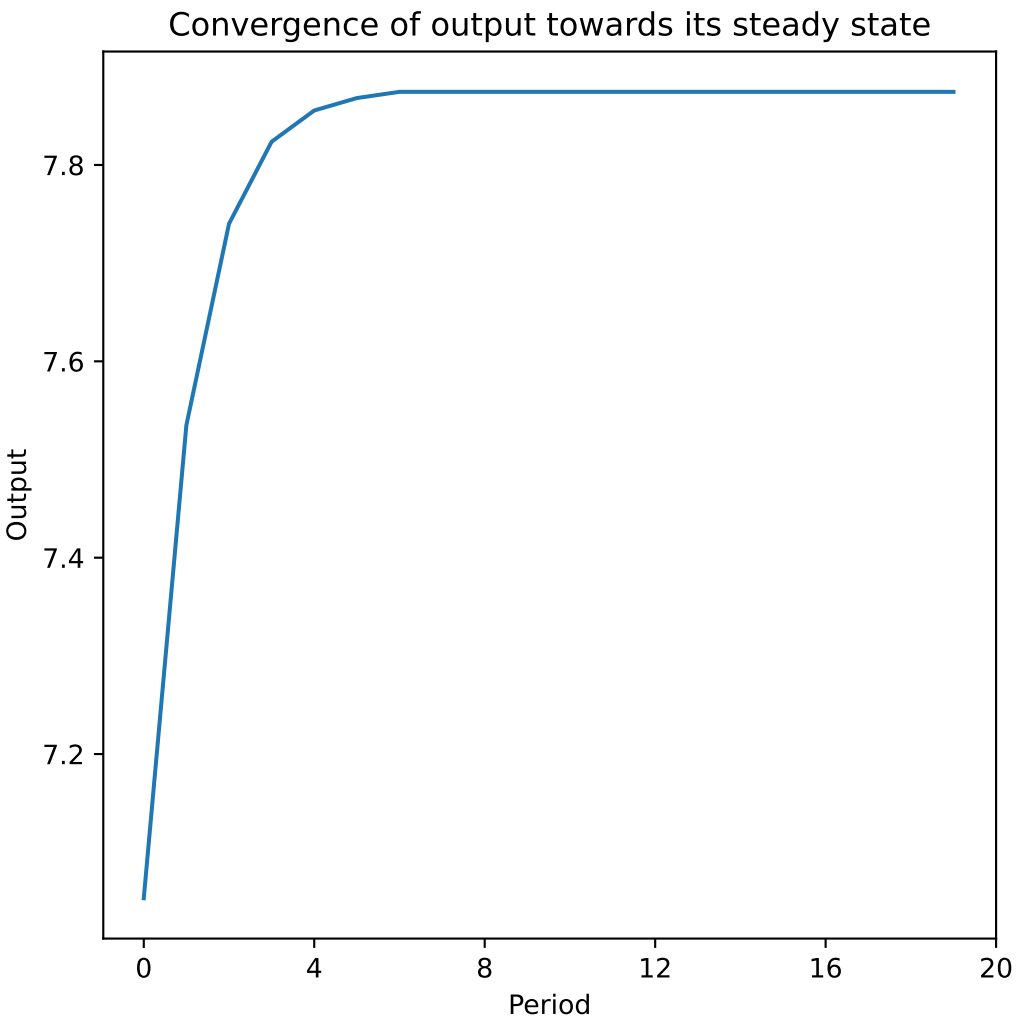
2. We then create a loop that:

    - given the position of k_0, finds the position of the optimal next period capital that corresponds to it.
    - uses the initial capital and the position of the new optimal capital to calculate the values of output, consumption, and the next period capital, using the optimal policy vectors.

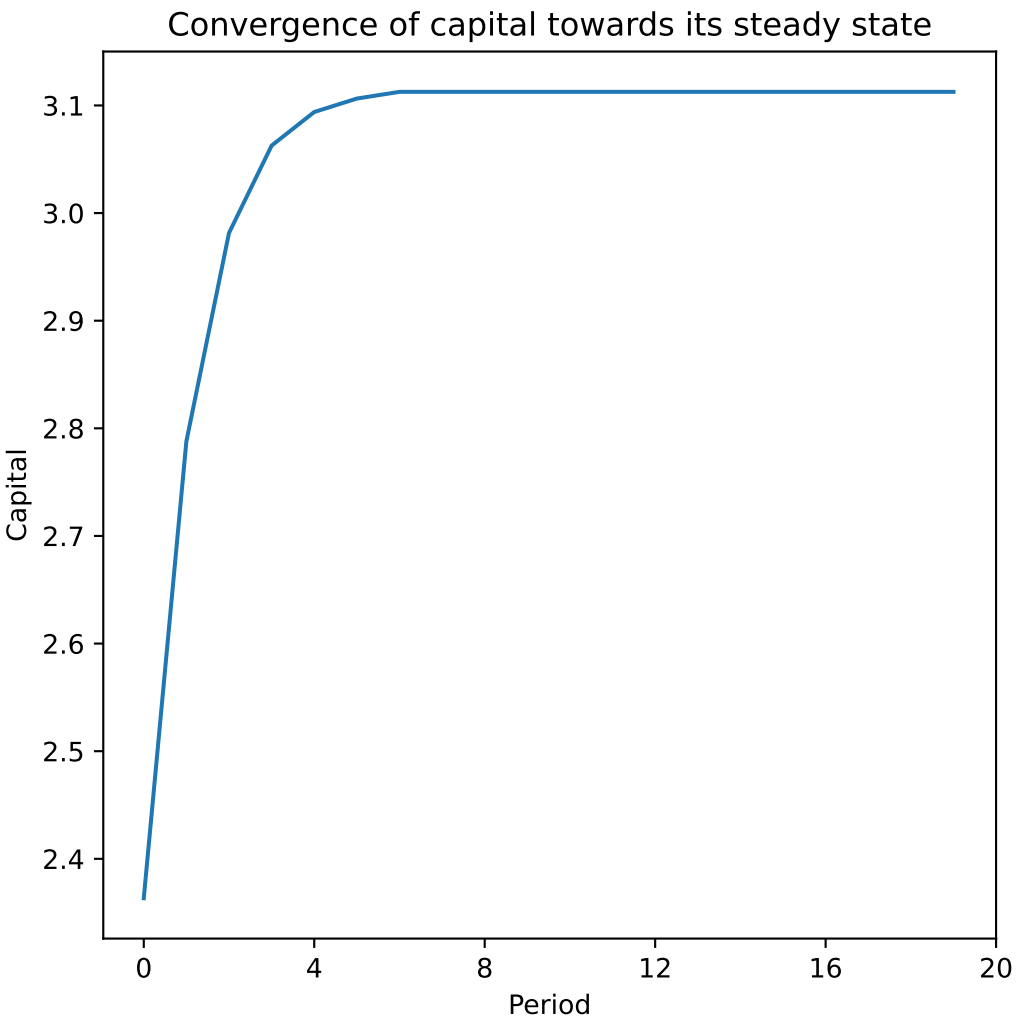
. . .

> **Note**
>
> We need $lk = 500$ ($lk = 2$ is too small) to demonstrate the proper picture. To achieve this: change `lk` in step 2, run all above, then rerun the next cell.

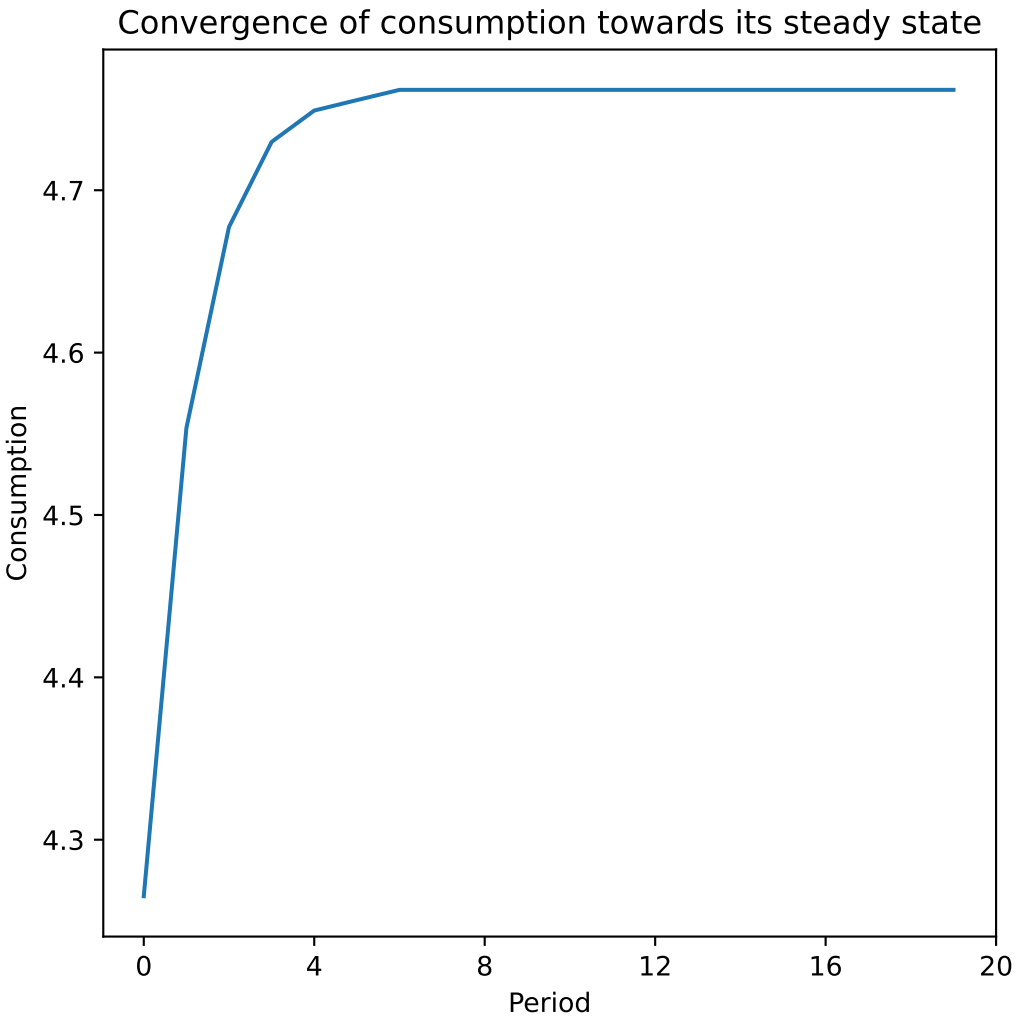
```
indk = 0
kopt = np.zeros(T + 1)
output = np.zeros(T)
cons = np.zeros(T)

kopt[0] = polk[indk]

for i in range(T):
    indk = optim[indk]
    output[i] = A * kopt[i] ** alpha
    kopt[i + 1] = polk[indk]
    cons[i] = output[i] - kopt[i + 1]

plt.figure(figsize=(6, 6))
plt.plot(range(T), output)
plt.title("Convergence of output towards its steady state")
plt.xticks(range(0, T + 1, 4))
plt.xlabel("Period")
plt.ylabel("Output")
plt.show()
```



```
plt.figure(figsize=(6, 6))
plt.plot(range(T), kopt[:T])
plt.title("Convergence of capital towards its steady state")
plt.xticks(range(0, T + 1, 4))
plt.xlabel("Period")
plt.ylabel("Capital")
plt.show()
```



```
plt.figure(figsize=(6, 6))
plt.plot(range(T), cons)
plt.title("Convergence of consumption towards its steady state")
plt.xticks(range(0, T + 1, 4))
plt.xlabel("Period")
plt.ylabel("Consumption")
plt.show()
```



2 Brock and Mirman with iid shocks

2.1 IID Shocks

- The social planner solves the following problem:

$$\max_{\{c_t, k_{t+1}\}} E_0 \sum_{t=0}^{\infty} \beta^t \log(c_t)$$
$$\text{s.t. } c_t + k_{t+1} \leq \theta_t k_t^{\alpha}$$
$$\theta_t \in [\theta_1, \theta_2];\ \Pr(\theta_t = \theta_1) = \Pr(\theta_t = \theta_2) = 0.5$$
$$k_0, \theta_0 \text{ given}; c_t, k_t \geq 0\ \forall t$$

- The controls are c_t, k_{t+1}; the states are k_t and θ_t.[1]
- We can formulate as a dynamic programming problem:

$$V(k,\theta) = \max_{k'} \{\log(\theta_t k^\alpha - k') + \beta \mathbb{E} V(k',\theta')\}$$

- We have to solve for:

$$k' = g_k(k,\theta)$$
$$g_c(k,\theta) = \theta k^\alpha - g_k(k,\theta)$$

2.2 Step 1: Initialization of the parameters and steady state

```
# Model Parameters
alpha = 0.4
beta = 0.9888

# Algorithm Parameters
T = 20  # periods to simulate
tolv = 1e-8  # tolerance
```

2.3 Steady State

- To calculate the steady state, we use:

$$\frac{1}{c} = \beta \mathbb{E} \frac{1}{c'} (\alpha \theta' {k'}^{\alpha-1})$$
$$c + k' = k^\alpha$$

- With $\mathbb{E}(\theta') = \mathbb{E}(\theta) = 1$, and $x = x'$ for all variables, we get:

$$k_{ss} = \left(\frac{1}{\beta\alpha}\right)^{\frac{1}{\alpha-1}}$$
$$c_{ss} = k_{ss}^\alpha - k_{ss}$$
$$y_{ss} = k_{ss}^\alpha$$

. . .

```
# steady state
k_ss = (1 / (beta * alpha)) ** (1 / (alpha - 1))
y_ss = k_ss**alpha
c_ss = y_ss - k_ss

print("k_ss=", k_ss)
```

[1] We assume that the shock θ is iid and the expected value of the shock is 1.

```
k_ss= 0.21311503110303762
```

2.4 Step 2: Creation of a grid for the states

- Create a grid vector for θ and k with l_θ and l_k values:

$$\theta \in [\theta_1, \dots, \theta_{l_\theta}], \quad k \in [k_1, \dots, k_{l_k}]$$

...

```
lt = 2
theta = np.linspace(1 + 1.625 / 100, 1 - 1.625 / 100, num=lt)
print(theta)

lk = 500
k_min = 0.5 * k_ss
k_max = 1.5 * k_ss
k = np.linspace(k_min, k_max, num=lk)
print(k[:5])
```

```
[1.01625 0.98375]
[0.10655752 0.1069846  0.10741168 0.10783877 0.10826585]
```

2.5 Step 3: Computation of the return function

- For each value of θ, k and k', calculate:

$$c = \theta k^\alpha - k'$$
$$U(c) = \log(c)$$

- Create a matrix of dimensions $g_k \times l_\theta$, where $g_k = l_k \times l_k$.

...

```
gk = lk * lk
c = np.zeros((gk, lt))
for t in range(lt):
    for i in range(lk):
        for j in range(lk):
            c[i * lk + j][t] = theta[t] * k[i] ** alpha - k[j]
            if c[i * lk + j][t] < 0:
                c[i * lk + j][t] = tolv
print(c[:5])
```

```
[[0.3084292  0.29515779]
 [0.30800211 0.29473071]
 [0.30757503 0.29430362]
 [0.30714795 0.29387654]
 [0.30672086 0.29344945]]
```

To calculate the return function, simply take the log of the consumption matrix:

```
u = np.log(c)
print(u[:5])
```

```
[[-1.17626296 -1.22024518]
 [-1.17764863 -1.2216932 ]
 [-1.17903622 -1.22314332]
 [-1.18042574 -1.22459554]
 [-1.18181719 -1.22604987]]
```

2.6 Step 4: Initialization and computation of the optimal value function

- Our candidate solution for the value function is an $l_k \times l_\theta$ matrix:

$$V(k,\theta) = \begin{bmatrix} V(k_1,\theta_1) & V(k_1,\theta_2) \\ V(k_2,\theta_1) & V(k_2,\theta_2) \\ \vdots & \vdots \\ V(k_{l_k},\theta_1) & V(k_{l_k},\theta_2) \end{bmatrix}$$

- Initialize V_0 as zeros of dimension $l_k \times l_\theta$ and $V_1 = T(V_0)$. At iteration n, calculate:

$$V_n(k_i,\theta_t) = \max_{k'} \left\{ \begin{bmatrix} U(\theta_t k_i^\alpha - k'_1) \\ \vdots \\ U(\theta_t k_i^\alpha - k'_{l_k}) \end{bmatrix} + \beta \begin{bmatrix} \sum_{m=1}^{2} \frac{1}{2} V_{n-1}(k'_1,\theta'_m) \\ \vdots \\ \sum_{m=1}^{2} \frac{1}{2} V_{n-1}(k'_{l_k},\theta'_m) \end{bmatrix} \right\}$$

2.7 Value Function Iteration

```
# Initialize value function and operator and iterate
V0 = np.ones((lk, lt))
V1 = np.zeros((lk, lt))
p = np.array([0.5, 0.5])
start_time = time.time()
while abs(np.linalg.norm(V1 - V0)) > tolv:
    V0 = np.copy(V1)
    for j in range(lt):
        for i in range(lk):
            vtemp = u[i * lk : (i + 1) * lk, j] + beta * V0 @ p
            V1[i, j] = np.max(vtemp)
V0 = np.copy(V1)
print(V0[:5])
print(f"Time taken: {time.time() - start_time:.2f} seconds")
```

```
[[-100.6080923  -100.66186288]
 [-100.60544596 -100.65921516]
 [-100.60280894 -100.65657915]
 [-100.60018353 -100.65395331]
 [-100.59756785 -100.65133759]]
Time taken: 4.96 seconds
```

2.8 Step 5: Computation of the optimal policy functions

For each point in the grid, we find the index of the capital maximizer using the max operator.

```
optim = np.zeros((lk, lt), dtype="int")  # "int" type for index.
for j in range(lt):
    for i in range(lk):
        vtemp = u[i * lk : (i + 1) * lk, j] + beta * V0 @ p
        optim[i, j] = np.argmax(vtemp)
print(optim[:5])
```

```
[[135 123]
 [135 123]
 [136 124]
 [137 124]
 [137 125]]
```

Using this matrix, the policy functions can be calculated as follows:

$$\text{polk} = \begin{bmatrix} g_k(k_1, \theta_1) & g_k(k_1, \theta_2) \\ \vdots & \vdots \\ g_k(k_{l_k}, \theta_1) & g_k(k_{l_k}, \theta_2) \end{bmatrix}, \text{polc} = \begin{bmatrix} g_c(k_1, \theta_1) & g_c(k_1, \theta_2) \\ \vdots & \vdots \\ g_c(k_{l_k}, \theta_1) & g_c(k_{l_k}, \theta_2) \end{bmatrix}$$

$$\text{and} \quad \text{polc} = \begin{bmatrix} \theta_1 k_1^{\alpha} & \theta_2 k_1^{\alpha} \\ \vdots & \vdots \\ \theta_1 k_{l_k}^{\alpha} & \theta_2 k_{l_k}^{\alpha} \end{bmatrix} - \text{polk}$$

```
polk = k[optim]
print(polk[:5])
```

```
[[0.16421389 0.15908888]
 [0.16421389 0.15908888]
 [0.16464097 0.15951596]
 [0.16506806 0.15951596]
 [0.16506806 0.15994304]]
```

2.9 Simulating the model

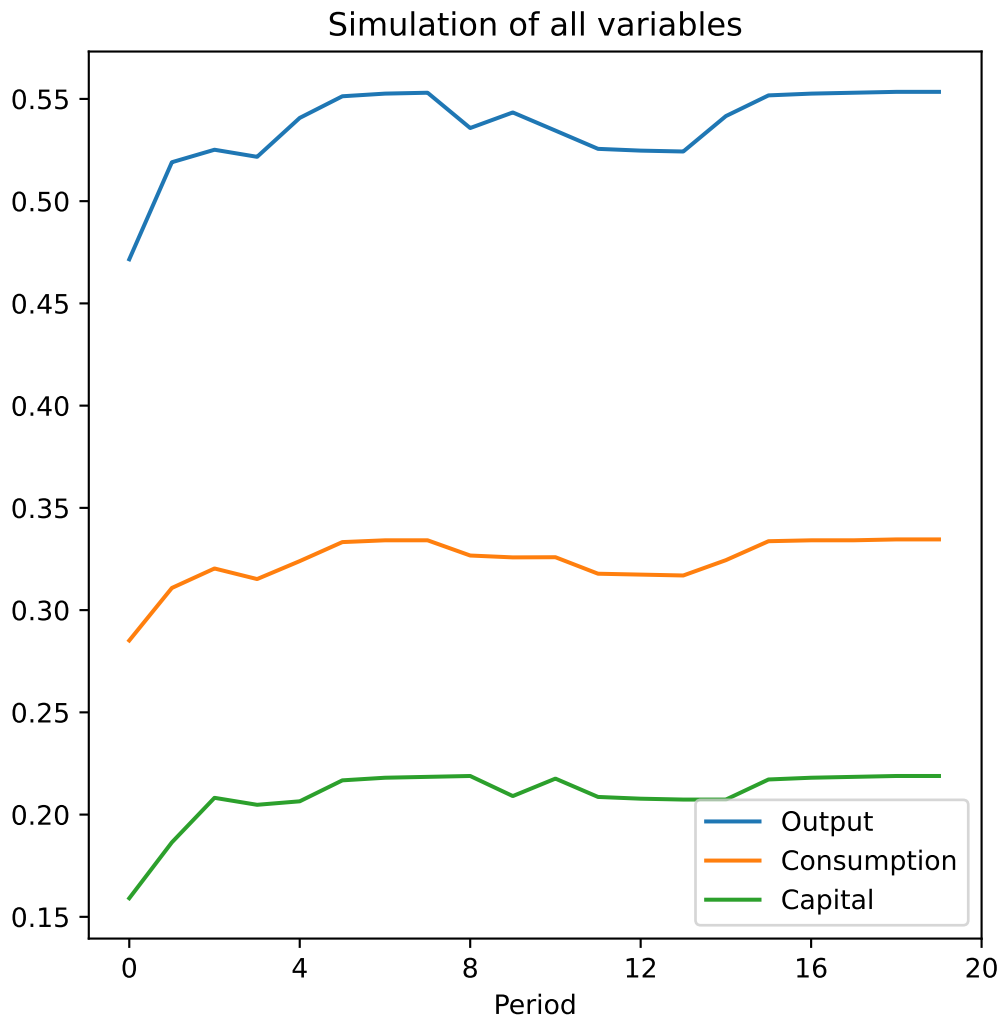
1. Choose an initial value for k_0 and θ_0 and use the policy matrix to extract the optimal first-period capital.
2. We then create a loop that:
    - given the initial capital and shock, finds the position of the optimal next period capital.
    - uses the initial capital, the initial shock index, and the next capital to calculate output, consumption, and stores variables for each period. It also generates a new shock index for next period.

```
indk = 1
shock = 1
kopt = np.zeros(T + 1)
sho = np.zeros(T)
output = np.zeros(T)
cons = np.zeros(T)

kopt[0] = polk[indk, shock]
```

```
for i in range(T):
    indk = optim[indk, shock]
    sho[i] = theta[shock]
    kopt[i + 1] = polk[indk, shock]
    output[i] = theta[shock] * kopt[i] ** alpha
    cons[i] = output[i] - kopt[i + 1]
    shock = np.random.randint(0, 2)
```

2.10 Plotting the results

```
plt.figure(figsize=(6, 6))
plt.plot(range(T), output, label="Output")
plt.plot(range(T), cons, label="Consumption")
plt.plot(range(T), kopt[:T], label="Capital")
plt.title("Simulation of all variables")
plt.xticks(range(0, T + 1, 4))
plt.xlabel("Period")
plt.legend()
plt.show()
```



3 Stochastic Growth Model with Markov Shocks

3.1 Markov Shocks

- The social planner solves the following problem:

$$\max_{\{c_t, k_{t+1}\}} E_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma}}{1-\gamma}$$
$$\text{s.t. } c_t + i_t \leq \theta_t A k_t^{\alpha}$$
$$k_{t+1} = i_t + (1-\delta)k_t$$
$$k_0, \theta_0 \text{ given; } c_t, k_t \geq 0, \ \forall t$$

- The controls are c_t, i_t, k_{t+1} and the states are k_t and θ_t.
- The dynamic programming formulation is:

$$V(k,\theta) = \max_{k'}\{\log(\theta Ak^{\alpha} - k' + (1-\delta)k) + \beta EV(k',\theta')\}$$

- We have to solve for:

$$k' = g_k(k,\theta)$$
$$g_c(k,\theta) = \theta Ak^{\alpha} + (1-\delta)k - g_k(k,\theta)$$

- The log of the productivity shock follows an AR(1) process:

$$x_t = \rho x_{t-1} + \epsilon_t \text{ where } \epsilon_t \sim N(0,\sigma^2)\theta_t = \exp(x_t)$$

- We can use a Markov Chain to approximate the AR(1) process.

3.2 Markov-Chain Approximation of AR(1) Productivity Shocks

Let the (finite) state space be $S = \{\theta_1, \dots, \theta_N\}$.

- **Markov property (memorylessness).** For any i, j,

$$\Pr(\theta_{t+1} = \theta_j \mid \theta_t = \theta_i, \theta_{t-1}, \dots) = \Pr(\theta_{t+1} = \theta_j \mid \theta_t = \theta_i) = P_{ij}.$$

- **Transition matrix** $P \in \mathbb{R}^{N\times N}$:

$$P_{ij} \geq 0, \qquad \sum_{j=1}^{N} P_{ij} = 1 \;\; \forall i = 1, 2, ..., N.$$

- **Stationary (invariant) distribution.** A vector $\pi \in \Delta^{N-1}$ is stationary if

$$\pi^{\top} = \pi^{\top} P, \qquad \pi_i \geq 0, \; \sum_i \pi_i = 1.$$

  If the chain is **irreducible** and **aperiodic** (ergodic), then π is unique and for any initial distribution μ_0,

$$\mu_t^{\top} = \mu_0^{\top} P^t \to \pi^{\top}.$$

- **How expectations enter DP.** If current shock is $\theta = \theta_i$,

$$\mathbb{E}[V(k',\theta') \mid \theta = \theta_i] \;\approx\; \sum_{j=1}^{N} P_{ij} V(k',\theta_j).$$

3.3 Approximating AR(1) shocks: Tauchen (1986) discretization

- It is standard to discretize **log productivity** to ensure positivity.
- Assume

$$x_t = \rho x_{t-1} + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma^2), \qquad \theta_t = \exp(x_t).$$

- **(i) Choose a grid** of N states for x
  - Unconditional standard deviation:

$$\sigma_x = \frac{\sigma}{\sqrt{1-\rho^2}}.$$

  - Pick a truncation width $m > 0$ (often $m = 3$) and set

$$x_1 = -m\sigma_x, \quad x_N = m\sigma_x, \quad x_i = x_1 + (i-1)\Delta, \quad \Delta = \frac{x_N - x_1}{N-1}.$$

- **(ii) Build the transition matrix**
  - Define interval cutoffs

$$b_{j-} = x_j - \frac{\Delta}{2}, \qquad b_{j+} = x_j + \frac{\Delta}{2},$$

    with $b_{1-} = -\infty$ and $b_{N+} = +\infty$. Then

$$P_{ij} = \Pr(x_{t+1} \in [b_{j-}, b_{j+}] \mid x_t = x_i) = \Phi\left(\frac{b_{j+} - \rho x_i}{\sigma}\right) - \Phi\left(\frac{b_{j-} - \rho x_i}{\sigma}\right),$$

    where $\Phi(\cdot)$ is the standard normal CDF.

. . .

> **Note**
>
> When ρ is close to 1 and N is small, Rouwenhorst (1995) often matches persistence better than Tauchen (1986).

3.4 Step 1: Initialization of the parameters and steady state

```
# Model and Algorithm Parameters
delta = 0.1
alf = 0.33
A = 1
beta = 0.9
gamma = 2
tolv = 1e-7
```

3.5 Construction of a Markov Chain

We use the `quantecon` library to construct a Markov Chain.

- First, we set up the parameter values for the AR(1) process that drives the aggregate productivity shock.
- We then use Tauchen's method to discretize this process into a finite Markov chain.

. . .

```
# Step 1: Specify the AR(1) process parameters and discretize with Tauchen's method
import quantecon as qe

rho = 0.95  # persistence of the AR(1) process
sigmae = 0.00712  # standard deviation of the error term
lt = 4  # number of grid states for the Markov chain
n_std = 3  # number of standard deviations to cover in the grid

# Use Tauchen's method to approximate the AR(1) with a Markov chain
x_mc = qe.markov.approximation.tauchen(rho=rho, sigma=sigmae, n_std=n_std, n=lt, mu=0)
```

- Next, we extract the transition matrix, the grid for the state variable representing productivity shocks, compute the values of productivity in levels (by exponentiating), and obtain the stationary distribution over states.

. . .

```
# Step 2: Extract transition matrix and productivity states, and compute stationary distribution
P = x_mc.P  # transition matrix (to match standard notation)
x_grid = x_mc.state_values  # discretized grid for the state variable
teta = np.exp(x_grid)  # productivity in levels, since shock is in logs
invdist = np.array(x_mc.stationary_distributions)  # stationary distribution
```

```
print(f"Stationary distribution: {invdist}")
print(f"Productivity states: {teta}")
print(f"Transition matrix: {P[0,:]}")
print(f"Check (sum of first row): {np.sum(P[0,:])}")
```

```
Stationary distribution: [[0.05317954 0.44682046 0.44682046 0.05317954]]
Productivity states: [0.93388054 0.97745576 1.02306421 1.07080077]
Transition matrix: [0.99675735 0.00324265 0.         0.        ]
Check (sum of first row): 1.0
```

> **Note**
>
> The transition matrix `P[i,j]` denotes the probability of transitioning from state `i` to state `j`. That is:
>
> $$P_{ij} = \Pr(x_{t+1} \in [b_{j-}, b_{j+}] \mid x_t = x_i) = \Pr(x_{t+1} = x_j \mid x_t = x_i)$$

3.6 Steady State

The equilibrium is given by:

$$c^{-\gamma} = \beta E[(c')^{-\gamma}(\alpha\theta' A k'^{\alpha-1} + 1 - \delta)]$$
$$c + i = \theta A k^{\alpha}$$
$$i = k' - (1-\delta)k$$

Imposing $\theta_{ss} = E\theta' = E\theta$ and $x = x'$ gives:

$$k_{ss} = \left(\frac{1-\beta(1-\delta)}{\beta\alpha A\theta_{ss}}\right)^{\frac{1}{\alpha-1}}$$
$$i_{ss} = \delta k_{ss}$$
$$c_{ss} = \theta_{ss} A k_{ss}^{\alpha} - i_{ss}$$
$$y_{ss} = A\theta_{ss} k_{ss}^{\alpha}$$

```
Eteta = invdist @ teta
k_ss = (A * Eteta * alf * beta / (1 - beta * (1 - delta))) ** (1 / (1 - alf))
y_ss = Eteta * A * (k_ss**alf)
i_ss = delta * k_ss
c_ss = y_ss - i_ss
print(k_ss)
```

```
[1.94925376]
```

> **Caution**
>
> Note that `k_ss` is actually a `nd.array` of shape `(1,)`. This is because `Eteta` is a `nd.array` of shape `(1,)` not a scalar (`float`).

3.7 Step 2: Creation of a grid for the states

Create a grid for the shock and for k with l_k values, i.e. $k \in [k_1 < ... < k_{l_k}]$.

```
# Grid for capital
lk = 500
k_min = 0.9 * k_ss
k_max = 1.1 * k_ss
k = np.linspace(k_min, k_max, num=lk)
print(k[:5])
```

```
[[1.75432839]
 [1.75510965]
 [1.75589092]
 [1.75667218]
 [1.75745344]]
```

3.8 Step 3: Computation of the return function

For each θ, k and k', calculate:

$$c = \theta A k^{\alpha} + (1 - \delta)k - k'$$

$$U(c) = \frac{c^{1-\gamma}}{1-\gamma}$$

```
gk = lk * lk
c = np.zeros((gk, lt))
for t in range(lt):
    for i in range(lk):
        for j in range(lk):
            c[i * lk + j][t] = A * teta[t] * k[i] ** alf + (1 - delta) * k[i] - k[j]
            if c[i * lk + j][t] < 0:
                c[i * lk + j][t] = tolv
print(c[0, :])
```

```
/var/folders/s9/94vsjp21473gbgm66xdz80k00000gn/T/ipykernel_27303/1062897660.py:6: DeprecationWarnin
  c[i * lk + j][t] = A * teta[t] * k[i] ** alf + (1 - delta) * k[i] - k[j]
```

```
[0.9487784  1.00123451 1.05613825 1.11360382]
```

To calculate the return function, take logs if $\gamma = 1$, or compute $\frac{c^{1-\gamma}}{1-\gamma}$ otherwise.

```
if gamma == 1:
    U = np.log(c)
else:
    U = c ** (1 - gamma) / (1 - gamma)
print(U[0, :])
```

```
[-1.0539869  -0.99876701 -0.94684574 -0.89798543]
```

3.9 Step 4: Initialization and computation of the optimal value function

Our candidate solution for the value function is an $l_k \times l_\theta$ matrix.

```
# Initialize of the value function
V0 = np.ones((lk, lt))
V1 = np.zeros((lk, lt))
start_time = time.time()
while abs(np.linalg.norm(V1 - V0)) > tolv:
    V0 = np.copy(V1)
    for j in range(lt):
        for i in range(lk):
            vtemp = U[i * lk : (i + 1) * lk, j] + beta * V0 @ P[j, :]
            V1[i, j] = np.max(vtemp)

V0 = np.copy(V1)
print(V0[0, :])
end_time = time.time()
print(f"Time taken: {end_time - start_time} seconds")
```

```
[-10.5240125   -9.98068286  -9.45989273  -8.96829951]
Time taken: 1.9204938411712646 seconds
```

3.10 Step 5: Computation of the optimal policy functions

```
optim = np.zeros((lk, lt), dtype="int")
for j in range(lt):
    for i in range(lk):
        vtemp = U[i * lk : (i + 1) * lk, j] + beta * V0 @ P[j, :]
        optim[i, j] = np.argmax(vtemp)
print(optim[0, :])
```

```
[ 0 20 42 64]
```

- For each pair (θ_j, k_i), `optim[i, j]` is the index of the next period capital maximizer.
- If $l_k = 2$ and $l_\theta = 4$, the policy functions are:

$$\text{polk} = \begin{bmatrix} g_k(k_1,\theta_1) & g_k(k_1,\theta_2) & g_k(k_1,\theta_3) & g_k(k_1,\theta_4) \\ g_k(k_2,\theta_1) & g_k(k_2,\theta_2) & g_k(k_2,\theta_3) & g_k(k_2,\theta_4) \end{bmatrix}$$

$$\text{polc} = \begin{bmatrix} \theta_1 k_1^\alpha & \theta_2 k_1^\alpha & \theta_3 k_1^\alpha & \theta_4 k_1^\alpha \\ \theta_1 k_2^\alpha & \theta_2 k_2^\alpha & \theta_3 k_2^\alpha & \theta_4 k_2^\alpha \end{bmatrix} + (1-\delta) \begin{bmatrix} k_1 & k_1 & k_1 & k_1 \\ k_2 & k_2 & k_2 & k_2 \end{bmatrix} - \text{polk}$$

```
polk = k[optim]
print(polk[0, :])
```

```
[[1.75432839]
 [1.76995367]
 [1.78714148]
 [1.80432929]]
```

3.11 Simulating the model

1. Choose an initial value for k_0 and θ_0 and use the policy matrix to extract the optimal first period capital.
2. In a loop:
    - given the initial capital and shock, find the optimal next period capital.
    - use the current values to compute and store output, consumption, and capital, generating a new shock index for next period.

```
T = 2000  # simulate longer if you want moments
burn = 200  # burn-in to remove dependence on initial state

# initial indices
ik = lk // 2  # start near middle of k grid
s = np.argmax(invdist)  # start at most likely shock state (or pick 0)

k_path = np.empty(T + 1)
c_path = np.empty(T)
y_path = np.empty(T)
i_path = np.empty(T)
theta_path = np.empty(T, dtype=float)
s_path = np.empty(T, dtype=int)

k_path[0] = k[ik]

rng = np.random.default_rng(123)  # reproducible
```

```
/var/folders/s9/94vsjp21473gbgm66xdz80k00000gn/T/ipykernel_27303/2521805887.py:15: DeprecationWarni
  k_path[0] = k[ik]
```

```
for t in range(T):
    theta = teta[s]
    theta_path[t] = theta
    s_path[t] = s

    # policy: choose next capital index and level
    ik_next = optim[ik, s]
    k_next = k[ik_next]

    # compute flows from resource constraint
    y = theta * A * (k_path[t] ** alf)
    i = k_next - (1 - delta) * k_path[t]
    c = y - i

    # store
    y_path[t] = y
    i_path[t] = i
    c_path[t] = c
```

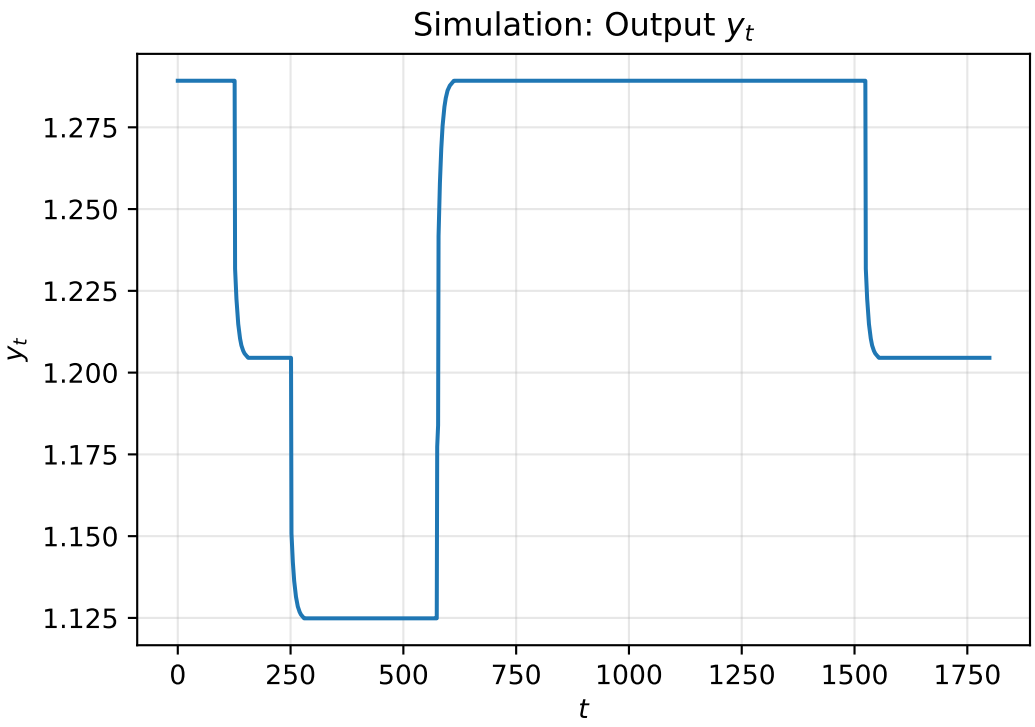
```
    k_path[t + 1] = k_next

    # update indices
    ik = ik_next
    s = rng.choice(lt, p=P[s, :])  # Markov transition
```

```
/var/folders/s9/94vsjp21473gbgm66xdz80k00000gn/T/ipykernel_27303/1559416502.py:17: DeprecationWarni
  i_path[t] = i
/var/folders/s9/94vsjp21473gbgm66xdz80k00000gn/T/ipykernel_27303/1559416502.py:18: DeprecationWarni
  c_path[t] = c
/var/folders/s9/94vsjp21473gbgm66xdz80k00000gn/T/ipykernel_27303/1559416502.py:19: DeprecationWarni
  k_path[t+1] = k_next
```

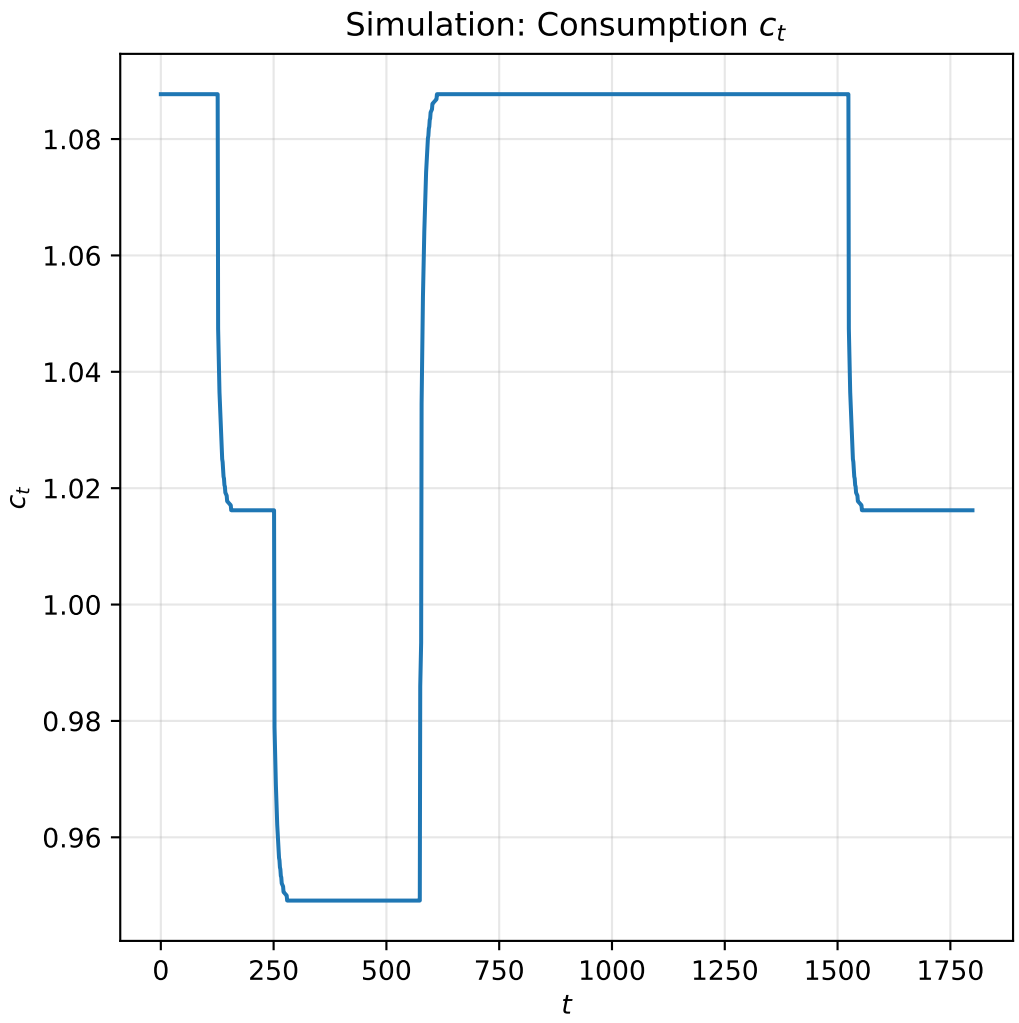
```
# drop burn-in
k_sim = k_path[burn:]
c_sim = c_path[burn:]
y_sim = y_path[burn:]
i_sim = i_path[burn:]
theta_sim = theta_path[burn:]

print("Simulation moments (post burn-in):")
print(f"E[k]={k_sim.mean():.4f} E[c]={c_sim.mean():.4f} E[y]={y_sim.mean():.4f}")
print(f"min(c)={c_sim.min():.4f} min(i)={i_sim.min():.4f}")
```

```
Simulation moments (post burn-in):
E[k]=1.9399 E[c]=1.0469 E[y]=1.2409
min(c)=0.9491 min(i)=0.1719
```

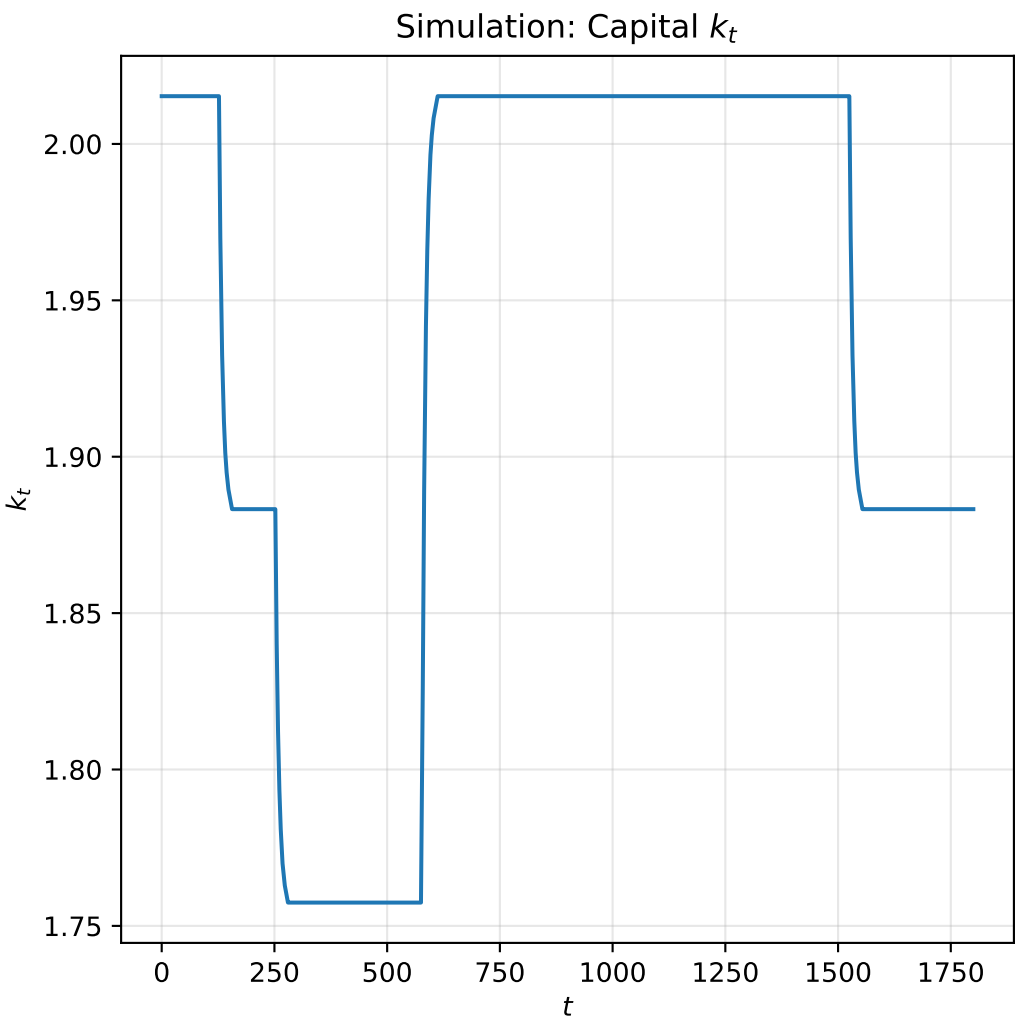
```
plt.figure(figsize=(6, 4))
plt.plot(range(len(y_sim)), y_sim)
plt.title("Simulation: Output $y_t$")
plt.xlabel("$t$")
plt.ylabel("$y_t$")
plt.grid(True, alpha=0.3)
plt.show()
```



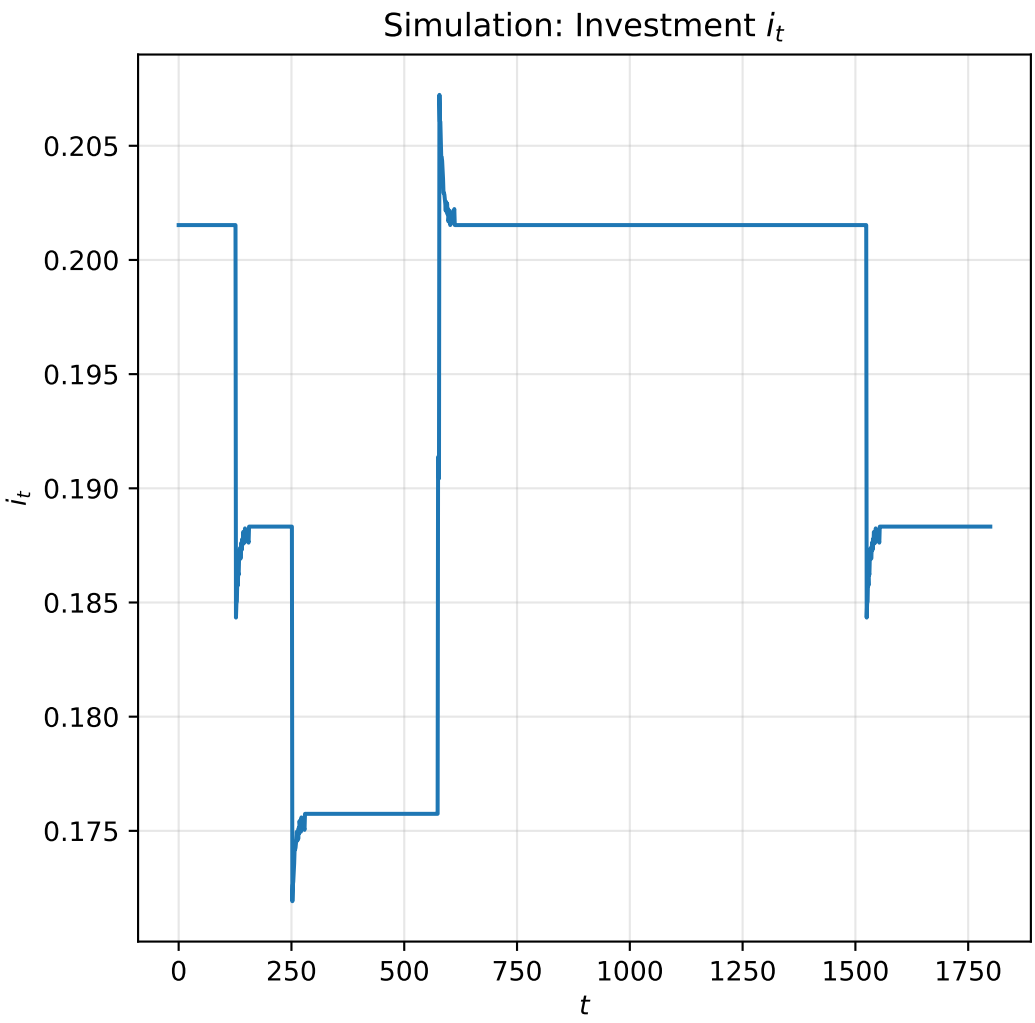
```
plt.figure(figsize=(6, 6))
plt.plot(range(len(c_sim)), c_sim)
plt.title("Simulation: Consumption $c_t$")
plt.xlabel("$t$")
plt.ylabel("$c_t$")
plt.grid(True, alpha=0.3)
plt.show()
```



```
plt.figure(figsize=(6, 6))
plt.plot(range(len(k_sim)), k_sim)
plt.title("Simulation: Capital $k_t$")
plt.xlabel("$t$")
plt.ylabel("$k_t$")
plt.grid(True, alpha=0.3)
plt.show()
```



```
plt.figure(figsize=(6, 6))
plt.plot(range(len(i_sim)), i_sim)
plt.title("Simulation: Investment $i_t$")
plt.xlabel("$t$")
plt.ylabel("$i_t$")
plt.grid(True, alpha=0.3)
plt.show()
```



```
plt.figure(figsize=(6, 6))
plt.plot(range(len(theta_sim)), theta_sim)
plt.title(r"Simulation: Productivity Shock $\theta_t$")
plt.xlabel("$t$")
plt.ylabel(r"$\theta_t$")
plt.grid(True, alpha=0.3)
plt.show()
```

---

4 Exercise: Stochastic Growth Model with Irreversible Investment

4.1 Irreversible Investment

The social planner solves the following problem:

$$\max_{\{c_t,k_{t+1}\}} E_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma}}{1-\gamma}$$
$$c_t + i_t \leq \theta_t A k_t^{\alpha}$$
$$k_{t+1} = i_t + (1-\delta)k_t$$
$$i_t \geq 0$$

k_0, θ_0 given, $c_t, k_t \geq 0$ for all t.
The code is basically the same as above, except for a small modification in step 2.

References

Brock, William A., and Leonard J. Mirman. 1972. "Optimal Economic Growth and Uncertainty: The Discounted Case." *Journal of Economic Theory* 4 (3): 479–513.

Rouwenhorst, K. Geert. 1995. "Asset Pricing Implications of Equilibrium Business Cycle Models." *Handbooks in Operations Research and Management Science* 9: 945–74.

Tauchen, George. 1986. "Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions." *Economics Letters* 20 (2): 177–81.