Aiyagari Model

Lecturer: Bo Li TA: Chen Gao

2026-03-27

Table of contents

1 Model Description

1.1 A Standard Savings Problem

The economy is populated by a large number of ex-ante identical but ex-post heterogeneous households.

Households trade in a single security to insure against risks.

There is no aggregate uncertainty, implying that the aggregate variables will be constant in the stationary equilibrium.

Individual income at period t, denoted by s_t, follows an m-state Markov chain with transition matrix π.

We have that:

- $s_t \in \{s_1, s_2, \dots, s_m\}$
- $\pi_{ij} = \Pr(s_{t+1} = s_j \mid s_t = s_i)$ for $i = 1, \dots, m$ and $j = 1, \dots, m$

1.2 A Standard Savings Problem

If the realization of the process at period t is s_t, a household receives labor income equal to ws_t, where w is the wage rate.

If the asset holdings of the household at the beginning of the period are equal to a_t, she receives a gross return of $(1+r)a_t$.

For given (w, r) and (a_0, s_0), the household solves:

$$\begin{aligned} \max_{\{c_t, a_{t+1}\}_{t=0}^{\infty}} \quad & \mathbb{E}_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right] \\ \text{s.t.} \quad & c_t + a_{t+1} = (1+r)a_t + ws_t \\ & a_{t+1} \in A \end{aligned}$$

1.3 Assumptions of the Standard Savings Problem

We assume that $\beta \in (0,1)$ and $u(c)$ satisfies $\lim_{c \to 0} u'(c) = \infty$. This last assumption ensures that $c > 0$. In addition, $u(c)$ is strictly increasing, strictly concave, and twice continuously differentiable.

An important assumption is $\beta(1+r) < 1$. This prevents assets from growing to infinity.

Regarding the factor prices (w, r), we distinguish between two cases:

- **Pure exchange economy:** Each household can borrow or lend in a zero net supply asset at a constant r, while the wage w is also constant. This is the set-up studied by Huggett (1993).
- **Production economy:** Both factor prices are determined by the firm in equilibrium. This is the set-up studied by Aiyagari (1994).

1.4 Dynamic Programming Problem

If we discretize the state space, the problem of the household can be solved using standard numerical dynamic programming techniques.

- The individual state is given by the pair (a_t, s_t), where s_t is already discrete.
- We assume that the asset holdings can only take a finite number of values, i.e., $a_t \in \{a_1, a_2, \dots, a_n\}$, incorporating upper and lower limits on how much can be borrowed.

The dynamic programming problem of the household is then given by:

$$v(a,s) = \max_{a' \in A} \left\{ u\big[(1+r)a + ws - a'\big] + \beta \sum_j \pi_{ij} v(a', s_j) \right\} \text{ subject to } \quad c + a' = (1+r)a + ws$$

1.5 Solution and Stationary Distribution

Given (w, r), the solution consists of the optimal value and policy functions $v(a, s)$, $a' = g_a(a, s)$ and $c = g_c(a, s)$.
The policy functions have to satisfy market clearing when aggregated over households.

To aggregate, however, we need the asset-employment distribution, making this type of model more complicated than the standard representative household environment.

We use the fact that a **stationary employment-wealth distribution** can be obtained by defining a Markov process for the distribution and by calculating the stationary distribution of this Markov chain.

1.6 Solution and Stationary Distribution

We approximate the process for (a, s) with a Markov chain, and define a stationary probability vector $\lambda(a_n, s_m)$, where

$$\lambda(a_n, s_m) = \Pr(a_t = a_n, s_t = s_m)$$

$\lambda(a, s)$ represents the fraction of time that a household with state (a, s) spends in that state.

Alternatively, $\lambda(a, s)$ can be interpreted as the fraction of households with asset holdings a and labor endowment s. There is an equivalence between the cross-sectional and time-series properties of this distribution.

Let the individual state of the economy be represented by the vector

$$x = \begin{pmatrix} (s_1, a_1) & \cdots & (s_1, a_n) & (s_2, a_1) & \cdots & (s_2, a_n) & \cdots & (s_m, a_1) & \cdots & (s_m, a_n) \end{pmatrix}^\top$$

where each (s_i, a_j) is a possible state.

1.7 Solution and Stationary Distribution

The policy function $a' = g_a(a, s)$ and the shock transition matrix Π induce a Markov chain on x via the following formula:

$$\begin{aligned} \Pi &= \Pr[(a_{t+1} = a',\ s_{t+1} = s') \mid (a_t = a,\ s_t = s)] \\ &= \Pr[a_{t+1} = a' \mid s_t = s,\ a_t = a] \cdot \Pr[s_{t+1} = s' \mid s_t = s] \\ &= Y(a', a, s)\, \pi \end{aligned}$$

where $Y(a', a, s)$ is an indicator function equal to one if $a' = g(a, s)$ and zero otherwise.

The stationary probability vector of this Markov chain is given by:

$$p_\infty = \begin{pmatrix} \lambda(s_1, a_1) & \cdots & \lambda(s_1, a_n) & \lambda(s_2, a_1) & \cdots & \lambda(s_2, a_n) & \cdots & \lambda(s_m, a_1) & \cdots & \lambda(s_m, a_n) \end{pmatrix}^\top$$

The aggregate asset holdings are equal to $p_\infty^\top x$.

1.8 The Model of Aiyagari (1994)

In the model of Aiyagari (1994), the asset holdings a_{t+1} represent holdings of capital K_{t+1}, which evolves according to

$$K_{t+1} = (1 - \delta)K_t + x_t,$$

where x_t is gross investment and δ is the depreciation rate.

The consumption of a household is therefore equal to

$$c_t + a_{t+1} = (1 + r)a_t + ws_t,$$

where $a_{t+1} = K_{t+1}$.

The factor prices $r = \tilde{r} - \delta$ and w are determined from the marginal conditions of the firm.

1.9 The Model of Aiyagari (1994)

The author assumed that there is an aggregate production function whose arguments are the average levels of capital and employment:

$$F(K, N) = AK^\alpha N^{1-\alpha}.$$

Therefore,

$$w = \frac{\partial F(K, N)}{\partial N}$$

and

$$\tilde{r} = \frac{\partial F(K, N)}{\partial K}.$$

1.10 Stationary Equilibrium

A **stationary equilibrium** consists of a value function $v(a, s)$, policy functions $g_k(k, s)$ and $g_c(k, s)$, a probability distribution $\lambda(k, s)$, factor prices $(w, \tilde{r})$, and aggregate capital and labor (K, N), such that:

1. **Factor prices** satisfy the conditions for profit maximization.
2. **Value and policy functions** solve the household problem given factor prices.
3. **Probability distribution** $\lambda(k, s)$ is the stationary distribution associated with $g_k(k, s)$ and π.
4. **Markets clear**:

$$K = \sum_{k,s} \lambda(k, s)\, g_k(k, s)$$

$$N = p_\infty^\top s$$

2 Code

2.1 HH problem

We start with the imports.

```
import jax
import jax.numpy as jnp
import time
import quantecon as qe
from collections import namedtuple
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings("ignore")
```

let's use 64 bit floats here.

```
jax.config.update("jax_enable_x64", True)
```

for a helper function, we need to compute stationary distributions of stochastic matrices.

```python
@jax.jit
def compute_stationary(P):
    n = P.shape[0]
    I = jnp.eye(n)
    O = jnp.ones((n, n))
    A = I - P.T + O
    return jnp.linalg.solve(A, jnp.ones(n))


print(compute_stationary(jnp.array([[0.9, 0.1], [0.1, 0.9]])))
```

```
[0.5 0.5]
```

2.2 Firms

Consider

$$Y = AK^{\alpha}N^{1-\alpha}$$

the firm solves:

$$\max_{K,N} \left\{ AK^{\alpha}N^{1-\alpha} - (r+\delta)K - wN \right\}$$

we use a namedtuple to store the parameters of the firm.

```python
Firm = namedtuple("Firm", ("A", "N", "alpha", "delta"))


def create_firm(
    A=1.0,
    N=1.0,
    alpha=0.33,
    delta=0.05,
):
    return Firm(A=A, N=N, alpha=alpha, delta=delta)
```

---

it's easy to see that

$$r = A\alpha \left(\frac{N}{K}\right)^{1-\alpha} - \delta$$

```
@jax.jit
def r_given_k(K, firm: Firm):
    A, N, alpha, delta = firm
    return A * alpha * (N / K) ** (1 - alpha) - delta
```

then for labor:

$$w(r) = A(1-\alpha)\left(\frac{A\alpha}{r+\delta}\right)^{\alpha/(1-\alpha)}$$

```
@jax.jit
def w_given_r(r, firm: Firm):
    A, N, alpha, delta = firm
    return A * (1 - alpha) * (A * alpha / (r + delta)) ** (alpha / (1 - alpha))
```

2.3 Households

Infinitely lived households / consumers face idiosyncratic income shocks. The savings problem faced by a typical household is:

$$\max \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

subject to:

$$a_{t+1} + c_t \leq wz_t + (1+r)a_t, c_t \geq 0, a_t \geq -B$$

we setup the household using namedtuple, too.

```
HH = namedtuple("HH", ("beta", "a_grid", "z_grid", "Pi"))


def create_HH(
    beta=0.96,
    a_min=1e-10,
    a_max=20,
    a_size=400,
):

    z_grid, Pi = jnp.array([0.1, 1.0]), jnp.array([[0.9, 0.1], [0.1, 0.9]])
    a_grid = jnp.linspace(a_min, a_max, a_size)
    return HH(beta=beta, a_grid=a_grid, z_grid=z_grid, Pi=Pi)
```

we use a CRRA utility function here.

```
@jax.jit
def u(c, gamma=2):
    return jnp.where(c > 0, c ** (1 - gamma) / (1 - gamma), -jnp.inf)
```

finally we define the prices as a namedtuple.

```
Prices = namedtuple("Prices", ("r", "w"))


def create_prices(r=0.01, w=1.0):
    return Prices(r=r, w=w)
```

2.4 Bellman Equation

now we can define the Bellman equation. Consider the value of being in state (a, z) with choosing a' as the next period's asset. The value is:

$$\begin{aligned} B(a, z, a') &= u(wz + (1+r)a - a') + \beta \sum_{z'} v(a', z')\Pi(z, z') \\ &= u(wz + (1+r)a - a') + \beta \mathbb{E}[v(a', z')|z] \end{aligned}$$

```
def B_scalar(v, hh, prices, i, j, ip):
    beta, a_grid, z_grid, Pi = hh
    r, w = prices
    a_size, z_size = len(a_grid), len(z_grid)
    a = a_grid[i]
    z = z_grid[j]
    ap = a_grid[ip]
    c = w * z + (1 + r) * a - ap
    EV = jnp.sum(v[ip, :] * Pi[j, :])
    return jnp.where(c > 0, u(c) + beta * EV, -jnp.inf)
```

We use `vmap` to vectorize the B function.

```
B_1 = jax.vmap(B_scalar, in_axes=(None, None, None, None, None, 0))
B_2 = jax.vmap(B_1, in_axes=(None, None, None, None, 0, None))
B_vmap = jax.vmap(B_2, in_axes=(None, None, None, 0, None, None))
```

Then it's adviced to use `jax.jit` to jit compile the B function.

```
@jax.jit
def B(v, hh, prices):
    beta, a_grid, z_grid, Pi = hh
    a_size, z_size = a_grid.size, z_grid.size
    a_indices = jnp.arange(a_size)
    z_indices = jnp.arange(z_size)
    return B_vmap(v, hh, prices, a_indices, z_indices, a_indices)
```

> ⚠ Warning
>
> Note that we need to use `jax.arange` to create the indices. And we need to use `jax.jit` to jit compile the whole B function (rather than the `B_scalar` or `B_vmap` alone).

Then it's easy to get the greedy policy.

```
@jax.jit
def get_greedy(v, hh, prices):
    return jnp.argmax(B(v, hh, prices), axis=-1)
```

Before we start the VFI, we need to define the Bellman operator T as follows:

```
@jax.jit
def T(v, hh, prices):
    return jnp.max(B(v, hh, prices), axis=-1)
```

Let's then first do the VFI version of this HH problem:

```
@jax.jit
def VFI(hh, prices, max_iter=10000, tol=1e-8):
    def body_fun(k_v_err):
        k, v, err = k_v_err
        vp = T(v, hh, prices)
        err = jnp.max(jnp.abs(vp - v))
        return k + 1, vp, err

    def cond_fun(k_v_err):
        k, v, err = k_v_err
```

```
        return jnp.logical_and(k < max_iter, err > tol)

    sizes = (hh.a_grid.size, hh.z_grid.size)
    k, v, err = jax.lax.while_loop(cond_fun, body_fun, (0, jnp.zeros(sizes), jnp.inf))
    return v, get_greedy(v, hh, prices)
```

Let's then test the VFI function.

```
hh = create_HH()
prices = create_prices()
v_vfi, sigma_vfi = VFI(hh, prices)
```

Let's re-run to see the time used:

```
start_time = time.time()
v_vfi, sigma_vfi = VFI(hh, prices)
v_vfi.block_until_ready()
end_time = time.time()
vfi_time = end_time - start_time
print(f"Time used: {vfi_time} seconds")
```

```
Time used: 0.04320406913757324 seconds
```

It's surprising the VFI is so fast!

2.5 Howard Policy Iteration

Let's then try something faster, that is, the HPI.

> **Algorithm: Howard Policy Iteration**
>
> The HPI alternates between two steps until convergence:
>
> 1. **Policy Evaluation**: Given a policy σ, solve the linear system $R_\sigma v_\sigma = r_\sigma$ to get the exact value function v_σ.
> 2. **Policy Improvement**: Update the policy greedily: $\sigma'(a,z) = \arg\max_{a'} B(a,z,a';v_\sigma)$.
>
> Compared to VFI which applies T one step at a time, HPI jumps directly to the *exact* value of the

> current policy in each iteration — this is why it typically converges in far fewer iterations.

To do this, we first need to define the reward function r_σ given a policy σ.

$$r_\sigma(a, z) = u(wz + (1 + r)a - \sigma(a, z))$$

Given the reward function r_σ, we can get the value function v_σ by solving the following equation:

$$v_\sigma = r_\sigma + \beta \mathbb{E}[v_\sigma | z] = r_\sigma + \beta P_\sigma v_\sigma \tag{1}$$

where P_σ is the transition matrix of the policy σ. Let $n = |a| \times |z|$ be the total number of the states. Then we have $v_\sigma, r_\sigma \in \mathbb{R}^n, P_\sigma \in \mathbb{R}^{n \times n}$.

Transform Equation 1 into a matrix equation:

$$v_\sigma = r_\sigma + \beta P_\sigma v_\sigma$$

we have $v_\sigma = (I - \beta P_\sigma)^{-1} r_\sigma$. Then we can define

$$R_\sigma = I - \beta P_\sigma$$

So that $r_\sigma = R_\sigma v_\sigma$

Note that for state (a, z), we have

$$R_\sigma v_\sigma(a, z) = v_\sigma(a, z) - \beta \sum_{z'} v_\sigma(\sigma(a, z), z') \Pi(z, z') = r_\sigma(a, z)$$

Then we have the code:

```
def R_sigma_scalar(v, hh, prices, sigma, i, j):
    beta, a_grid, z_grid, Pi = hh
    EV = jnp.sum(v[sigma[i, j], :] * Pi[j, :])
    return v[i, j] - beta * EV


R_sigma_1 = jax.vmap(R_sigma_scalar, in_axes=(None, None, None, None, None, 0))
R_sigma_vmap = jax.vmap(R_sigma_1, in_axes=(None, None, None, None, 0, None))
```

```
@jax.jit
def R_sigma(v, hh, prices, sigma):
    a_size, z_size = hh.a_grid.size, hh.z_grid.size
    a_indices = jnp.arange(a_size)
    z_indices = jnp.arange(z_size)
    return R_sigma_vmap(v, hh, prices, sigma, a_indices, z_indices)
```

---

so from $v_\sigma = R_\sigma^{-1} r_\sigma$, we can then solve for the value function v_σ. But first, we need to compute the reward function r_σ.

```
def r_sigma_scalar(hh, prices, sigma, i, j):
    beta, a_grid, z_grid, Pi = hh
    r, w = prices
    a = a_grid[i]
    z = z_grid[j]
    ap = a_grid[sigma[i, j]]
    c = w * z + (1 + r) * a - ap
    return jnp.where(c > 0, u(c), -jnp.inf)


r_sigma_1 = jax.vmap(r_sigma_scalar, in_axes=(None, None, None, None, 0))
r_sigma_vmap = jax.vmap(r_sigma_1, in_axes=(None, None, None, 0, None))


@jax.jit
def r_sigma(hh, prices, sigma):
    a_size, z_size = hh.a_grid.size, hh.z_grid.size
    a_indices = jnp.arange(a_size)
    z_indices = jnp.arange(z_size)
    return r_sigma_vmap(hh, prices, sigma, a_indices, z_indices)
```

All prepared, let's solve for the value function v_σ.

```python
@jax.jit
def get_v_sigma(hh, prices, sigma):
    r_sigma_value = r_sigma(hh, prices, sigma)
    _R_sigma = lambda v: R_sigma(v, hh, prices, sigma)
    return jax.scipy.sparse.linalg.bicgstab(_R_sigma, r_sigma_value)[0]
```

> **Why `bicgstab` instead of direct inversion?**
>
> R_σ is an $n \times n$ matrix where $n = |a| \times |z|$. With $|a| = 400$ and $|z| = 2$, we have $n = 800$, so $R_\sigma \in \mathbb{R}^{800\times800}$ — direct inversion is feasible here.
> However, in general n can be very large, so we use an iterative solver (`bicgstab`) instead. Crucially, we never need to *materialize* R_σ as a matrix — we only need the **linear map** $v \mapsto R_\sigma v$, which is what `_R_sigma` provides. This makes the approach memory-friendly for larger state spaces.

So here is the Howard policy iteration.

```python
from jax import debug


@jax.jit
def HPI_loop(hh, prices, max_iter=10000, tol=1e-8):
    def body_fun(k_v_err):
        k, v, err = k_v_err
        sigma_p = get_greedy(v, hh, prices)
        v_sigma_p = get_v_sigma(hh, prices, sigma_p)
        err = jnp.max(jnp.abs(v_sigma_p - v))
        return k + 1, v_sigma_p, err

    def cond_fun(k_v_err):
        k, v_sigma, err = k_v_err
        return jnp.logical_and(k < max_iter, err > tol)

    sizes = (hh.a_grid.size, hh.z_grid.size)
    k, v_sigma, err = jax.lax.while_loop(cond_fun, body_fun, (0, jnp.zeros(sizes), jnp.inf))
    return v_sigma, get_greedy(v_sigma, hh, prices)
```

Let's then test the HPI function.

```
hh = create_HH()
prices = create_prices()
v_sigma, sigma = HPI_loop(hh, prices)
```

Let's re-run to see the time used:

```
start_time = time.time()
v_sigma_hpi, sigma_hpi = HPI_loop(hh, prices)
v_sigma_hpi.block_until_ready()
end_time = time.time()
hpi_time = end_time - start_time
print(f"Time used: {hpi_time} seconds")
print(f"Speedup: {vfi_time / hpi_time}")
```

```
Time used: 0.006680965423583984 seconds
Speedup: 6.466740418242809
```

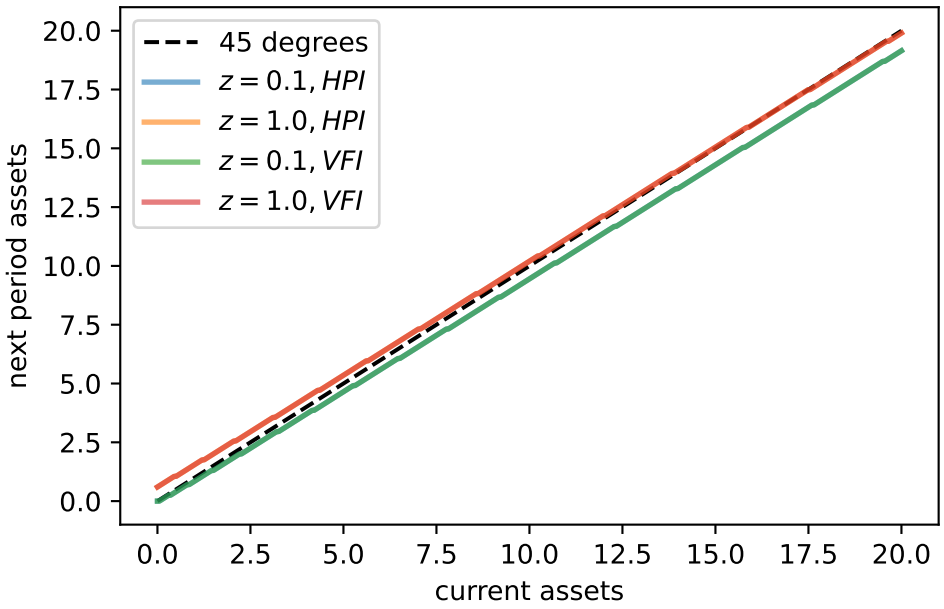
Let's look at the result of the VFI and HPI.

```
hh = create_HH()

fig, ax = plt.subplots()
ax.plot(hh.a_grid, hh.a_grid, "k--", label="45 degrees")
for j, z in enumerate(hh.z_grid):
    lb = f"$z = {z:.2}, HPI$"
    policy_vals = hh.a_grid[sigma_hpi[:, j]]
    ax.plot(hh.a_grid, policy_vals, lw=2, alpha=0.6, label=lb)
    ax.set_xlabel("current assets")
    ax.set_ylabel("next period assets")

for j, z in enumerate(hh.z_grid):
    lb = f"$z = {z:.2}, VFI$"
    policy_vals = hh.a_grid[sigma_vfi[:, j]]
    ax.plot(hh.a_grid, policy_vals, lw=2, alpha=0.6, label=lb)

ax.legend(loc="upper left")
plt.show()
```



Check if they are the same:

```
print(jnp.allclose(sigma_hpi, sigma_vfi))
```

```
True
```

2.6 Endogenous Grid Method

Let's then try a third way to solve the household problem: the **endogenous grid method (EGM)**.

The main idea is different from both VFI and HPI. Instead of iterating on the Bellman equation and repeatedly maximizing over a', we iterate directly on the **consumption policy** using the Euler equation.

For the household problem

$$a' + c = wz + (1 + r)a, \qquad a' \geq a_{\min},$$

the Euler equation on the interior is

$$u'(c(a, z)) = \beta(1 + r) \sum_{z'} u'(c(a', z'))\Pi(z, z').$$

Suppose we already have an approximation $c_n(a, z)$ to the consumption policy.
Fix an **exogenous grid for next period assets** $a' \in A$. For each pair (a', z), define

$$\mathscr{M}_n(a', z) := \sum_{z'} u'(c_n(a', z'))\Pi(z, z').$$

Then the Euler equation implies

$$c_{n+1}^{\text{endo}}(a', z) = (u')^{-1}\left(\beta(1+r)\mathscr{M}_n(a', z)\right).$$

Given this consumption choice and the budget constraint, the associated **current** asset level must satisfy

$$a_{n+1}^{\text{endo}}(a', z) = \frac{c_{n+1}^{\text{endo}}(a', z) + a' - wz}{1+r}.$$

So for each z, the pairs

$$\left(a_{n+1}^{\text{endo}}(a', z), a'\right)$$

form an endogenous grid. Once we have these points, we interpolate back to the original asset grid and recover the policy $a' = g(a, z)$.

If a is below the first endogenous grid point, then the borrowing / saving constraint binds and we set $a' = a_{\min}$. In the code below, this is handled automatically by the left endpoint of `jnp.interp`.

> **Algorithm: EGM**
>
> 1. Start from a guess of the consumption policy $c_n(a, z)$.
> 2. For each exogenous grid point a' and current shock z, compute expected marginal utility.
> 3. Invert the Euler equation to get $c_{n+1}^{\text{endo}}(a', z)$.
> 4. Use the budget constraint to infer the endogenous current asset grid $a_{n+1}^{\text{endo}}(a', z)$.
> 5. Interpolate from the endogenous grid back to the original asset grid to obtain $g(a, z)$ and then recover consumption.
>
> Compared to VFI and HPI, EGM avoids the `argmax` step entirely, which is why it is often much faster in one-asset incomplete-markets problems.

We keep the same CRRA specification as before, so the marginal utility and its inverse are:

```python
@jax.jit
def u_prime(c, gamma=2):
    return jnp.maximum(c, 1e-14) ** (-gamma)


@jax.jit
def u_prime_inv(up, gamma=2):
    return jnp.maximum(up, 1e-14) ** (-1 / gamma)
```

For each productivity state, we need to interpolate from the endogenous current-asset grid back to the original exogenous grid.

```python
def interp_policy_col(a_vals, a_endo_col, sigma_a_col):
    return jnp.interp(a_vals, a_endo_col, sigma_a_col)


interp_policy = jax.vmap(interp_policy_col, in_axes=(1, 1, 1), out_axes=1)
```

2.7 Role of `vmap` in `interp_policy`

`interp_policy_col` handles **a single column only**, i.e., interpolation for a single z (productivity) state:

- `a_vals`: the original asset grid, shape `(n,)`
- `a_endo_col`: the endogenous asset grid (current z column), shape `(n,)`
- `sigma_a_col`: the corresponding savings policy (current z column), shape `(n,)`

All three input matrices have shape `(n_a, n_z)`. `in_axes=(1, 1, 1)` means that we slice along **axis 1 (columns)**, i.e., for each z state, we pass one column at a time to `interp_policy_col`. `out_axes=1` then stacks all column results along the columns, so the final output is again shape `(n_a, n_z)`.

This is equivalent to interpolating separately for each z_j:

$$g(a, z_j) = \text{interp}\Big(a,\ a^{\text{endo}}(\cdot, z_j),\ a'(\cdot, z_j)\Big), \quad j = 1, \dots, n_z$$

But `vmap` replaces a Python for loop with a **single vectorized operation** for efficiency and much better performance on GPU/TPU.

Now we can write one EGM update.

```python
@jax.jit
def EGM_step(sigma_c, hh, prices):
    """
    Perform one EGM (Endogenous Grid Method) iteration step for the consumption-savings policy.
    """
    beta, a_grid, z_grid, Pi = hh
    r, w = prices
    R = 1 + r  # Gross interest rate

    # Compute expected marginal utility for next period
    Emu = u_prime(sigma_c) @ Pi.T  # (n_a, n_z) @ (n_z, n_z) → (n_a, n_z)
    # Back out consumption on the endogenous grid using inverse marginal utility
    c_endo = u_prime_inv(beta * R * Emu)

    # a_prime: Future asset grid, shape (n_a, n_z)
    a_prime = jnp.broadcast_to(a_grid[:, jnp.newaxis], c_endo.shape)
    # Map to endogenous current-asset grid
    a_endo = (c_endo + a_prime - w * z_grid[jnp.newaxis, :]) / R

    # Current asset grid (used for interpolation)
    a_now = jnp.broadcast_to(a_grid[:, jnp.newaxis], c_endo.shape)
    # Interpolate savings policy from endogenous to standard grid
    sigma_a = interp_policy(a_now, a_endo, a_prime)
    # Ensure no borrowing: impose lower bound on assets
    sigma_a = jnp.maximum(sigma_a, a_grid[0])

    # Update the consumption policy using the budget constraint
    sigma_c_new = w * z_grid[jnp.newaxis, :] + R * a_now - sigma_a
    return sigma_a, sigma_c_new
```

Then we iterate on the consumption policy until convergence.

```python
@jax.jit
def EGM(hh, prices, max_iter=10_000, tol=1e-8):
    beta, a_grid, z_grid, Pi = hh
    r, w = prices
    R = 1 + r
    a_min = a_grid[0]
```

```
    a_now = a_grid[:, jnp.newaxis]
    sigma_c_init = w * z_grid[jnp.newaxis, :] + R * a_now - a_min

    def body_fun(k_c_err):
        k, sigma_c, err = k_c_err
        sigma_a_new, sigma_c_new = EGM_step(sigma_c, hh, prices)
        err = jnp.max(jnp.abs(sigma_c_new - sigma_c))
        return k + 1, sigma_c_new, err

    def cond_fun(k_c_err):
        k, sigma_c, err = k_c_err
        return jnp.logical_and(k < max_iter, err > tol)

    k, sigma_c, err = jax.lax.while_loop(
        cond_fun,
        body_fun,
        (0, sigma_c_init, jnp.inf),
    )
    sigma_a, sigma_c = EGM_step(sigma_c, hh, prices)
    return sigma_a, sigma_c
```

Let's test the EGM solver.

```
hh = create_HH()
prices = create_prices()
sigma_a_egm, sigma_c_egm = EGM(hh, prices)
```

And let's check the runtime:

```
start_time = time.time()
sigma_a_egm, sigma_c_egm = EGM(hh, prices)
sigma_a_egm.block_until_ready()
end_time = time.time()
egm_time = end_time - start_time
print(f"Time used: {egm_time} seconds")
print(f"Speedup vs VFI: {vfi_time / egm_time}")
print(f"Speedup vs HPI: {hpi_time / egm_time}")
```

```
Time used: 0.003384113311767578 seconds
Speedup vs VFI: 12.766732422150204
Speedup vs HPI: 1.9742144568127378
```

Because EGM returns a continuous asset policy rather than an index-valued policy, it is natural to compare it to the HPI asset policy `hh.a_grid[sigma_hpi]`.

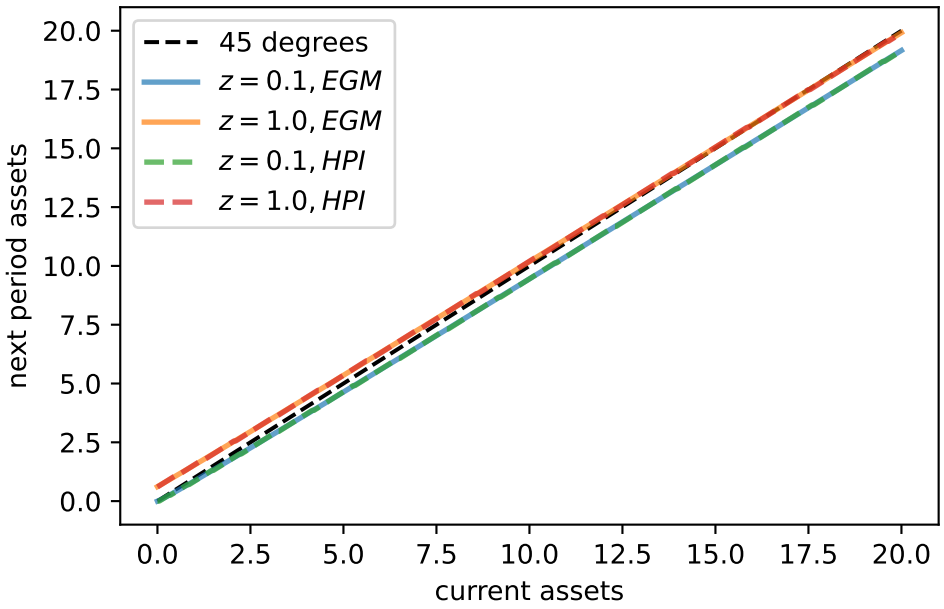
Let's plot the two policy functions together.

```
sigma_a_hpi = hh.a_grid[sigma_hpi]
fig, ax = plt.subplots()
ax.plot(hh.a_grid, hh.a_grid, "k--", label="45 degrees")

for j, z in enumerate(hh.z_grid):
    lb = f"$z = {z:.2}, EGM$"
    ax.plot(hh.a_grid, sigma_a_egm[:, j], lw=2, alpha=0.7, label=lb)

for j, z in enumerate(hh.z_grid):
    lb = f"$z = {z:.2}, HPI$"
    ax.plot(hh.a_grid, sigma_a_hpi[:, j], "--", lw=2, alpha=0.7, label=lb)

ax.set_xlabel("current assets")
ax.set_ylabel("next period assets")
ax.legend(loc="upper left")
plt.show()
```



So in this Aiyagari household problem:

- VFI and HPI solve for the value function first, and then recover the policy.
- EGM solves directly for the policy using the Euler equation.
- EGM is especially attractive here because the choice variable is one-dimensional and CRRA utility makes the Euler equation easy to invert.

For the equilibrium section below, we keep using HPI because the distribution code is currently written for an **index-valued** policy $\sigma(a, z) \in \{0, 1, \ldots, |A| - 1\}$. To use EGM all the way through, we would need to modify the law of motion so that probability mass is split across neighboring grid points after interpolation.

2.8 Equilibrium

We need to know how much capital households supply at a given interest rate r.

After we have the policy σ, we now know the transition matrix P_σ where

$$P_\sigma(i, j) = \Pr\{a_{t+1} = a_j, z_{t+1} = z_j | a_t = a_i, z_t = z_i\}$$

In quantecon's tutorial here, they use this following code:

```jax.jit
def compute_asset_stationary(σ, household):
    # Unpack
    β, a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

    # Construct P_σ as an array of the form P_σ[i, j, ip, jp]
    ap_idx = jnp.arange(a_size)
    ap_idx = jnp.reshape(ap_idx, (1, 1, a_size, 1))
    σ = jnp.reshape(σ, (a_size, z_size, 1, 1))
    A = jnp.where(σ == ap_idx, 1, 0)
    Π = jnp.reshape(Π, (1, z_size, 1, z_size))
    P_σ = A * Π

    # Reshape P_σ into a matrix
    n = a_size * z_size
    P_σ = jnp.reshape(P_σ, (n, n))

    # Get stationary distribution and reshape back onto [i, j] grid
    ψ = compute_stationary(P_σ)
    ψ = jnp.reshape(ψ, (a_size, z_size))

    # Sum along the rows to get the marginal distribution of assets
    ψ_a = jnp.sum(ψ, axis=1)
    return ψ_a
```

2.9 Power Iteration

But let's do a slightly different version.

That is, we'll do the power iteration to get the stationary distribution instead of solving the transition matrix first, this is memeory-friendly cuz if we have a large number of states, the transition matrix will be too large to store in memory.

```jax.jit
def push_forward(psi, sigma, hh):
    I, J = hh.a_grid.size, hh.z_grid.size
    psi_next = jnp.zeros_like(psi)

    def body_fun(k, psi_next):
        i, j = jnp.unravel_index(k, (I, J))
        ip = sigma[i, j]
```

```
        psi_next = psi_next.at[ip, :].add(psi[i, j] * hh.Pi[j, :])
        return psi_next

    psi_next = jax.lax.fori_loop(0, I * J, body_fun, psi_next)
    return psi_next
```

2.10 Stationary Distribution

Now we can get the stationary distribution by iterating the `push_forward` function until it converges.

```
@jax.jit
def get_stat_asset(sigma, hh, tol=1e-12, max_iter=10_000):
    I, J = hh.a_grid.size, hh.z_grid.size
    # we make it a uniform distribution initially
    psi = jnp.ones((I, J)) / (I * J)

    def body_fun(k_psi_err):
        k, psi, err = k_psi_err
        psi_next = push_forward(psi, sigma, hh)
        err = jnp.max(jnp.abs(psi_next - psi))
        return k + 1, psi_next, err

    def cond_fun(k_psi_err):
        k, psi, err = k_psi_err
        return jnp.logical_and(k < max_iter, err > tol)

    init_val = (0, psi, jnp.inf)
    k, psi, err = jax.lax.while_loop(cond_fun, body_fun, init_val)
    return psi
```

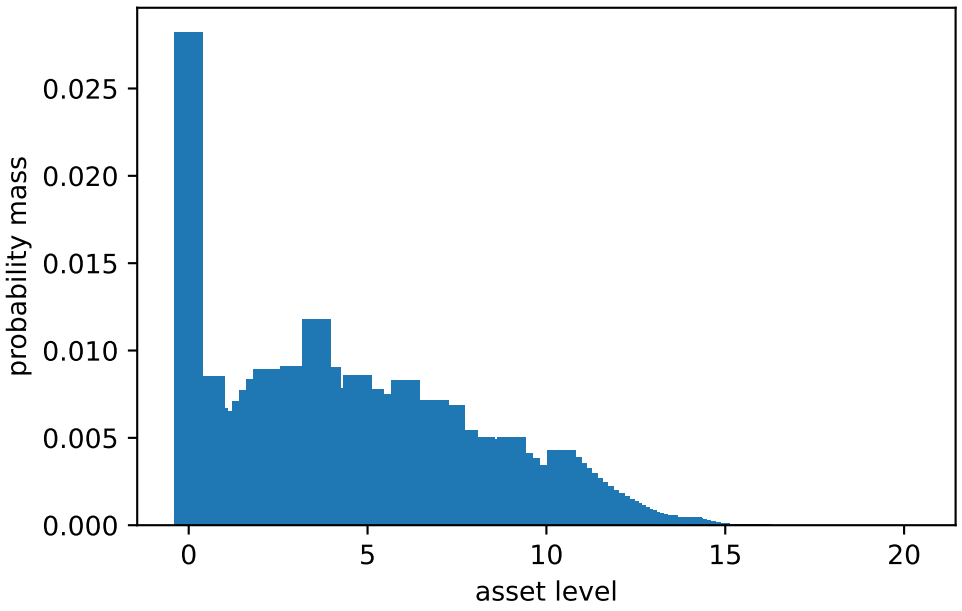
Now it's time to test the power iteration method.

```
psi_iter = get_stat_asset(sigma_hpi, hh)
psi_iter_asset = jnp.sum(psi_iter, axis=1)
psi_solve_asset = compute_asset_stationary(sigma_hpi, hh)
print(jnp.allclose(psi_iter_asset, psi_solve_asset))
print(jnp.max(jnp.abs(psi_iter_asset - psi_solve_asset)))
```

```
True
2.5466139613739003e-11
```

Let's check the distribution now:

```
fig, ax = plt.subplots()
ax.bar(hh.a_grid, psi_iter_asset)
ax.set_xlabel("asset level")
ax.set_ylabel("probability mass")
plt.show()
```



Check if it sums to one:

```
print(jnp.sum(psi_iter_asset))
```

```
1.0000000000000147
```

Now we can get the aggregate capital supply by households.

```
@jax.jit
def capital_supply(sigma, hh):
    psi_asset = get_stat_asset(sigma, hh).sum(axis=1)
    return jnp.sum(psi_asset * hh.a_grid)
```

Let's test the capital supply function:

```
print(capital_supply(sigma_hpi, hh))
```

```
5.417634374417349
```

Alright, then remeber what we have in `r_given_k` and `w_given_r`, we can now write the equilibrium condition.

Let's consider the mapping $K_{n+1} = G(K_n)$, where $G(K)$ is the capital supply function.

That is

```
@jax.jit
def G(K, firm, hh):
    r = r_given_k(K, firm)
    w = w_given_r(r, firm)
    prices = create_prices(r=r, w=w)
    _, sigma = HPI_loop(hh, prices)
    return capital_supply(sigma, hh)
```

Let's test the `G` function:

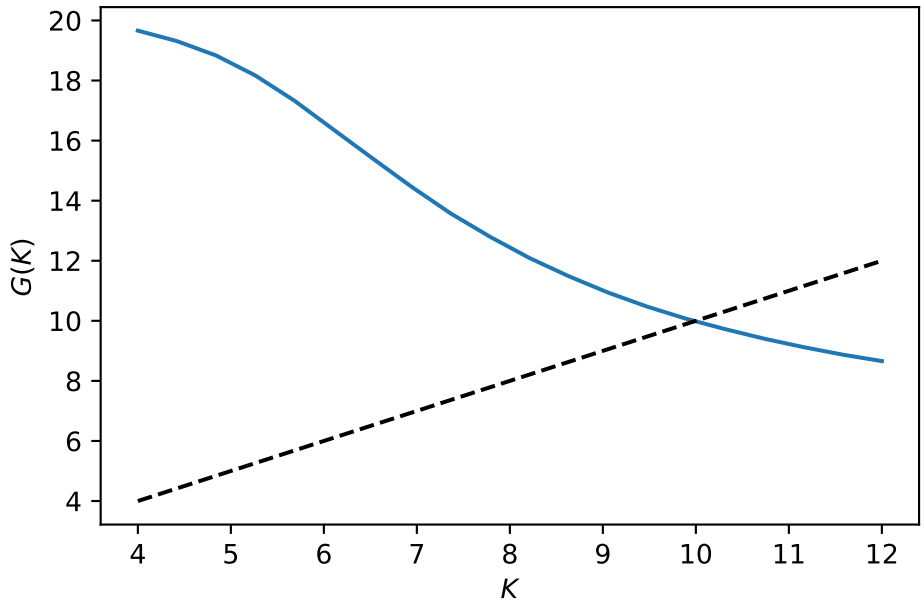
```
k_vals = jnp.linspace(4, 12, 20)
firm = create_firm()
hh = create_HH()

G_vals = [G(k, firm, hh) for k in k_vals]
G_vals[-1].block_until_ready()
```

```
Array(8.65889621, dtype=float64)
```

Let's plot the `G` function:

```
fig, ax = plt.subplots()
ax.plot(k_vals, G_vals)
ax.plot(k_vals, k_vals, "k--", label="45 degrees")
ax.set_xlabel("$K$")
ax.set_ylabel("$G(K)$")
plt.show()
```



So a natural idea is to use the damped iteration scheme

$$K_{n+1} = \alpha K_n + (1 - \alpha) G(K_n)$$

where α is a damping factor.

```
@jax.jit
def equilibrium_damped(K0, firm, hh, alpha=0.6, max_iter=100, tol=1e-8):
    def body_fun(k_K_err):
        k, K, err = k_K_err
        K_next = alpha * K + (1 - alpha) * G(K, firm, hh)
        err = jnp.abs(K_next - K)
        return k + 1, K_next, err
```

```
    def cond_fun(k_K_err):
        k, K, err = k_K_err
        return jnp.logical_and(k < max_iter, err > tol)

    init_val = (0, K0, jnp.inf)
    k, K, err = jax.lax.while_loop(cond_fun, body_fun, init_val)
    return K
```

Let's test the `equilibrium_damped` function:

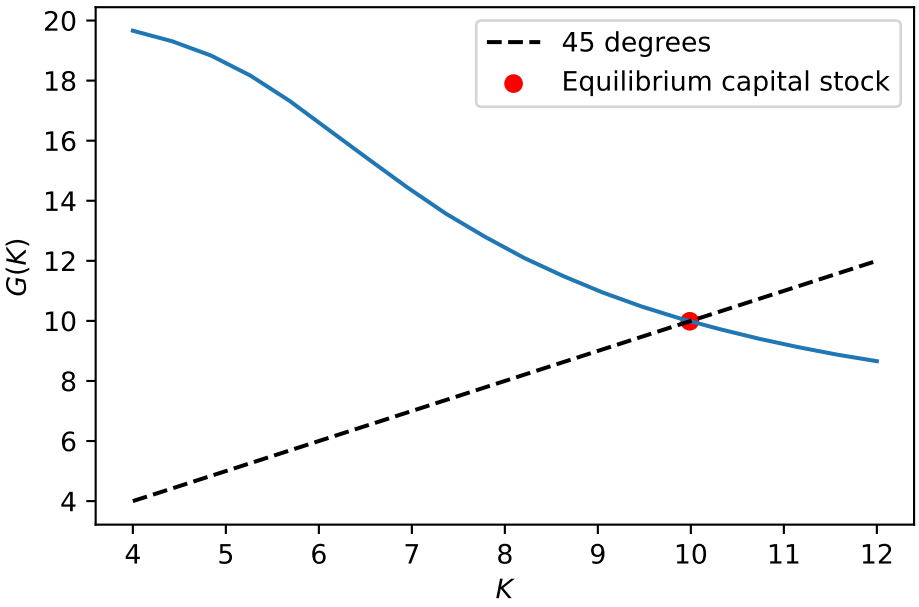
```
K0 = 6
firm = create_firm()
hh = create_HH()
K_star = equilibrium_damped(K0, firm, hh)
print(f"Equilibrium capital stock: {K_star}")

start_time = time.time()
K_star = equilibrium_damped(K0, firm, hh).block_until_ready()
end_time = time.time()
damped_time = end_time - start_time
print(f"Time used: {damped_time} seconds")
```

```
Equilibrium capital stock: 9.988106877194669
Time used: 0.20470809936523438 seconds
```

Let's plot to check the result!

```
fig, ax = plt.subplots()
ax.plot(k_vals, G_vals)
ax.plot(k_vals, k_vals, "k--", label="45 degrees")
ax.scatter(K_star, G(K_star, firm, hh), color="red", label="Equilibrium capital stock")
ax.set_xlabel("$K$")
ax.set_ylabel("$G(K)$")
ax.legend()
plt.show()
```


45 degrees
Equilibrium capital stock
G(K)
K
20
18
16
14
12
10
8
6
4
4
5
6
7
8
9
10
11
12