Aiyagari Model in JAX

Chen Gao

2026-01-08

Table of contents

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")
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels inste
```

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.

```
@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]
```

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.

```
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))
```

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:

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

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)
```

> ⚠ Warning
>
> In this version of code, the HPI falls when we have some tricky `a_size` like 500 or 1000, I don't know why but it seems that currently this is enough.

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)
```

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:

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

We use `vmap` to vectorize the B function.

```
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)
```

Then we can 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.

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)
```

Note that we need to use `jax.lax.while_loop` to implement the while loop. And we still need to use `jax.jit` to jit compile the VFI function.

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: 0.04872488975524902 seconds
```

It's surprising the VFI is so fast! Let's then try something nicer, that is, the HPI.

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

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

Similarly, we do the vectorization and jit compilation using `vmap` and `jax.jit`.

```
def r_sigma_scalar(v, 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 u(c)


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)
```

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:

```
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.

```
@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]
```

So here is the Howard policy iteration.

```
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)
```

> **Note**
>
> We use `jax.lax.cond` to print the progress of the HPI loop. This is basically a `if` statement in Python. The reason why we need to use `jax.lax.cond` is because Python needs to convert it to true/false, but it's a tracer (which only knows at runtime).

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()
```

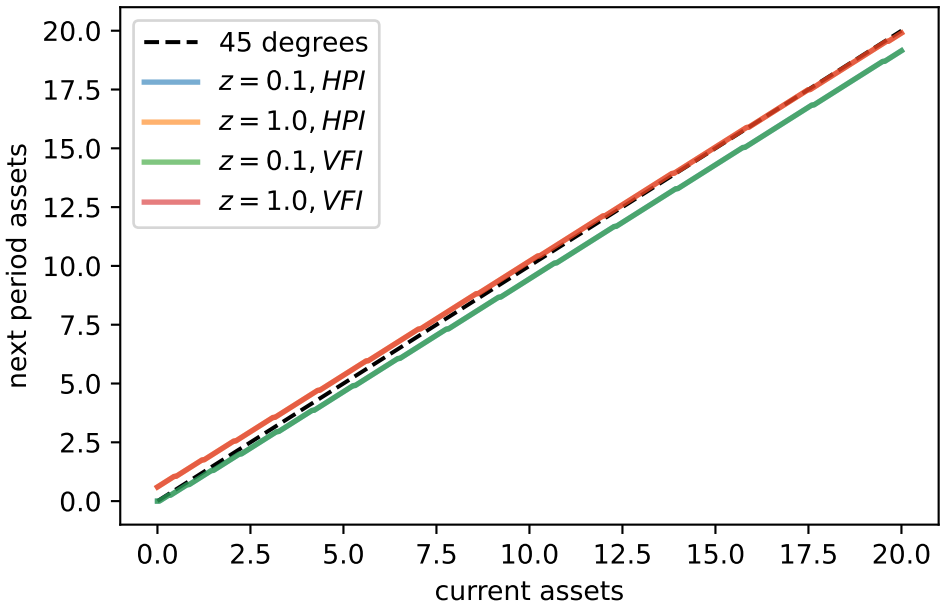
```
hpi_time = end_time - start_time
print(f"Time used: {hpi_time} seconds")
print(f"Speedup: {vfi_time / hpi_time}")
```

```
Time used: 0.007254123687744141 seconds
Speedup: 6.716854006441859
```

We can see that the HPI is much faster than the VFI. But given this super fast VFI, the HPI is not that much faster.

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
```

That's nice!

Think about the equilibrium

Now we need to think about the equilibrium of the model.

- 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( , household):
    # Unpack
     , a_grid, z_grid, Π = household
    a_size, z_size = len(a_grid), len(z_grid)

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

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

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

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

But I'm so bad at broadcasting, so I'll do a slightly different version. That is, I'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
```

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
```

This is something boring but let's see how fast they are:

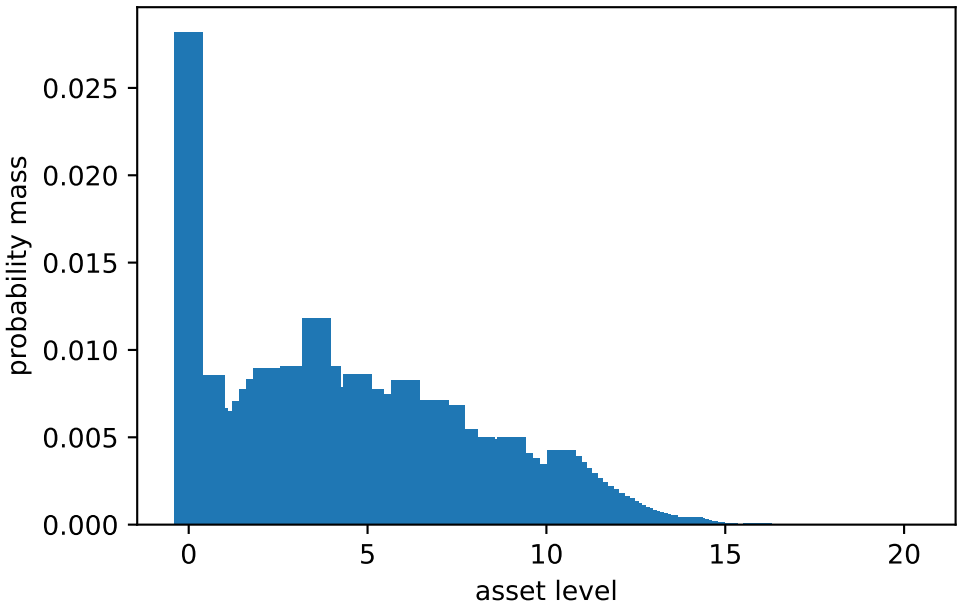
```
start_time = time.time()
psi_iter = get_stat_asset(sigma_hpi, hh)
psi_iter.block_until_ready()
end_time = time.time()
power_iter_time = end_time - start_time
print(f"Power iteration time used: {power_iter_time} seconds")
start_time = time.time()
psi_solve = compute_asset_stationary(sigma_hpi, hh)
psi_solve.block_until_ready()
end_time = time.time()
solve_time = end_time - start_time
print(f"Solve time used: {solve_time} seconds")
print(f"Speedup: {solve_time / power_iter_time}")
```

```
Power iteration time used: 0.002087116241455078 seconds
Solve time used: 0.002167940139770508 seconds
Speedup: 1.0387251542152158
```

We can see that power iteration is faster than solving the transition matrix first.

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
```

Nice! 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.

Using the slow version in quantecon website, we 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)
```

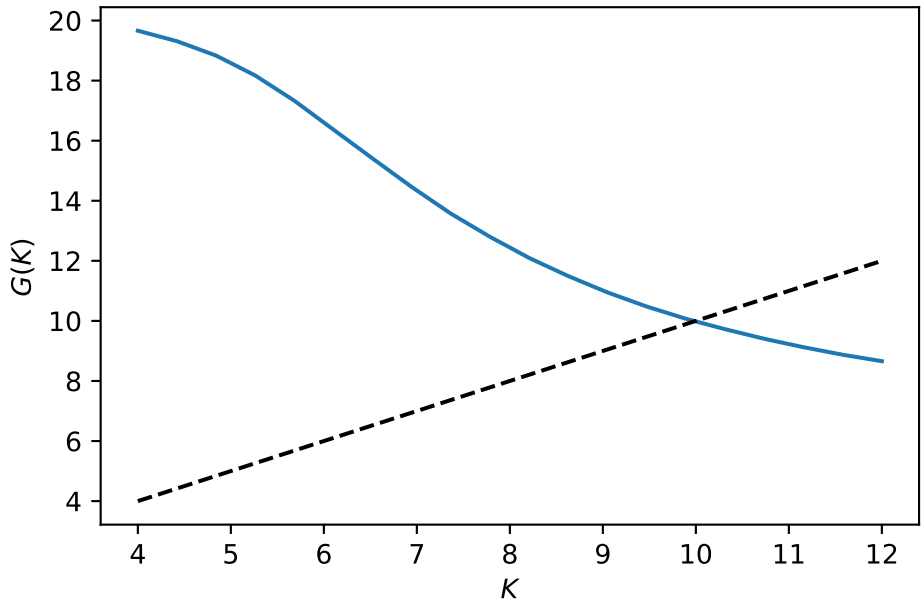
Let's test the `G` function:

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

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

end_time = time.time()
print(f"Time used: {end_time - start_time} seconds")
```

```
Time used: 10.686196088790894 seconds
```

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(static_argnums=(6,))
def equilibrium_damped(K0, firm, hh, alpha=0.6, max_iter=100, tol=1e-8, verbose=True):
    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)
        if verbose:
            jax.lax.cond(
                k % 5 == 0,
                lambda _: debug.print(
                    "Concluded loop {k} with error {err}.", k=k, err=err
                ),
                lambda _: None,
                operand=None,
            )
        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:

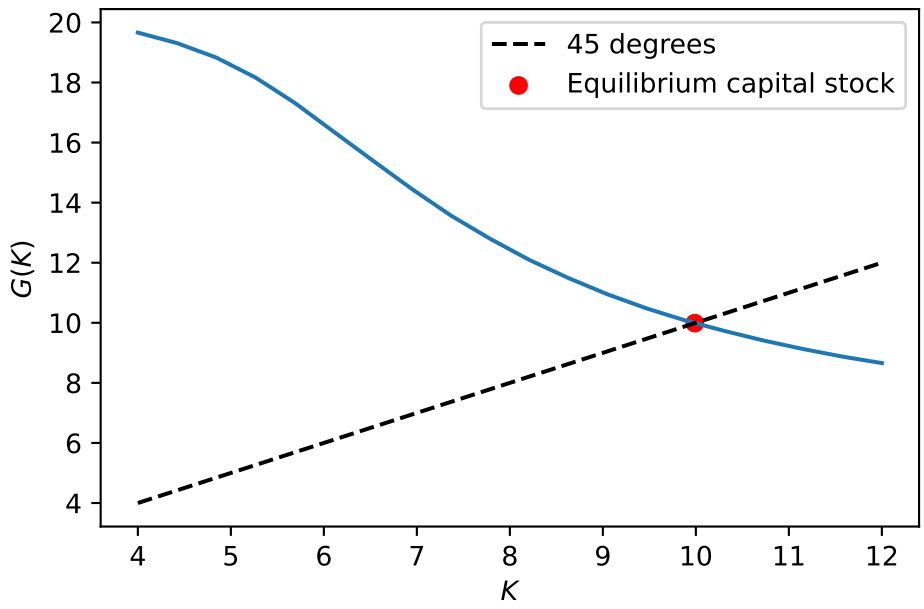
```
K0 = 6
firm = create_firm()
hh = create_HH()
K_star = equilibrium_damped(K0, firm, hh, verbose=True)
print(f"Equilibrium capital stock: {K_star}")

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

```
Concluded loop 0 with error 4.248556554763983.
Concluded loop 5 with error 0.00023333847830642185.
Concluded loop 10 with error 1.814440007308349e-05.
Concluded loop 15 with error 1.410908549104306e-06.
Concluded loop 20 with error 1.0971224817524217e-07.
Concluded loop 25 with error 8.531223727459292e-09.
Equilibrium capital stock: 9.988106877194669
Concluded loop 0 with error 4.248556554763983.
Concluded loop 5 with error 0.00023333847830642185.
Concluded loop 10 with error 1.814440007308349e-05.
Concluded loop 15 with error 1.410908549104306e-06.
Concluded loop 20 with error 1.0971224817524217e-07.
Concluded loop 25 with error 8.531223727459292e-09.
Time used: 0.20618796348571777 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()
```



So this is pretty fast, but we can do better.

Let's use Bisection to find the equilibrium capital stock.

```
import scipy.optimize


def equilibrium_bisection(firm, hh, tol=1e-8):
    @jax.jit
    def f(K):
        return G(K, firm, hh) - K

    return scipy.optimize.bisect(f, 4, 12, xtol=tol)
```

Let's test the `equilibrium_bisection` function:

```
K_star = equilibrium_bisection(firm, hh)
print(f"Equilibrium capital stock: {K_star}")
```

```
start_time = time.time()
K_star = equilibrium_bisection(firm, hh)
end_time = time.time()
bisection_time = end_time - start_time
print(f"Time used: {bisection_time} seconds")
print(f"Speedup: {damped_time / bisection_time}")
```

```
Equilibrium capital stock: 9.98810688406229
Time used: 3.729807138442993 seconds
Speedup: 0.05528113273218488
```

> **Warning**
>
> Why is it so slow? The reason is that we called the `f` function 32 times to find the equilibrium capital stock. And this is a big overhead because we need to switch to Python to call the `scipy.optimize.bisect` function and this is very slow.

Is it possible to make it faster? Let's try to write a bisection method by hand in Jax.

```
def bisection_jax(f, a, b, tol=1e-8, max_iter=1000):
    fa, fb = f(a), f(b)

    def cond_fun(i_a_b_fa_fb):
        i, a, b, fa, fb = i_a_b_fa_fb
        return jnp.logical_and(i < max_iter, b - a > tol)

    def body_fun(i_a_b_fa_fb):
        i, a, b, fa, fb = i_a_b_fa_fb
        m = (a + b) / 2
        fm = f(m)
        a_next = jnp.where(fa * fm < 0, a, m)
        fa_next = jnp.where(fa * fm < 0, fa, fm)
        b_next = jnp.where(fa * fm < 0, m, b)
        fb_next = jnp.where(fa * fm < 0, fb, fm)
        return i + 1, a_next, b_next, fa_next, fb_next

    init_val = (0, a, b, fa, fb)
    i, a, b, fa, fb = jax.lax.while_loop(cond_fun, body_fun, init_val)
    return (a + b) / 2
```

Let's test the `bisection_jax` function:

```python
@jax.jit
def equilibrium_bisection_jax(firm, hh, tol=1e-8):
    def f(K):
        return G(K, firm, hh) - K

    return bisection_jax(f, 4, 12, tol=tol)


K_star = equilibrium_bisection_jax(firm, hh).block_until_ready()
print(f"Equilibrium capital stock: {K_star}")

start_time = time.time()
K_star = equilibrium_bisection_jax(firm, hh).block_until_ready()
end_time = time.time()
bisection_jax_time = end_time - start_time
print(f"Time used: {bisection_jax_time} seconds")
print(f"Jax Speedup: {bisection_time/ bisection_jax_time}")
print(f"Bisection Speedup: {damped_time/ bisection_jax_time}")
```

```
Equilibrium capital stock: 9.98810688778758
Time used: 0.2413489818572998 seconds
Jax Speedup: 15.45399988738416
Bisection Speedup: 0.854314619017654
```

> **Tip**
>
> We can use the `optimistix` lib to find the equilibrium capital stock. But it's not that much faster and even worse at this task.

So the result is, using a damped iteration scheme is the fastest method to find the equilibrium capital stock!