EGM using JAX

Chen Gao

2026-01-07

Before going to NN, let's look at the basic EGM algorithm.

Consider this problem:

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

subject to:

$$a_{t+1} \leq R(a_t - c_t) + y_{t+1}, \quad c_t \geq 0, \quad a_t \geq 0 \quad t = 0, 1, \ldots$$

where $\{y_t\}$ is a Markov chain with transition matrix P.

$$\log y_{t+1} = \rho \log y_t + \nu \epsilon_{t+1}$$

where $\epsilon_{t+1} \sim N(0, 1)$.

We use a CRRA utility function:

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

Define the model:

```
import jax
import jax.numpy as jnp
import quantecon as qe
import time
import matplotlib.pyplot as plt
import matplotlib

matplotlib.rcParams["text.usetex"] = True
matplotlib.rcParams["text.latex.preamble"] = r"\usepackage{libertinus}"

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

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

Now let's define the model:

```
@jax.jit
def create_model(
    beta=0.97,
    R=1.01,
    gamma=2.0,
    s_max=20.0,
    s_min=1e-10,
    s_size=1000,
    rho=0.99,
    nu=0.02,
    y_size=25,
):
    # require R   < 1 for convergence
    assert R * beta < 1, "Stability condition failed."
    # note that I use saving here instead of assets.
    s_grid = jnp.linspace(s_min, s_max, s_size)
    mc = qe.tauchen(n=y_size, rho=rho, sigma=nu)
    y_grid, Q = jnp.exp(mc.state_values), jnp.array(mc.P)
    params = (beta, R, gamma)
    sizes = (s_size, y_size)
    arrays = (s_grid, y_grid, Q)
    return params, sizes, arrays
```

> **!** Important
>
> Note that I use saving here instead of assets. This is crucial in the idea of EGM.

Alright, now let's look at the problem itself.

- State: $(a_t, y_t) \in \mathcal{S} := \mathcal{A} \times \mathcal{Y}$
- Action: $\sigma^* : \mathcal{S} \to \mathbb{R}_+$ where

$$c = \sigma^*(a, Y), a' = R(a - c) + y' = R(a - \sigma^*(a, y)) + y'$$

Now let's consider the Lagrangian:

$$\mathscr{L} = \mathbb{E} \sum_{t=0}^{\infty} \beta^t \left[u(c_t) + \lambda_t (a_{t+1} - R(a_t - c_t) - y_{t+1}) + \mu_t (a_t - c_t) \right]$$

Taking foc wrt c_t:

$$\frac{\partial \mathscr{L}}{\partial c_t} = \beta^t \left[u'(c_t) + \lambda_t R - \mu_t\right] = 0$$

That is, $u'(c_t) = \mu_t - \lambda_t R$.

Taking foc wrt a_{t+1}:

$$\frac{\partial \mathscr{L}}{\partial a_{t+1}} = \beta^t \lambda_t - \beta^{t+1}\lambda_{t+1} R + \beta^{t+1}\mu_{t+1} = 0$$

That is, $\lambda_t = \beta \mathbb{E}_t[\lambda_{t+1} R - \mu_{t+1}] = -\beta \mathbb{E}_t[u'(c_{t+1})]$, so

$$u'(c_t) - \mu_t = \beta R \mathbb{E}_t[u'(c_{t+1})]$$

or equivalently:

$$u'(c_t) = \mu_t + \beta R \mathbb{E}_t[u'(c_{t+1})] \geq \beta R \mathbb{E}_t[u'(c_{t+1})]$$

and the equality strictly holds if $\mu_t = 0 \Rightarrow a_t > c_t$.

so we can write the Euler equation as:

$$u'(c_t) = \max\{u'(a_t), \beta R \mathbb{E}_t[u'(c_{t+1})]\}$$

For simplicity, we remove the time index t. Then we have:

$$u'(c) = \max\{u'(a), \beta R \mathbb{E}[u'(c')]\} \tag{1}$$

substituting $c = \sigma(a, y)$,$c' = R(a - c) + y'$ into Equation 1, we get:

$$(u' \circ \sigma)(a, y) = \max\{u'(a), \beta R \mathbb{E}[(u' \circ \sigma)(R(a - \sigma(a, y)) + y', y')]\}$$

Given mapping σ, we can define the mapping $K\sigma : \mathcal{S} \to [0, a]$ as the $c \in [0, a]$ that solves:

$$u'(c) = \max\{u'(a), \beta R \mathbb{E}[(u' \circ \sigma)(R(a - c) + y', y')]\} \tag{2}$$

For a that's small enough, we have $u'(a) \to \infty$, so the second term in the max dominates, and we have $c = a$. And if a is large enough, we have $u'(a) \to 0$, so the first term in the max dominates, and we have the equality holds strictly. Therefore, we have the threshold structure:

$$u'(c) = \begin{cases} u'(a) & \text{if } a \leq \bar{a}(y) \\ \beta R \mathbb{E}[(u' \circ \sigma)(R(a - c) + y', y')] & \text{if } a > \bar{a}(y) \end{cases}$$

And equivalently, we have:

$$c = \begin{cases} a & \text{if } a \leq \bar{a}(y) \\ (u')^{-1}\{\beta R \mathbb{E}[(u' \circ \sigma)(R(a - c) + y', y')]\} & \text{if } a > \bar{a}(y) \end{cases} \tag{3}$$

Consider an **exogenous** grid of saving values $0 = s_0 < ... < s_{N-1}$. We can then build a grid and c grid as follows:

1. $a_0 = c_0 = 0$ since this is the only feasible choice
2. $\forall i > 0$: consider $c_i = (u')^{-1}\{\beta R \mathbb{E}[(u' \circ \sigma)(Rs_i + y', y')]\}$ where $s_i = a_i - c_i > 0$. Given Equation 3, we then have the lower part holds, that is:

$$c_i = (u')^{-1}\{\beta R \mathbb{E}[(u' \circ \sigma)(Rs_i + y', y')]\}$$

3. In this case, now we have $a_i = s_i + c_i > \bar{a}(y)$, so the pair (a_i, c_i) satisfies:

$$c_i = \beta R \mathbb{E}[(u' \circ \sigma)(R(a_i - c_i) + y', y')]$$

4. This is precise the definition of $K\sigma(a_i, y)$.

Let's look at the code:

```
@jax.jit
def u_prime(c, gammma):
    return c ** (-gammma)


@jax.jit
def u_prime_inv(u, gammma):
    return u ** (-1 / gammma)
```

It seems that we need to take it slow. Let's look at the Equation 2, the input of the function $K\sigma$ is (a, y), and the output is c. Also note that we need to iterate the asset grid a, so the input shape and the output shape are `(a_size,y_size),(a_size,y_size)`. And given the shape of a being `s_size`, we need to make sure the input and output shape are `(s_size,y_size),(s_size,y_size)`.

```
model = create_model()
params, sizes, arrays = model
s_grid, y_grid, P = arrays
beta, R, gamma = params
s_size, y_size = sizes
```

Let's look at the broadcast:

```
a_grid = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
print(a_grid.shape)

y_hat = jnp.reshape(y_grid, (1, 1, y_size))
```

```
s = jnp.reshape(s_grid, (s_size, 1, 1))
a_next = R * s + y_hat
print(a_next.shape)
```

```
(1000, 25)
(1000, 1, 25)
```

`vmap` vs. `jnp.vectorize`

Let's take a look at the difference between `vmap` and `jnp.vectorize`.

From quantecon's website, it's advised to use `jnp.vectorize`:

```
def _sigma_vec(a_in, sigma_in):

    def sigma(a, y_idx):
        return jnp.interp(a, a_in[:, y_idx], sigma_in[:, y_idx])

    return jnp.vectorize(sigma)


a_in = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
sigma_in = jnp.copy(a_grid)

sigma_vec = _sigma_vec(a_in, sigma_in)

y_indices = jnp.arange(y_size)[jnp.newaxis, jnp.newaxis, :]

_ = sigma_vec(a_in, y_indices)

start_time = time.time()
ans_vec = sigma_vec(a_next, y_indices).block_until_ready()
end_time = time.time()
print(f"Time used: {end_time - start_time} seconds")
print(a_next.shape, y_indices.shape, ans_vec.shape)
```

```
Time used: 0.021235227584838867 seconds
(1000, 1, 25) (1, 1, 25) (1000, 1, 25)
```

This is somewhat complex, but now let's look at the `vmap` version:

```
def _sigma_col(a_col, y_idx, a_in, sigma_in):
    return jnp.interp(a_col, a_in[:, y_idx], sigma_in[:, y_idx])


sigma_vmap = jax.vmap(_sigma_col, in_axes=(1, 0, None, None))
a_next = a_next[:, 0, :]
y_indices = jnp.arange(y_size)
_ = sigma_vmap(a_next, y_indices, a_in, sigma_in)
start_time = time.time()
ans_vmap = sigma_vmap(a_next, y_indices, a_in, sigma_in).block_until_ready()
end_time = time.time()
print(f"Time used: {end_time - start_time} seconds")
print(a_next.shape, y_indices.shape, ans_vmap.shape)
```

```
Time used: 0.0022039413452148438 seconds
(1000, 25) (25,) (25, 1000)
```

Note that we need to transpose the answer from vmap:

```
ans_vmap = ans_vmap.T
print(jnp.allclose(ans_vmap, ans_vec[:, 0, :]))
```

```
True
```

But there's another way to do this:

```
sigma_vmap_2 = jax.vmap(
    _sigma_col,
    in_axes=(1, 0, None, None),
    out_axes=1,
)

_ = sigma_vmap_2(a_next, y_indices, a_in, sigma_in)
start_time = time.time()
ans_vmap_2 = sigma_vmap_2(a_next, y_indices, a_in, sigma_in).block_until_ready()
end_time = time.time()
print(f"Time used: {end_time - start_time} seconds")
print(a_next.shape, y_indices.shape, ans_vmap_2.shape)
print(jnp.allclose(ans_vmap_2, ans_vec[:, 0, :]))
```

```
Time used: 0.0021219253540039062 seconds
(1000, 25) (25,) (1000, 25)
True
```

So a primary observation is that proper `vmap` is faster than `jnp.vectorize`. But there's something interesting about the `vmap` itself.

Let's look at the function `_sigma_col`:

- we try to fix one `y_idx`
- then use `a_in[:,y_idx]` and `sigma_in[:,y_idx]` to interpolate the value of `a_col`
- the input is `a_col` which should be a column vector, and in this case `len(a_col) = s_size = 200`.
- return is a column vector, and in this case `len(return) = s_size = 200`.

Now look at the parameter of `vmap`:

```
sigma_vmap = jax.vmap(_sigma_col, in_axes=(1, 0, None, None))
```

This means:

Parameter	Actual Parameter	`in_axes`	Meaning
`a_col`	`a_next`	1	Map along axis=1 (over y dimension)
`y_idx`	`y_indices`	0	Map along axis=0 (over y_indices)
`a_in`	`a_in`	None	No mapping, shared by all calls (broadcasted)
`sigma_in`	`sigma_in`	None	No mapping, shared by all calls (broadcasted)

So this is basically

```
for y in range(y_size):
    out[y] = _sigma_col(
        a_next[:, y],
        y_indices[y],
```

```
        a_in,
        sigma_in,
    )
```

Then how about the `out_axes=1`? The default value is `out_axes=0`, that is to put the mapped y-dimensional data into `axis=0`. , so the output is then `(y_size, a_size)`.

If we set `out_axes=1`, then the output is then `(a_size, y_size)`, which is what we want.

Back to EGM

Now let's finish the K function:

```
@jax.jit
def K(a_in, sigma_in, constants, sizes, arrays):
    beta, R, gamma = constants
    s_grid, y_grid, P = arrays
    s_size, y_size = s_grid.shape[0], y_grid.shape[0]

    a_next = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
    y_indices = jnp.arange(y_size)
    sigma_next = sigma_vmap_2(a_next, y_indices, a_in, sigma_in)
    up = u_prime(sigma_next, gamma)
    EV = up @ P.T
    c = u_prime_inv(beta * R * EV, gamma)

    # sigma_out = c.at[0, :].set(0)
    sigma_out = jnp.minimum(c, a_next)
    a_out = s_grid[:, jnp.newaxis] + sigma_out
    return a_out, sigma_out
```

> **Caution**
>
> Note that we have this:
>
> ```
> sigma_out = c.at[0, :].set(0)
> ```
>
> This is a forced setting: When `s=0`, `c=0`. Therefore, `a_out=0`. This ensures that the intrinsic mesh always starts from 0, the interpolation always has a left-end anchor point, and prevents potential explosion.
> But we can do better, recall the economic interpretation of the borrowing constraint: when $s = 0$, we need to let $c = a$, therefore, we just need to take the minimum of c and a. That is,

```
sigma_out = jnp.minimum(c, a_next)
```

Now we write the main loop:

```
@jax.jit(static_argnums=(3, 4))
def EGM_loop(
    model,
    max_iter=10000,
    tol=1e-8,
    verbose=True,
    print_skip=10,
):
    constants, sizes, arrays = model
    beta, R, gamma = constants
    s_size, y_size = sizes
    s_grid, y_grid, P = arrays
    a_init = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
    sigma_init = jnp.ones_like(a_init)

    def body_fun(k_a_sigma_err):
        k, a, sigma, err = k_a_sigma_err
        a_new, sigma_new = K(a, sigma, constants, sizes, arrays)
        err = jnp.max(jnp.abs(sigma_new - sigma))
        jax.lax.cond(
            verbose and k % print_skip == 0,
            lambda _: jax.debug.print(
                "Concluded loop {k} with error {err}.", k=k, err=err
            ),
            lambda _: None,
            operand=None,
        )
        return k + 1, a_new, sigma_new, err

    def cond_fun(k_a_sigma_err):
        k, a, sigma, err = k_a_sigma_err
        return jnp.logical_and(k < max_iter, err > tol)

    k, a, sigma, err = jax.lax.while_loop(
        cond_fun,
        body_fun,
        (0, a_init, sigma_init, jnp.inf),
    )
```

```
    return a, sigma
```

Let's test the EGM now:

```
model = create_model()
a, sigma_egm = EGM_loop(model)
sigma_egm.block_until_ready()
start_time = time.time()
a, sigma_egm = EGM_loop(model)
sigma_egm.block_until_ready()
egm_time = time.time() - start_time
print(f"Time used: {egm_time} seconds")
```

```
Concluded loop 0 with error 0.34644508861857604.
Concluded loop 10 with error 0.022023088926695022.
Concluded loop 20 with error 0.01265346366425768.
Concluded loop 30 with error 0.014019866624134458.
Concluded loop 40 with error 0.015533822625478422.
Concluded loop 50 with error 0.017211265401372522.
Concluded loop 60 with error 0.019069766059998106.
Concluded loop 70 with error 0.020833280750971905.
Concluded loop 80 with error 0.014491303171589642.
Concluded loop 90 with error 0.0028411057982036247.
Concluded loop 100 with error 0.00024755093628137104.
Concluded loop 110 with error 5.958195049338144e-06.
Concluded loop 120 with error 6.533547036724485e-09.
Concluded loop 0 with error 0.34644508861857604.
Concluded loop 10 with error 0.022023088926695022.
Concluded loop 20 with error 0.01265346366425768.
Concluded loop 30 with error 0.014019866624134458.
Concluded loop 40 with error 0.015533822625478422.
Concluded loop 50 with error 0.017211265401372522.
Concluded loop 60 with error 0.019069766059998106.
Concluded loop 70 with error 0.020833280750971905.
Concluded loop 80 with error 0.014491303171589642.
Concluded loop 90 with error 0.0028411057982036247.
Concluded loop 100 with error 0.00024755093628137104.
Concluded loop 110 with error 5.958195049338144e-06.
Concluded loop 120 with error 6.533547036724485e-09.
Time used: 0.0611269474029541 seconds
```

Let's first look at the result:

```
# for egm, let's convert savings to assets
a_grid = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
for i in (0, jnp.floor(y_size / 2).astype(int), y_size - 1):
    plt.plot(a_grid[:, i], sigma_egm[:, i], label=f"y={y_grid[i]:.3f}")
plt.plot(a_grid[:, 0], a_grid[:, 0], "k--", label=r"$c = a$")
plt.xlabel(r"$a$")
plt.ylabel(r"$\sigma(a, y)$")
plt.ylim(0.6, 2.5)
plt.title("EGM Policy Function")
plt.legend()
plt.tight_layout()
plt.show()
```

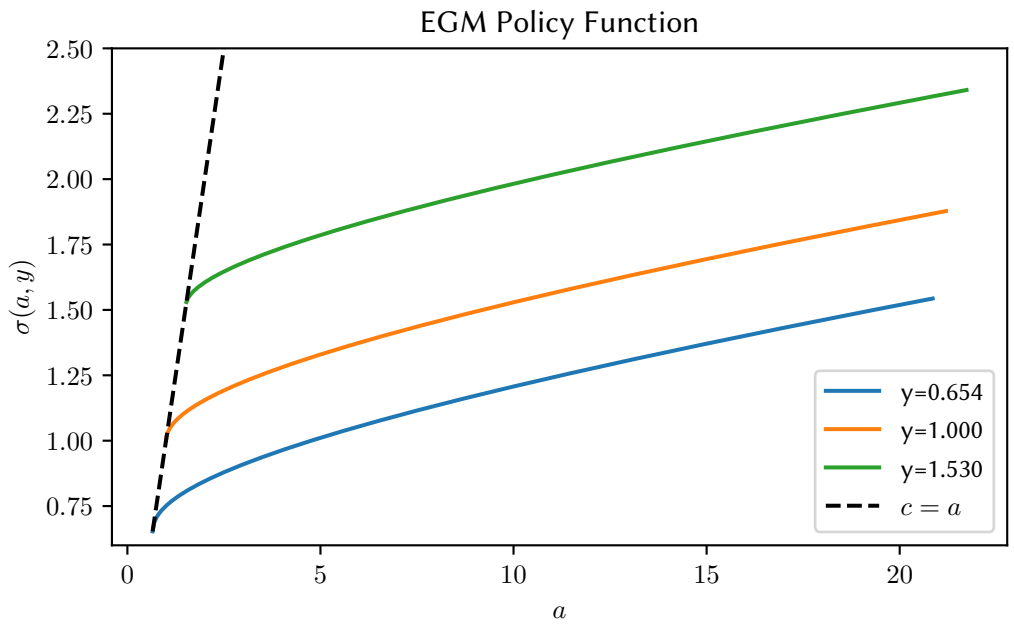

Figure 1: EGM Policy Function

This is pretty good! And Let's see VFI version:

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

```
def _B(v, constants, sizes, arrays, i, j, ip):
    beta, R, gamma = constants
    a_grid, y_grid, P = arrays

    a, y, ap = a_grid[i], y_grid[j], a_grid[ip]
    c = a + (y - ap) / R
    EV = jnp.sum(v[ip, :] * P[j, :])
    return u(c, gamma) + beta * EV


_B_1 = jax.vmap(_B, in_axes=(None, None, None, None, None, None, 0))
_B_2 = jax.vmap(_B_1, in_axes=(None, None, None, None, None, 0, None))
B_vmap = jax.vmap(_B_2, in_axes=(None, None, None, None, 0, None, None))


@jax.jit
def B(v, constants, sizes, arrays):
    a_grid, y_grid, P = arrays
    a_size, y_size = a_grid.shape[0], y_grid.shape[0]
    a_indices, y_indices = jnp.arange(a_size), jnp.arange(y_size)
    return B_vmap(v, constants, sizes, arrays, a_indices, y_indices, a_indices)


@jax.jit
def get_greedy(v, constants, sizes, arrays):
    return jnp.argmax(B(v, constants, sizes, arrays), axis=-1)


@jax.jit
def T(v, constants, sizes, arrays):
    return jnp.max(B(v, constants, sizes, arrays), axis=-1)


@jax.jit
def VFI(model, max_iter=10000, tol=1e-8, verbose=True, print_skip=100):
    constants, sizes, arrays = model
    beta, R, gamma = constants
    a_grid, y_grid, P = arrays
    a_size, y_size = a_grid.shape[0], y_grid.shape[0]

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

```
        vp = T(v, constants, sizes, arrays)
        err = jnp.max(jnp.abs(vp - v))
        jax.lax.cond(
            verbose and k % print_skip == 0,
            lambda _: jax.debug.print(
                "Concluded loop {k} with error {err}.", k=k, err=err
            ),
            lambda _: None,
            operand=None,
        )
        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)

    k, v, err = jax.lax.while_loop(
        cond_fun,
        body_fun,
        (0, jnp.zeros((a_size, y_size)), jnp.inf),
    )
    return v, get_greedy(v, constants, sizes, arrays)
```

Let's test the VFI function:

```
model = create_model()
v_vfi, sigma_vfi = VFI(model)

sigma_vfi.block_until_ready()
start_time = time.time()
v_vfi, sigma_vfi = VFI(model)

sigma_vfi.block_until_ready()
vfi_time = time.time() - start_time
print(f"Time used: {vfi_time} seconds")
print(f"Speedup: {vfi_time / egm_time}")
```

```
Concluded loop 0 with error 1.5453942470108515.
Concluded loop 100 with error 0.055460774393488066.
Concluded loop 200 with error 0.002422419624252825.
Concluded loop 300 with error 0.00011176280347768852.
```

```
Concluded loop 400 with error 5.257851476869746e-06.
Concluded loop 500 with error 2.49075846170399e-07.
Concluded loop 600 with error 1.1828326762497454e-08.
Concluded loop 0 with error 1.5453942470108515.
Concluded loop 100 with error 0.055460774393488066.
Concluded loop 200 with error 0.002422419624252825.
Concluded loop 300 with error 0.00011176280347768852.
Concluded loop 400 with error 5.257851476869746e-06.
Concluded loop 500 with error 2.49075846170399e-07.
Concluded loop 600 with error 1.1828326762497454e-08.
Time used: 1.610954999923706 seconds
Speedup: 26.354252393860794
```

> **Note**
>
> We can see that the VFI is much faster than the EGM. The speedup increases as the grid size increases and decreases as $\beta R \to 1$.

Note that we actually have the asset policy here in VFI, so to compare it, let's convert it to consumption policy:

```
constants, sizes, arrays = model
s_grid, y_grid, P = arrays

a_vfi = s_grid
ap_vfi = a_vfi[sigma_vfi]
c_vfi = a_vfi[:, jnp.newaxis] + (y_grid[jnp.newaxis, :] - ap_vfi) / R

a_egm = R * s_grid[:, jnp.newaxis] + y_grid[jnp.newaxis, :]
s_egm = s_grid
s_vfi = a_vfi[:, jnp.newaxis] - c_vfi


for i in (0, jnp.floor(y_size / 2).astype(int), y_size - 1):
    plt.plot(a_vfi + y_grid[i], c_vfi[:, i], label=f"y={y_grid[i]:.3f}, VFI")
    plt.plot(a_egm[:, i], sigma_egm[:, i], label=f"y={y_grid[i]:.3f}, EGM")
plt.plot(a_vfi, a_vfi, "k--", label=r"$c = a$")
plt.ylim(0.6, 2.5)
plt.xlabel(r"$a$ for EGM, $a + y$ for VFI")
plt.ylabel(r"$\sigma(a, y)$")
plt.legend()
plt.show()
```

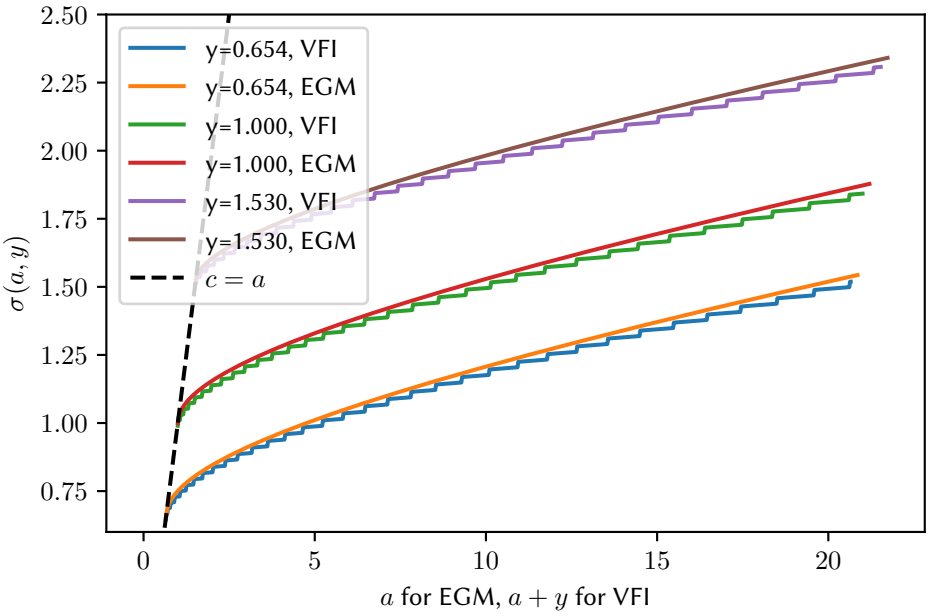

Figure 2: Policy Function Comparison

We can see that EGM gives a better policy function than VFI. The results are different because the timing used for VFI is actually differnt from the timing used for EGM.

So to align these, interpret the a in VFI version as $a + y$, that is, cash on hand.

Takeaways

- EGM is fast, harder to implement, but gives a better policy function.
- VFI is slower, easier to implement, but gives a worse policy function.
- In total, they're both pretty fast
- EGM < HPI < VFI especially when the grid size is large.

Make sure you learn `jax.vmap` as in the doc.