The Krusell-Smith Method

Lecturer: Bo Li TA: Chen Gao

2026-03-27

Table of contents

1 Introduction

- Agents face both **idiosyncratic** and **aggregate risk**.
- Markets are **incomplete**:
  - The only asset available is **physical capital**.
  - Negative capital positions are **ruled out by a borrowing constraint**.
- For comparison:
  - **Bewley/Huggett** models:
    - No aggregate risk.
    - Only asset is a one-period bond in zero net supply.
  - **Aiyagari model**:
    - No aggregate risk.
    - Only asset is capital.
    - Aggregate stock of capital always remains at steady-state.

- **Key challenge**:
  How do we solve a model featuring both idiosyncratic and aggregate risk, where the **distribution of wealth** is itself an **endogenous state variable**?

2 Model

2.1 Households

- **Continuum of consumers:** $j \in [0, 1]$
- **Preferences:**

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t \frac{c_{jt}^{1-\sigma} - 1}{1 - \sigma}$$

- **Labor income:** each employed household supplies $\tilde{l}$ efficiency units of labor, so income is $w_t \tilde{l} \varepsilon_{jt}$ with $\varepsilon_{jt} \in \{0, 1\}$.
- **Budget constraint:**

$$c_{jt} + k_{j,t+1} = w_t \tilde{l} \varepsilon_{jt} + (1 + r_t - \delta) k_{jt}$$

- **Borrowing constraint:**

$$k_{j,t+1} \geq \underline{k} = 0$$

> **Note**
>
> If $\min\{w_t \tilde{l} \varepsilon_{jt}\} = 0$, then $\underline{k} = 0$ is a natural borrowing limit: there exists a path along which an indebted household receives zero labor income and therefore can never repay any positive amount of debt.

2.2 Firms

- **Competitive firms** produce output:

$$Y_t = e^{z_t} K_t^{\alpha} L_t^{1-\alpha}$$

- **First Order Conditions (FOC) for factor prices:**
  - Wage:

$$w_t = (1 - \alpha) e^{z_t} K_t^{\alpha} L_t^{-\alpha}$$

– Rental rate of capital before depreciation:

$$r_t = \alpha e^{z_t} K_t^{\alpha-1} L_t^{1-\alpha}$$

2.3 Shocks

- **Aggregate productivity:** $z_t \in \{z^g, z^b\}$ follows a *two-state Markov chain* with transition probabilities
$$\Pr(z_{t+1} = z^{s'} \mid z_t = z^s) = \pi_{ss'}, \qquad s, s' \in \{g, b\}.$$
- **Idiosyncratic employment:** $\varepsilon_{jt} \in \{0, 1\}$.
- **Correlated transitions:** the employment transition probabilities depend on the aggregate state transition. Let u_g and u_b denote the unemployment rates in good and bad times.
- **Joint transition matrix:** rows correspond to current states
$$(z, \varepsilon) \in \{(g, 1), (g, 0), (b, 1), (b, 0)\},$$
and columns correspond to next-period states (z', ε') in the same order.

- **How is the joint matrix constructed?**
- Each row must sum to one.
- Marginalizing over employment must recover the aggregate transition:
$$\sum_{\varepsilon' \in \{0,1\}} \Pr(z', \varepsilon' \mid z, \varepsilon) = \Pr(z' \mid z) = \pi_{zz'}.$$
- The stationary employment shares within each aggregate state must match the target unemployment rates:
$$\Pr(\varepsilon = 0 \mid z = g) = u_g, \qquad \Pr(\varepsilon = 0 \mid z = b) = u_b.$$
- In applications, the remaining free parameters are chosen to match job-finding and job-separation durations, so bad times feature both higher unemployment and more persistent non-employment.

- **Aggregate employment:** depends only on the aggregate state:

$$L(z^s) = \tilde{l}(1 - u_s), \qquad s \in \{g, b\}.$$

The aggregate state is $s_t = (z_t, \mu_t)$, where μ_t is the distribution of households over (k, ε) and aggregate capital is

$$K_t = \int k \, d\mu_t(k, \varepsilon).$$

2.4 Timing And Market Clearing

Within period t:

1. The aggregate state (z_t, μ_t) is observed.
2. Firms rent capital K_t and labor L_t, and competitive prices (r_t, w_t) are determined.
3. Households receive income, consume c_t, and choose savings k_{t+1}.
4. Next period shocks are realized, inducing the new distribution μ_{t+1}.

- **Capital market clearing:**
$$K_t = \int k \, d\mu_t(k, \varepsilon).$$
- **Aggregation of savings:**
$$K_{t+1} = \int k'(k, \varepsilon; z_t, \mu_t) \, d\mu_t(k, \varepsilon).$$
- **Goods market clearing:**
$$C_t + K_{t+1} = Y_t + (1 - \delta)K_t.$$

2.5 Recursive Formulation

Let the household's value function be

$$v(k, \varepsilon; z, \mu) = \max_{c, k'} \left\{ \frac{c^{1-\sigma} - 1}{1 - \sigma} + \beta \, \mathbb{E}\left[v\left(k', \varepsilon'; z', \mu'\right) \mid z, \varepsilon \right] \right\}$$

subject to

$$c + k' = \left[1 + r(z, \mu) - \delta\right] k + w(z, \mu) \tilde{l} \varepsilon$$

$$\mu' = H(\mu, z, z')$$

$$k' \geq 0$$

Explanations:

- **States:**
  - k is current asset holding
  - ε is the idiosyncratic employment status
  - z is the aggregate productivity shock
  - μ is the cross-sectional distribution of agents over (k, ε)
- **Choice variables:**
  - c is consumption
  - k' is next period's asset
- **Prices:**
  - $r(z, \mu)$ is the rental rate of capital before depreciation
  - $w(z, \mu)$ is the equilibrium wage
  - Thus the gross return on one unit of capital is $1 + r(z, \mu) - \delta$

2.6 Equilibrium

A **recursive competitive equilibrium** is a collection of functions:

- $k'(k, \varepsilon; z, \mu)$ (the household policy)
- $r(z, \mu)$ (interest rate)
- $w(z, \mu)$ (wage)
- $H(\mu, z, z')$ (law of motion for the distribution)

such that:

1. **Household optimality:** Taking $r(z, \mu)$, $w(z, \mu)$, and $H(\mu, z, z')$ as given, the policy $k'(k, \varepsilon; z, \mu)$ solves the household problem. Equivalently, it satisfies

$$c(k,\varepsilon;z,\mu)^{-\sigma} \geq \beta \mathbb{E}\left[(1+r(z',\mu')-\delta)\, c(k',\varepsilon';z',\mu')^{-\sigma} \mid \varepsilon, z\right]$$

with equality if $k'(k,\varepsilon;z,\mu) > \underline{k}$, where

$$c(k,\varepsilon;z,\mu) = w(z,\mu)\tilde{l}\varepsilon + [1+r(z,\mu)-\delta]\, k - k'(k,\varepsilon;z,\mu),$$

and $\mu' = H(\mu, z, z')$.

---

2. **Price consistency:** Let

$$K(\mu) = \int k\, d\mu(k,\varepsilon), \qquad L(z) = \tilde{l}(1-u_z).$$

Then prices satisfy

$$r(z,\mu) = \alpha e^z K(\mu)^{\alpha-1} L(z)^{1-\alpha}$$

$$w(z,\mu) = (1-\alpha) e^z K(\mu)^{\alpha} L(z)^{-\alpha}$$

3. **Law of motion for distribution:** For every realized aggregate transition $z \to z'$ and every measurable set Δ_k,

$$H(\mu,z,z')(\varepsilon',\Delta_k) = \sum_{\varepsilon\in\{0,1\}} \int 1\{k'(k,\varepsilon;z,\mu) \in \Delta_k\} \Pr(\varepsilon' \mid \varepsilon, z, z')\, d\mu(k,\varepsilon)$$

3 Computation

3.1 Computational Difficulty

- The endogenous wealth distribution, μ, is a state variable.
  - **Why?** Since consumers have different propensities to save, we need to know the distribution to forecast next period's capital stock K'. This, in turn, determines the return on capital r'. Hence, μ is part of the value function.
- The problem is in fact more severe:

  - To forecast r'' (and further into the future), we also need to forecast K'', since consumption/savings decisions are forward looking. Consequently, μ' (the next period distribution) is part of the value function in the next period.

- The challenge:
  - μ is an infinite-dimensional object (think: moment generating function).
  - Furthermore, the transition equation $H(\cdot)$ (which governs the evolution of μ) is also infinite dimensional.

3.2 From Exact To Approximate Aggregation

- **Exact recursive problem:** households should solve

$$v(k, \varepsilon; z, \mu),$$

  because future prices depend on future aggregates, and future aggregates depend on the entire distribution μ.

- **Krusell-Smith idea:** replace the true infinite-dimensional forecasting problem with a perceived law of motion for aggregate capital:

$$K' = G(K, z).$$

- If this perceived law is accurate enough, households can solve the much simpler problem

$$v(k, \varepsilon; z, K)$$

  instead of carrying the whole distribution as a state variable.

- The approximation is not exact. It is a numerical claim that, for forecasting K', the additional information in μ beyond current K is small.

3.3 Approximate Aggregation KS Solution

In an approximate equilibrium, agents will use a **forecasting rule** for aggregate capital next period:

$$z = z^g : \quad \log K' = a_0 + a_1 \log K$$

$$z = z^b : \quad \log K' = b_0 + b_1 \log K$$

- This rule is **recursive**, so it also helps forecast K'', K''', etc.

- **Interpretation:** Either as bounded rationality, or as the best linear forecast for capital next period.

With this, the model **simplifies** to:

$$v(k, \varepsilon; z, K) = \max_{c,k'} \left\{ \frac{c^{1-\sigma} - 1}{1-\sigma} + \beta \, \mathbb{E}\left[v\left(k', \varepsilon'; z', K'\right) \mid z, \varepsilon \right] \right\}$$

subject to:

$$c + k' = \left[1 + r(z, K) - \delta\right] k + w(z, K) \tilde{l} \varepsilon$$

$$\log K' = \begin{cases} a_0 + a_1 \log K & \text{if } z = z^g \\ b_0 + b_1 \log K & \text{if } z = z^b \end{cases}$$

$$k' \geq 0$$

Idea:
Find coefficients $\{a_0, a_1, b_0, b_1\}$ as a fixed point to this problem:

1. Start with initial guesses for the coefficients.
2. Solve the value function with these guesses.
3. Update the guesses based on the solution.
4. Repeat until convergence.

3.4 Why This Reduction Can Work

- The wealth distribution matters for individual savings decisions.
- KS makes a narrower claim: for forecasting *aggregate* capital one period ahead, most of the relevant information in μ is often summarized by current aggregate capital K and the aggregate shock z.

- So the hard mapping

$$\mu_t \mapsto K_{t+1}$$

  is approximated by the simpler mapping

$$(K_t, z_t) \mapsto K_{t+1}.$$

- This is why the forecasting regressions and their errors are central to the method.

3.5 Policy Function Iteration

1. **Start with an initial guess:** $k'(k, \varepsilon; z, K)$
2. **Compute next period's capital for all possible future states:** For each possible realization of (ε', z'), calculate $k'' = k'(k', \varepsilon'; z', K')$
3. **Update consumption using the budget constraint:**

$$c'(k', \varepsilon'; z', K') = [1 + r(z', K') - \delta]\, k' + w(z', K')\tilde{l}\varepsilon' - k''$$

4. **Calculate marginal utility:**

$$u_c\left(c'(k', \varepsilon'; z', K')\right) = [c'(k', \varepsilon'; z', K')]^{-\sigma}$$

---

5. **Update today's consumption using the Euler equation:**

$$[c(k, \varepsilon; z, K)]^{-\sigma} = \beta\, \mathbb{E}\left[(1 + r(z', K') - \delta)\, [c'(k', \varepsilon'; z', K')]^{-\sigma}\right]$$

   Equivalently,

$$c(k, \varepsilon; z, K) = \left\{\beta\, \mathbb{E}\left[(1 + r(z', K') - \delta)\, [c'(k', \varepsilon'; z', K')]^{-\sigma}\right]\right\}^{-1/\sigma}$$

6. **Update today's capital:** Use the budget constraint to solve for today's k' and improve on your initial guess.

> **Note**
>
> Policy function iteration is typically faster than value function iteration for this problem.

3.6 Full Algorithm

1. **Guess forecasting coefficients:** start from (a_0, a_1, b_0, b_1).
2. **Solve the household problem:** taking the perceived law of motion for K' as given, compute $k'(k, \varepsilon; z, K)$.
3. **Simulate the economy:** generate long time series for many agents and recover the implied aggregate capital path $\{K_t\}$.
4. **Update the forecasting rule:** run separate OLS regressions in good and bad states:

$$\log K_{t+1} = a_0 + a_1 \log K_t \quad \text{if } z_t = z^g,$$

$$\log K_{t+1} = b_0 + b_1 \log K_t \quad \text{if } z_t = z^b.$$

5. **Damp the coefficient update if needed:** for example,

$$\theta^{new} = \lambda \hat{\theta} + (1 - \lambda)\theta^{old}, \qquad \lambda \in (0, 1].$$

6. **Repeat until convergence:** coefficients, forecast errors, and key simulated moments should all stabilize.

3.7 Accuracy Checks

- A high regression R^2 is useful, but it is not sufficient by itself.
- Also inspect the forecast error along the simulation path:

$$\log K_{t+1} - \widehat{\log K}_{t+1}.$$

- Compare simulated aggregates across alternative solution methods when possible; this is the spirit of Den Haan style accuracy tests.
- Check whether the implied business-cycle and distributional moments are stable across iterations.

3.8 When KS May Perform Poorly

- Approximate aggregation is an empirical regularity, not a general theorem.
- It may perform poorly when the cross-sectional distribution moves in ways that current aggregate capital does not summarize well.
- Common risk cases:
  - strong nonlinear policy experiments
  - many agents close to occasionally binding constraints
  - environments with extra aggregate prices, wedges, or state variables
  - distributions with fat tails or unusually strong precautionary saving motives

3.9 Conclusion

- Major technical advance.
- Approximate aggregation seems to work well in many applications (though not all—you need to check).
- The central approximation is to replace the infinite-dimensional state μ with a forecasting rule for K'.
- The method is only convincing after passing numerical accuracy checks.
- Baseline models often imply MPCs that are too low relative to empirical studies of temporary tax cuts, which find average MPCs around 0.25–0.3.
- Some further reading:
  - See (Young 2010) for algorithmic improvements using non-stochastic simulation.
  - See (Haan, Judd, and Juillard 2010) and (Maliar, Maliar, and Valli 2010) for discussion and code comparisons.
  - See (Werning 2015) and (Chipeniuk, Katz, and Walker 2016) for theoretical analysis of approximate aggregation.

4 Code

4.1 Preliminaries

We begin with a few imports and two tiny utility functions.

```
import time
import jax
import jax.numpy as jnp
import numpy as np
```

```
import matplotlib.pyplot as plt
from jax import lax
from typing import NamedTuple

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


def print_summary(title, rows):
    print(title)
    print("-" * len(title))
    for label, value in rows:
        print(f"{label:<24} {value}")


def block_tree(tree):
    """Force all JAX arrays in a pytree to finish computation."""
    for leaf in jax.tree_util.tree_leaves(tree):
        if hasattr(leaf, "block_until_ready"):
            leaf.block_until_ready()
    return tree


def time_call(fn, *args, warmup=False, **kwargs):
    """Time a call, optionally using one untimed warmup pass first."""
    if warmup:
        block_tree(fn(*args, **kwargs))
    start = time.perf_counter()
    out = fn(*args, **kwargs)
    block_tree(out)
    elapsed = time.perf_counter() - start
    return out, elapsed
```

> **Warning**
>
> As in the RBC lecture, we enable 64-bit precision because iterative dynamic-programming problems can become numerically fragile under JAX's default 32-bit mode.

4.2 A Small Amount Of Structure

Compared with standard incomplete markets models, this model has more moving parts. We still want the code to read line by line, but it helps to bundle related objects into `NamedTuples` so that JAX

can pass them around cleanly.

```
class ModelParams(NamedTuple):
    beta: float
    sigma: float
    alpha: float
    delta: float
    l_bar: float
    benefit_rate: float
    z_vals: jnp.ndarray
    ur_vals: jnp.ndarray
    er_vals: jnp.ndarray


class KSGrids(NamedTuple):
    k_grid: jnp.ndarray
    km_grid: jnp.ndarray
    k_min: float
    k_max: float
    km_min: float
    km_max: float


class ShockProcess(NamedTuple):
    P: jnp.ndarray
    P_joint: jnp.ndarray
    P_agg: jnp.ndarray
    P_cond_unemp: jnp.ndarray


class SimulationConfig(NamedTuple):
    T: int
    ndiscard: int
    N: int


class SolverConfig(NamedTuple):
    update_k: float
    tol_policy: float
    max_iter_policy: int
    future_states: jnp.ndarray
```

```
class HouseholdObjects(NamedTuple):
    labor_income: jnp.ndarray
    resources: jnp.ndarray


class KSOuterConfig(NamedTuple):
    tol_B: float
    max_iter_B: int
    update_B: float


class RegressionResults(NamedTuple):
    B_hat: np.ndarray
    R2_bad: float
    R2_good: float
    n_bad: int
    n_good: int
```

4.3 Parameters And Steady State

We first set the primitive parameters and then package them into the objects that later functions will use.

```
# Preferences and technology
beta = 0.99
sigma = 1.0
alpha = 0.36
delta = 0.025

# Aggregate productivity: bad and good
delta_a = 0.01
z_b = 1.0 - delta_a
z_g = 1.0 + delta_a

# Unemployment rates and labour endowment
ur_b = 0.10
ur_g = 0.04
er_b = 1.0 - ur_b
er_g = 1.0 - ur_g
l_bar = 1.0 / 0.9
benefit_rate = 0.0
```

```
params = ModelParams(
    beta=beta,
    sigma=sigma,
    alpha=alpha,
    delta=delta,
    l_bar=l_bar,
    benefit_rate=benefit_rate,
    z_vals=jnp.array([z_b, z_g]),
    ur_vals=jnp.array([ur_b, ur_g]),
    er_vals=jnp.array([er_b, er_g]),
)

simulation = SimulationConfig(
    T=1100,
    ndiscard=100,
    N=10_000,
)
```

The deterministic steady-state capital level is useful as a reference point for the grids and for the initial cross-sectional distribution.

```
kss = ((1.0 / params.beta - (1.0 - params.delta)) / params.alpha) ** (
    1.0 / (params.alpha - 1.0)
)

print_summary(
    "Model Setup Summary",
    [
        ("Steady-state capital", f"{kss:.4f}"),
        ("Benefit rate", f"{params.benefit_rate:.2f}"),
        ("Simulation length", f"{simulation.T}"),
        ("Number of agents", f"{simulation.N}"),
    ],
)
```

```
Model Setup Summary
--------------------
Steady-state capital     37.9893
```

```
Benefit rate              0.00
Simulation length         1100
Number of agents          10000
```

4.4 Build The Grids

We use a dense grid near the borrowing constraint for individual capital, because that is where the savings policy is most curved. Aggregate capital moves over a much narrower range, so a short uniform grid is enough.

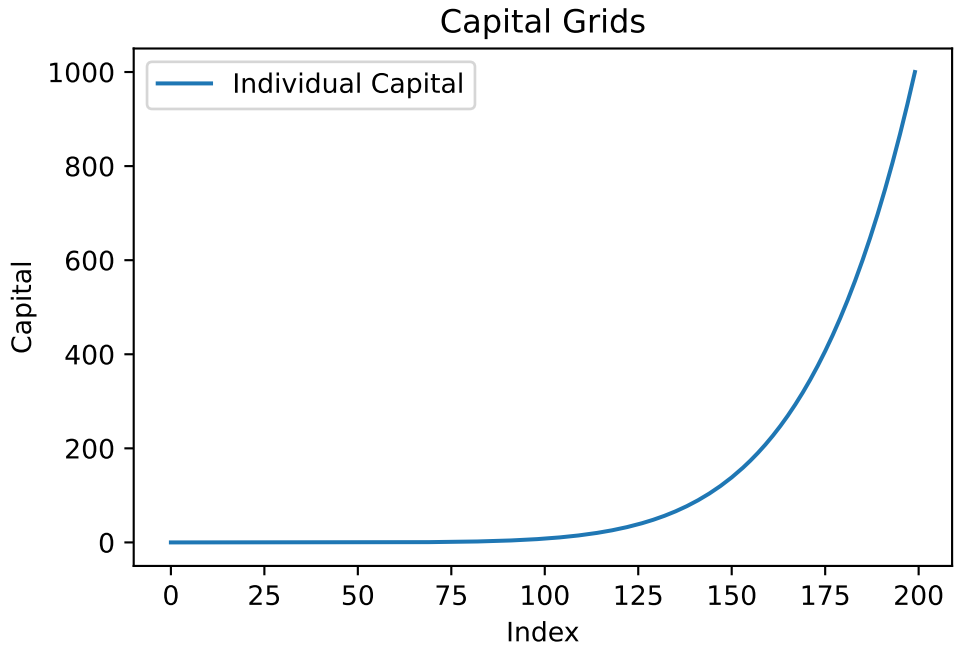
```
ngridk = 200
k_min = 0.0
k_max = 1000.0

x = jnp.linspace(0.0, 0.5, ngridk)
y = x**7 / jnp.max(x**7)
k_grid = k_min + (k_max - k_min) * y

ngridkm = 4
km_min = 30.0
km_max = 50.0
km_grid = jnp.linspace(km_min, km_max, ngridkm)

grids = KSGrids(
    k_grid=k_grid,
    km_grid=km_grid,
    k_min=k_min,
    k_max=k_max,
    km_min=km_min,
    km_max=km_max,
)
```

```
print_summary(
    "Grid Summary",
    [
        ("Individual grid points", f"{grids.k_grid.shape[0]}"),
        ("Aggregate grid points", f"{grids.km_grid.shape[0]}"),
        ("k range", f"[{float(grids.k_min):.1f}, {float(grids.k_max):.1f}]"),
        ("K range", f"[{float(grids.km_min):.1f}, {float(grids.km_max):.1f}]"),
```

```
    ],
)
```

```
Grid Summary
------------
Individual grid points   200
Aggregate grid points    4
k range                  [0.0, 1000.0]
K range                  [30.0, 50.0]
```

```
plt.plot(jnp.arange(grids.k_grid.shape[0]), grids.k_grid, label="Individual Capital")
plt.title("Capital Grids")
plt.xlabel("Index")
plt.ylabel("Capital")
plt.legend()
plt.show()
```



4.5 Build The Shock Process

The joint 4x4 matrix stores transitions over the pair (z, ε). From it we recover:

1. a 2x2 aggregate chain for good and bad times
2. the conditional probability of unemployment next period, given the aggregate transition and current employment status

```
def build_shock_process(P):
    """Construct aggregate and conditional objects from the 4×4 joint matrix."""
    P_joint = P.reshape(2, 2, 2, 2)
    P_agg = jnp.array(
        [
            [P[0, 0] + P[0, 1], P[0, 2] + P[0, 3]],
            [P[2, 0] + P[2, 1], P[2, 2] + P[2, 3]],
        ]
    )
    P_cond_unemp = jnp.array(
        [
            [
                [P[0, 0] / P_agg[0, 0], P[1, 0] / P_agg[0, 0]],
                [P[0, 2] / P_agg[0, 1], P[1, 2] / P_agg[0, 1]],
            ],
            [
                [P[2, 0] / P_agg[1, 0], P[3, 0] / P_agg[1, 0]],
                [P[2, 2] / P_agg[1, 1], P[3, 2] / P_agg[1, 1]],
            ],
        ]
    )
    return ShockProcess(
        P=P,
        P_joint=P_joint,
        P_agg=P_agg,
        P_cond_unemp=P_cond_unemp,
    )
```

```
# Rows and columns are ordered as: (bad, unemp), (bad, emp), (good, unemp), (good, emp)
P = jnp.array(
    [
        [0.525000, 0.350000, 0.031250, 0.093750],
        [0.038889, 0.836111, 0.002083, 0.122917],
        [0.093750, 0.031250, 0.291667, 0.583333],
        [0.009115, 0.115885, 0.024306, 0.850694],
    ]
```

```
)

shocks = build_shock_process(P)

print("P_agg:\n", shocks.P_agg)
print("P(unemp'|unemp):\n", shocks.P_cond_unemp[:, :, 0])
print("P(unemp'|emp):\n", shocks.P_cond_unemp[:, :, 1])
```

```
P_agg:
 [[0.875 0.125]
 [0.125 0.875]]
P(unemp'|unemp):
 [[0.6        0.25      ]
 [0.75       0.33333371]]
P(unemp'|emp):
 [[0.04444457 0.016664  ]
 [0.07292    0.02777829]]
```

4.6 Simulate Aggregate And Idiosyncratic Shocks

We now generate one aggregate history and one panel of idiosyncratic employment histories. We will reuse these same shocks inside the outer KS loop so that the forecasting rule is updated against a fixed simulation environment.

```
@jax.jit(static_argnums=(1,))
def simulate_agshock(key, T, shocks):
    """Simulate a T-period aggregate state sequence, coded as 0=bad and 1=good."""
    # Step function: generate next state from previous using transition probabilities
    def step(carry, _):
        key, current_state = carry
        key, subkey = jax.random.split(key)
        # Draw next aggregate state (0=bad, 1=good)
        next_state = jnp.where(
            jax.random.uniform(subkey) < shocks.P_agg[current_state, 0],
            jnp.int32(0),
            jnp.int32(1),
        )
        return (key, next_state), next_state

    # Start in "bad" state (0), run T periods
    (key, _), agshock = lax.scan(step, (key, jnp.int32(0)), None, length=T)
```

```
    return key, agshock


@jax.jit(static_argnums=(2,))
def simulate_idshock(key, agshock, N, shocks, params):
    """Simulate an employment panel of shape (T, N), coded as 0=unemp and 1=emp."""
    # Initial employment status at t=0, random by cross-sectional unemployment rate
    key, subkey = jax.random.split(key)
    id0 = (jax.random.uniform(subkey, (N,)) > params.ur_vals[0]).astype(jnp.int32)

    # Step function: generate next period's employment state for each agent
    # using conditional probabilities given agg state today & tomorrow
    def step(carry, xs):
        key, current_id = carry
        agg_today, agg_tomorrow = xs
        key, subkey = jax.random.split(key)

        draws = jax.random.uniform(subkey, (N,))
        p_unemp = shocks.P_cond_unemp[agg_today, agg_tomorrow][current_id]  # Pr(unemp' | agg, id)
        # 0=unemp, 1=emp
        next_id = jnp.where(draws < p_unemp, jnp.int32(0), jnp.int32(1))
        return (key, next_id), next_id

    # Simulate over all periods, using realized aggregate shocks
    (_, _), id_rest = lax.scan(step, (key, id0), (agshock[:-1], agshock[1:]))
    # Concatenate initial and subsequent periods
    return jnp.concatenate([id0[None], id_rest], axis=0)
```

4.7 Test the Simulation Functions

```
key = jax.random.PRNGKey(42)
key, agshock = simulate_agshock(key, simulation.T, shocks)
idshock = simulate_idshock(key, agshock, simulation.N, shocks, params)

n_bad = (agshock == 0).sum()
n_good = (agshock == 1).sum()

print_summary(
    "Simulated Shock Summary",
    [
```

```
        ("Bad periods", f"{int(n_bad)} ({float(n_bad / simulation.T * 100):.1f}%)"),
        ("Good periods", f"{int(n_good)} ({float(n_good / simulation.T * 100):.1f}%)"),
        (
            "Unemp. rate (bad)",
            f"{float(1 - idshock[agshock == 0].mean()):.4f} (target {float(params.ur_vals[0]):.2f})
        ),
        (
            "Unemp. rate (good)",
            f"{float(1 - idshock[agshock == 1].mean()):.4f} (target {float(params.ur_vals[1]):.2f})
        ),
    ],
)
```

```
Simulated Shock Summary
-----------------------
Bad periods              555 (50.5%)
Good periods             545 (49.5%)
Unemp. rate (bad)        0.0999 (target 0.10)
Unemp. rate (good)       0.0400 (target 0.04)
```

4.8 The Household Problem: A Map Before Code

The main object in the household problem is the savings policy

$$k'(k, K, z, \varepsilon).$$

In code we store it as `policy[i_k, i_K, i_z, i_e]`, so its shape is

$$(n_k, n_K, n_z, n_e).$$

For this lecture:

- axis 0 is individual capital k
- axis 1 is aggregate capital K
- axis 2 is the aggregate state $z \in \{b, g\}$
- axis 3 is the idiosyncratic employment state $\varepsilon \in \{u, e\}$

```
solver_config = SolverConfig(
    update_k=0.70,
    tol_policy=1e-8,
    max_iter_policy=2_000,
    future_states=jnp.array(
        [
            [0, 0],
            [0, 1],
            [1, 0],
            [1, 1],
        ],
        dtype=jnp.int32,
    ),
)

print_summary(
    "Household Solver Setup",
    [
        ("policy shape", f"({ngridk}, {ngridkm}, 2, 2)"),
        ("P_joint shape", f"{shocks.P_joint.shape}"),
        ("Policy tolerance", f"{solver_config.tol_policy:.1e}"),
        ("Max policy iterations", f"{solver_config.max_iter_policy}"),
    ],
)
```

```
Household Solver Setup
----------------------
policy shape             (200, 4, 2, 2)
P_joint shape            (2, 2, 2, 2)
Policy tolerance         1.0e-08
Max policy iterations    2000
```

4.9 Step 1: Prices And Current Resources

The first ingredients are static objects:

1. prices as functions of aggregate capital and the aggregate state
2. labour income as a function of wages and employment
3. current resources at each grid point

```
@jax.jit
def prices_from_capital(K, agg_state, params):
    """Return the rental rate and wage for aggregate capital K in state agg_state."""
    z = params.z_vals[agg_state]
    employment_rate = params.er_vals[agg_state]
    capital_per_eff_labor = K / (employment_rate * params.l_bar)
    r = params.alpha * z * capital_per_eff_labor ** (params.alpha - 1.0)
    w = (1.0 - params.alpha) * z * capital_per_eff_labor ** params.alpha
    return r, w


@jax.jit
def net_labor_income(wage, agg_state, emp_state, params):
    """Labour income net of unemployment insurance transfers and taxes."""
    ur = params.ur_vals[agg_state]
    er = params.er_vals[agg_state]
    employed_income = wage * params.l_bar - params.benefit_rate * wage * (ur / er)
    unemployed_income = params.benefit_rate * wage
    return jnp.where(emp_state == 1, employed_income, unemployed_income)
```

We now build the current-period resource tensor. Its shape is the same as the policy tensor.

```
def build_household_objects(grids, params):
    """Precompute labour income and total resources on the full state grid."""
    k_mesh = grids.k_grid[:, None, None, None]
    km_mesh = grids.km_grid[None, :, None, None]
    agg_mesh = jnp.arange(params.z_vals.shape[0], dtype=jnp.int32)[None, None, :, None]
    emp_mesh = jnp.arange(2, dtype=jnp.int32)[None, None, None, :]

    r, w = prices_from_capital(km_mesh, agg_mesh, params)
    labor_income = net_labor_income(w, agg_mesh, emp_mesh, params)
    resources = (1.0 + r - params.delta) * k_mesh + labor_income

    return HouseholdObjects(
        labor_income=labor_income,
        resources=resources,
    )
```

```
household = build_household_objects(grids, params)
print_summary(
    "Household Objects",
    [
        ("labor_income shape", f"{household.labor_income.shape}"),
        ("resources shape", f"{household.resources.shape}"),
        (
            "resource range",
            f"[{household.resources.min():.4f}, {household.resources.max():.4f}]",
        ),
    ],
)
```

```
Household Objects
-----------------
labor_income shape       (1, 4, 2, 2)
resources shape          (200, 4, 2, 2)
resource range           [0.0000, 1020.3613]
```

4.10 Step 2: Interpolate On The (k, K) Grid

When we update the Euler equation, today's policy choice k' usually lies between grid points. So we need to interpolate the future policy function at off-grid points.

```
@jax.jit
def interp_2d_policy(values, kq, kmq, grids):
    """
    Perform bilinear interpolation over (k, K) grid to estimate the policy value at arbitrary query

    Args:
        values: 2D array of policy function, shaped (k_grid, km_grid).
        kq: array-like, query values for individual capital (k).
        kmq: array-like, query values for aggregate capital (K).
        grids: object containing discretized k_grid and km_grid.

    Returns:
        Interpolated values at (kq, kmq) locations.
    """
    # Clip queries so they're within the grid bounds
    kq = jnp.clip(kq, grids.k_grid[0], grids.k_grid[-1])
    kmq = jnp.clip(kmq, grids.km_grid[0], grids.km_grid[-1])
```

```
    nk = grids.k_grid.shape[0]  # number of individual capital grid points
    nK = grids.km_grid.shape[0]  # number of aggregate capital grid points

    # Find the upper and lower indices for interpolation in k_grid
    ik_hi = jnp.clip(jnp.searchsorted(grids.k_grid, kq, side="right"), 1, nk - 1)
    ik_lo = ik_hi - 1
    # Find the upper and lower indices for interpolation in km_grid
    iK_hi = jnp.clip(jnp.searchsorted(grids.km_grid, kmq, side="right"), 1, nK - 1)
    iK_lo = iK_hi - 1

    # Gather grid values at those indices
    k_lo = grids.k_grid[ik_lo]
    k_hi = grids.k_grid[ik_hi]
    K_lo = grids.km_grid[iK_lo]
    K_hi = grids.km_grid[iK_hi]

    # Compute normalized distances between grid points (weights)
    wk = (kq - k_lo) / (k_hi - k_lo)
    wK = (kmq - K_lo) / (K_hi - K_lo)

    # Obtain function values at the four corners
    f00 = values[ik_lo, iK_lo]  # bottom-left
    f10 = values[ik_hi, iK_lo]  # bottom-right
    f01 = values[ik_lo, iK_hi]  # top-left
    f11 = values[ik_hi, iK_hi]  # top-right

    # Compute weighted sum for bilinear interpolation
    return (
        (1.0 - wk) * (1.0 - wK) * f00
        + wk * (1.0 - wK) * f10
        + (1.0 - wk) * wK * f01
        + wk * wK * f11
    )
```

4.11 Step 3: Forecast Next Period Aggregate Capital

The KS approximation replaces the full law of motion of the distribution by a simple forecasting rule:

$$\log K' = B_{0,z} + B_{1,z} \log K.$$

We store the coefficients in

$$B = \begin{bmatrix} b_0 & b_1 \\ a_0 & a_1 \end{bmatrix},$$

where row 0 is the bad state and row 1 is the good state.

```
@jax.jit
def forecast_next_aggregate_capital(B, grids):
    """Compute K'(K, z) on the aggregate-capital grid."""
    log_K = jnp.log(grids.km_grid)[None, :]
    log_K_next = B[:, [0]] + B[:, [1]] * log_K
    K_next = jnp.exp(log_K_next)
    return jnp.clip(K_next.T, grids.km_min, grids.km_max)
```

4.12 Step 4: Rewrite The Euler Update As Small Pieces

This is the core numerical step. Conceptually we do four things:

1. take a current guess for k'
2. use it to evaluate future savings k''
3. compute expected marginal utility tomorrow
4. back out current consumption and current savings

The only difficult part is bookkeeping over future states, so we make that bookkeeping explicit.

```
@jax.jit
def reshape_policy_by_future_state(policy):
    """Reorder policy to shape (future_state, k, K)."""
    nk, nK = policy.shape[:2]
    return policy.transpose(2, 3, 0, 1).reshape(-1, nk, nK)


@jax.jit
def reshape_joint_probabilities(shocks):
    """Reorder P(z', e' | z, e) to shape (future_state, z, e)."""
    return shocks.P_joint.reshape(2, 2, -1).transpose(2, 0, 1)
```

For one specific future state (z', ε'), we can compute the Euler-equation contribution state by state.

```python
@jax.jit
def marginal_value_in_future_state(
    future_index,
    policy,
    predicted_K_next,
    future_policy_slices,
    params,
    grids,
    solver_config,
):
    """Return the Euler RHS contribution for one future state (z', eps')."""

    # Get aggregate (z') and employment (eps') state indices for this future state
    next_agg = solver_config.future_states[future_index, 0]
    next_emp = solver_config.future_states[future_index, 1]

    # Interpolate future savings policy k'' at this future state
    kpp = interp_2d_policy(
        future_policy_slices[future_index],
        policy,
        predicted_K_next,
        grids,
    )

    # Compute next period interest rate and wage given K'
    r_next, w_next = prices_from_capital(predicted_K_next, next_agg, params)
    # Compute labor income for next period
    labor_income_next = net_labor_income(w_next, next_agg, next_emp, params)

    # Compute next period consumption using budget constraint
    c_next = jnp.maximum(
        (1.0 + r_next - params.delta) * policy + labor_income_next - kpp,
        1e-12,  # avoid negative or zero consumption
    )

    # Compute and return discounted marginal utility term, as Euler equation RHS
    return (1.0 + r_next - params.delta) * c_next ** (-params.sigma)
```

Now we aggregate over the four future states with the correct transition probabilities.

```
@jax.jit
def expected_euler_rhs(
    policy,
    B,
    params,
    grids,
    shocks,
    solver_config,
):
    """
    Compute the expected RHS of the Euler equation at each current state.

    Returns E[(1 + r' - delta) * u_c(c')].
    - policy: current savings policy.
    - B: aggregate law coefficients.
    - params: model parameters.
    - grids: state grids.
    - shocks: idiosyncratic and aggregate shock structure.
    - solver_config: solver settings and state info.
    """
    # Forecast next-period aggregate capital from law of motion
    K_next = forecast_next_aggregate_capital(B, grids)
    # Broadcast K_next for shape compatibility with policy grid
    predicted_K_next = jnp.broadcast_to(K_next[None, :, :, None], policy.shape)

    # Reorganize policy by future states (z', eps') for vectorized calls
    future_policy_slices = reshape_policy_by_future_state(policy)
    # Reshape transition probabilities by future state
    future_probabilities = reshape_joint_probabilities(shocks)

    # Compute the RHS for each possible future state
    rhs_terms = jax.vmap(
        lambda future_index: marginal_value_in_future_state(
            future_index,
            policy,
            predicted_K_next,
            future_policy_slices,
            params,
            grids,
            solver_config,
        )
    )(jnp.arange(solver_config.future_states.shape[0]))
```

```
    # Weighted sum over future states using joint transition probabilities
    return jnp.sum(rhs_terms * future_probabilities[:, None, None, :, :], axis=0)
```

Finally, the Euler equation gives current consumption, and the budget constraint gives updated savings.

```
@jax.jit
def update_policy_once(policy, B, params, grids, shocks, household, solver_config):
    """One policy update implied by the Euler equation."""
    expected_rhs = expected_euler_rhs(policy, B, params, grids, shocks, solver_config)
    c_now = jnp.maximum((params.beta * expected_rhs) ** (-1.0 / params.sigma), 1e-12)
    kprime_new = household.resources - c_now
    return jnp.clip(kprime_new, grids.k_min, grids.k_max)
```

4.13 Step 5: Policy Function Iteration

The inner fixed point uses `lax.while_loop`, so the full iteration can still be compiled by JAX.

```
@jax.jit
def solve_individual_problem(
    B, policy_init, params, grids, shocks, household, solver_config
):
    """
    Iteratively solve for optimal individual savings policy k'(k, K, z, eps)
    given the perceived aggregate-law coefficients B, until convergence.

    Returns:
        policy_star: Converged policy function for savings
        c_star: Implied optimal consumption policy
        it: Number of iterations performed
        diff: Final sup-norm difference between iterations
    """

    def cond_fun(carry):
        # Continue while not converged and iteration limit not reached
        it, diff, policy = carry
        return jnp.logical_and(
            diff > solver_config.tol_policy,
            it < solver_config.max_iter_policy,
        )
```

```
    def body_fun(carry):
        # One policy iteration: update guess, check difference, apply damping
        it, _, policy = carry
        new_policy = update_policy_once(
            policy, B, params, grids, shocks, household, solver_config
        )
        diff = jnp.max(jnp.abs(new_policy - policy))  # Sup-norm difference
        # Damp the update for stability
        damped_policy = (
            solver_config.update_k * new_policy
            + (1.0 - solver_config.update_k) * policy
        )
        return it + 1, diff, damped_policy

    # Start with initial guess, infinite diff, and zero iterations
    init_carry = (jnp.int32(0), jnp.asarray(jnp.inf), policy_init)
    # Run the fixed-point iteration loop
    it, diff, policy_star = lax.while_loop(cond_fun, body_fun, init_carry)
    # Compute optimal consumption policy (budget constraint)
    c_star = jnp.maximum(household.resources - policy_star, 1e-12)
    return policy_star, c_star, it, diff
```

4.14 Initial Guess And First Solution

We start from the simplest possible forecast:

$$\log K' = \log K$$

in both aggregate states, and from a policy that saves 90 percent of individual capital everywhere on the grid.

```
B0 = jnp.array(
    [
        [0.0, 1.0],
        [0.0, 1.0],
    ]
)

policy_init = jnp.broadcast_to(
    0.9 * grids.k_grid[:, None, None, None],
```

```
    (grids.k_grid.shape[0], grids.km_grid.shape[0], 2, 2),
)
print(f"policy_init shape: {policy_init.shape}")
```

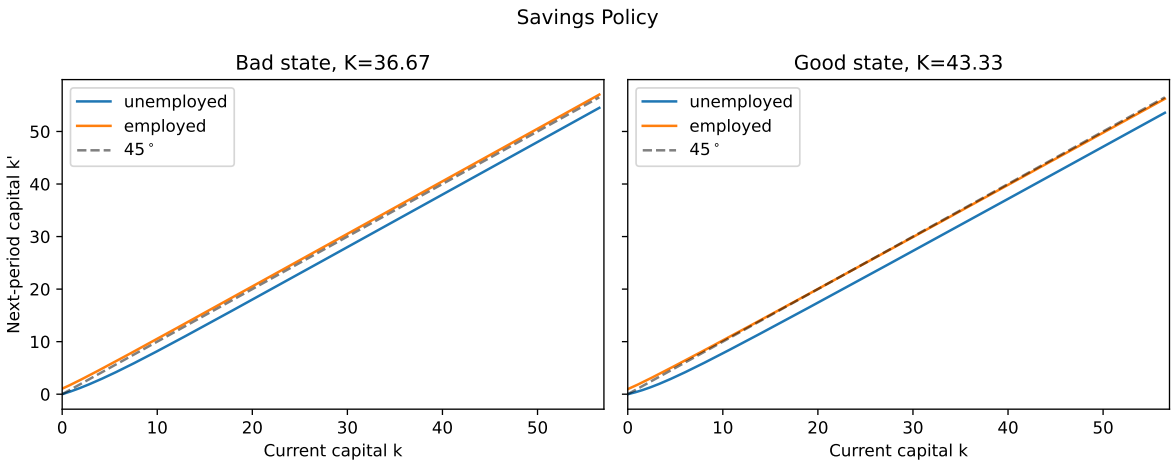
```
policy_init shape: (200, 4, 2, 2)
```

```
(kprime, c_policy, iters, final_diff), policy_elapsed = time_call(
    solve_individual_problem,
    B0,
    policy_init,
    params,
    grids,
    shocks,
    household,
    solver_config,
    warmup=True,
)

print_summary(
    "Individual Problem Summary",
    [
        ("Elapsed time", f"{policy_elapsed:.4f} s"),
        ("Iterations", f"{int(iters)}"),
        ("Final sup norm", f"{float(final_diff):.3e}"),
        ("Policy shape", f"{kprime.shape}"),
        (
            "Consumption range",
            f"[{float(c_policy.min()):.4f}, {float(c_policy.max()):.4f}]",
        ),
    ],
)
```

```
Individual Problem Summary
--------------------------
Elapsed time             0.2464 s
Iterations               1887
Final sup norm           9.979e-09
Policy shape             (200, 4, 2, 2)
Consumption range        [0.0000, 20.3613]
```

4.15 Visualize the Policy

```
k_max = 1.5 * kss
mask = (grids.k_grid >= 0) & (grids.k_grid <= k_max)
k_grid_plot = grids.k_grid[mask]

fig, axes = plt.subplots(1, 2, figsize=(10, 4), sharey=True)

axes[0].plot(k_grid_plot, kprime[mask, 1, 0, 0], label="unemployed")
axes[0].plot(k_grid_plot, kprime[mask, 1, 0, 1], label="employed")
axes[0].plot(k_grid_plot, k_grid_plot, "k--", alpha=0.5, label=r"$45^\circ$")
axes[0].set_title(f"Bad state, K={float(grids.km_grid[1]):.2f}")
axes[0].set_xlabel("Current capital k")
axes[0].set_ylabel("Next-period capital k'")
axes[0].set_xlim(0, k_max)
axes[0].legend()

axes[1].plot(k_grid_plot, kprime[mask, 2, 1, 0], label="unemployed")
axes[1].plot(k_grid_plot, kprime[mask, 2, 1, 1], label="employed")
axes[1].plot(k_grid_plot, k_grid_plot, "k--", alpha=0.5, label=r"$45^\circ$")
axes[1].set_title(f"Good state, K={float(grids.km_grid[2]):.2f}")
axes[1].set_xlabel("Current capital k")
axes[1].set_xlim(0, k_max)
axes[1].legend()

plt.suptitle("Savings Policy")
plt.tight_layout()
plt.show()
```



4.16 Simulate The Economy Given A Policy

Once we have a policy, the next KS step is mechanical: simulate many households, average their capital, and recover the implied aggregate capital path.

```
@jax.jit
def simulate_aggregate_path(policy, agshock, idshock, kcross0, grids):
    """
    Simulate the evolution of aggregate capital over time given a household policy and realized sho

    Args:
        policy: Household policy function (capital choice).
        agshock: Sequence of aggregate shocks (indices).
        idshock: Sequence of idiosyncratic employment status shocks (0=unemployed, 1=employed).
        kcross0: Initial capital levels for all households.
        grids: Grid information for interpolation and bounds.

    Returns:
        kmts: Sequence of aggregate capital (mean) over time (truncated to grid min/max), shape (T,
        kcross_T: Final cross-sectional capital levels, shape (N,).
    """

    def step(kcross, shocks_t):
        # Unpack current shocks for this period: agg_t (aggregate state), id_t (idiosyncratic emplo
        agg_t, id_t = shocks_t

        # Compute aggregate capital, clipped to allowed range
        km_t = jnp.clip(jnp.mean(kcross), grids.km_min, grids.km_max)

        # Interpolate next-period capital according to individual status (u: unemployed, e: employe
        kprime_u = interp_2d_policy(policy[:, :, agg_t, 0], kcross, km_t, grids)
        kprime_e = interp_2d_policy(policy[:, :, agg_t, 1], kcross, km_t, grids)

        # Select next-period capital based on employment status
        kcross_next = jnp.where(id_t == 0, kprime_u, kprime_e)

        # Apply bounds to capital choices
        kcross_next = jnp.clip(kcross_next, grids.k_min, grids.k_max)

        return kcross_next, km_t

    # Run the simulation forward in time for each period's shocks
    kcross_T, kmts = lax.scan(step, kcross0, (agshock, idshock))
```

```
    # Return path of mean capital and final distribution of capital
    return kmts, kcross_T
```

4.17 Test the Simulation Function

```
kcross0 = jnp.full((simulation.N,), kss)
kmts, kcross_T = simulate_aggregate_path(kprime, agshock, idshock, kcross0, grids)

print_summary(
    "Aggregate Simulation Summary",
    [
        ("Path shape", f"{kmts.shape}"),
        ("Initial K", f"{float(kmts[0]):.4f}"),
        ("Mean K", f"{float(kmts.mean()):.4f}"),
        ("Final mean K", f"{float(kcross_T.mean()):.4f}"),
    ],
)
```

```
Aggregate Simulation Summary
----------------------------
Path shape               (1100,)
Initial K                37.9893
Mean K                   39.9642
Final mean K             39.3181
```

4.18 Update The Forecasting Rule

Given a simulated aggregate path, the KS update is just two regressions:

$$\log K_{t+1} = b_0 + b_1 \log K_t \quad \text{in bad times,}$$
$$\log K_{t+1} = a_0 + a_1 \log K_t \quad \text{in good times.}$$

This step does not need JAX, so ordinary NumPy is perfectly fine.

```
def ols_coefficients(x, y):
    """OLS with an intercept, using NumPy host arrays."""
    X = np.column_stack([np.ones_like(x), x])
    beta_hat, _, _, _ = np.linalg.lstsq(X, y, rcond=None)
    y_hat = X @ beta_hat
    ssr = np.sum((y - y_hat) ** 2)
    sst = np.sum((y - y.mean()) ** 2)
    r2 = 1.0 - ssr / sst if sst > 0 else np.nan
    return beta_hat, r2


def update_forecasting_rule(kmts, agshock, ndiscard):
    """Run separate KS regressions in bad and good aggregate states."""
    km_np = np.asarray(kmts)
    ag_np = np.asarray(agshock)

    x = np.log(km_np[ndiscard:-1])
    y = np.log(km_np[ndiscard + 1 :])
    current_state = ag_np[ndiscard:-1]

    bad_mask = current_state == 0
    good_mask = current_state == 1

    B_bad, R2_bad = ols_coefficients(x[bad_mask], y[bad_mask])
    B_good, R2_good = ols_coefficients(x[good_mask], y[good_mask])

    return RegressionResults(
        B_hat=np.vstack([B_bad, B_good]),
        R2_bad=float(R2_bad),
        R2_good=float(R2_good),
        n_bad=int(bad_mask.sum()),
        n_good=int(good_mask.sum()),
    )
```

4.19 Test the Regression Function

```
regression = update_forecasting_rule(kmts, agshock, simulation.ndiscard)

print_summary(
    "Regression Update Summary",
```

```
    [
        ("B (bad state)", f"{regression.B_hat[0]}"),
        ("B (good state)", f"{regression.B_hat[1]}"),
        ("R^2 bad", f"{regression.R2_bad:.6f}"),
        ("R^2 good", f"{regression.R2_good:.6f}"),
        ("Obs. bad", f"{regression.n_bad}"),
        ("Obs. good", f"{regression.n_good}"),
    ],
)
```

```
Regression Update Summary
-------------------------
B (bad state)            [0.44174589 0.87942783]
B (good state)           [0.45837376 0.87654118]
R^2 bad                  0.999722
R^2 good                 0.999830
Obs. bad                 506
Obs. good                493
```

4.20 One Complete KS Update

At this point the full logic is easy to read: solve, simulate, regress.

```
def ks_update_step(
    B,
    policy_init,
    kcross0,
    agshock,
    idshock,
    params,
    grids,
    shocks,
    household,
    simulation,
    solver_config,
):
    """
    Perform one step of the KS (Krusell–Smith) outer update.

    Solves the individual household problem given guess B, simulates the aggregate path,
    then runs the forecasting regression update, returning all key objects.
```

```
    Returns:
        policy: updated individual policy function
        c_policy: updated consumption policy function
        kmts: simulated aggregate capital path
        kcross_T: simulated individual-level capital distribution (final)
        B_hat: updated forecasting coefficients (as jnp array)
        regression: full regression result object
    """
    # Solve individual optimization given current forecasting rule coefficients B
    policy, c_policy, _, _ = solve_individual_problem(
        B, policy_init, params, grids, shocks, household, solver_config
    )

    # Simulate the capital path and individual cross-sectional distribution
    kmts, kcross_T = simulate_aggregate_path(policy, agshock, idshock, kcross0, grids)

    # Update forecasting rule via regression on simulated data
    regression = update_forecasting_rule(kmts, agshock, simulation.ndiscard)

    # Return all relevant objects for outer update loop
    return (
        policy,  # new individual policy function
        c_policy,  # new consumption policy function
        kmts,  # simulated aggregate capital over time
        kcross_T,  # cross-sectional capital distribution at end
        jnp.asarray(regression.B_hat),  # updated B coefficients (as JAX array)
        regression,  # full regression object (with R^2, n, etc.)
    )
```

4.21 Show one step of the KS update

```
(policy_1, c_policy_1, kmts_1, kcross_1, B_hat_1, regression_1), ks_step_elapsed = (
    time_call(
        ks_update_step,
        B0,
        policy_init,
        kcross0,
        agshock,
        idshock,
        params,
```

```
        grids,
        shocks,
        household,
        simulation,
        solver_config,
    )
)

print_summary(
    "One-Step KS Update",
    [
        ("Elapsed time", f"{ks_step_elapsed:.4f} s"),
        ("||B_hat - B0||", f"{float(jnp.linalg.norm(B_hat_1 - B0)):.3e}"),
        ("R^2 bad", f"{regression_1.R2_bad:.6f}"),
        ("R^2 good", f"{regression_1.R2_good:.6f}"),
    ],
)
```

```
One-Step KS Update
------------------
Elapsed time             0.3767 s
||B_hat - B0||           6.596e-01
R^2 bad                  0.999722
R^2 good                 0.999830
```

4.22 Full Outer Loop

Now we simply repeat the previous step until the forecasting coefficients stop moving.

```
outer_config = KSOuterConfig(
    tol_B=1e-6,
    max_iter_B=50,
    update_B=0.50,
)
```

4.23 Warm Start In The Outer Loop

- In the KS algorithm, the forecasting coefficients B usually change only gradually across outer iterations.

- Because of that, the converged savings policy from iteration n is already a very good initial guess for iteration $n + 1$.
- So instead of restarting every inner solve from the same crude `policy_init`, we **warm start** from the previous converged policy.
- This does not change the fixed point we are targeting. It only makes the inner policy iteration start much closer to its new solution.

4.24 Warm Start In The Outer Loop

- The implementation is simple:
  - set `current_policy = policy_init` before the loop
  - solve the household problem using `current_policy`
  - after the solve, update `current_policy = policy`
  - use this updated policy as the initial guess in the next outer iteration

In practice, this substantially reduces the number of inner policy iterations in later KS rounds, so it is a pure algorithmic speedup with no change in the target solution.

We now define whole soltion function for the KS model.

```
def solve_ks_model(
    B_init,
    policy_init,
    kcross0,
    agshock,
    idshock,
    params,
    grids,
    shocks,
    household,
    simulation,
    solver_config,
    outer_config,
):
    """
    Solves the KS (Krusell–Smith) model using an outer fixed-point iteration
    on the forecasting rule coefficients. The routine updates the perceived
    law of motion until convergence.
```

```
    """
    B = np.asarray(B_init)  # Initial perceived law of motion coefficients
    current_kcross = jnp.asarray(
        kcross0
    )  # Initial cross-sectional capital distribution
    current_policy = jnp.asarray(policy_init)  # Initial policy guess
    history = []  # To store convergence history

    for it in range(
        outer_config.max_iter_B
    ):  # Main outer loop over forecasting rule updates
        # Solve the individual (household) dynamic programming problem
        policy, c_policy, _, _ = solve_individual_problem(
            jnp.asarray(B),
            current_policy,
            params,
            grids,
            shocks,
            household,
            solver_config,
        )
        # Simulate the aggregate capital path and update cross-sectional distribution
        kmts, current_kcross = simulate_aggregate_path(
            policy, agshock, idshock, current_kcross, grids
        )
        # Fit/update the forecasting rule via regression
        regression = update_forecasting_rule(kmts, agshock, simulation.ndiscard)

        B_hat = regression.B_hat  # Updated law of motion coefficients
        diff_B = float(np.linalg.norm(B_hat - B))  # Difference in coefficients

        history.append(
            dict(
                iteration=it + 1,
                diff_B=diff_B,
                R2_bad=regression.R2_bad,
                R2_good=regression.R2_good,
            )
        )

        # Use converged policy for next iteration (warm start)
        current_policy = policy
```

```
        # Relaxation update of coefficient vector
        B = outer_config.update_B * B_hat + (1.0 - outer_config.update_B) * B

        if diff_B <= outer_config.tol_B:  # Check for convergence
            break

    return dict(
        B=jnp.asarray(B),  # Converged coefficients (JAX array)
        policy=policy,  # Final policy function
        consumption=c_policy,  # Corresponding consumption policy
        kmts=kmts,  # Simulated aggregate capital path
        kcross=current_kcross,  # Final cross-sectional capital distribution
        history=history,  # Iteration history: convergence stats
        regression=regression,  # Final regression object
    )
```

```
ks_solution, ks_elapsed = time_call(
    solve_ks_model,
    B0,
    policy_init,
    kcross0,
    agshock,
    idshock,
    params,
    grids,
    shocks,
    household,
    simulation,
    solver_config,
    outer_config,
)

ks_history = ks_solution["history"]
ks_last = ks_history[-1]

print_summary(
    "Full KS Solve Summary",
    [
        ("Elapsed time", f"{ks_elapsed:.4f} s"),
```

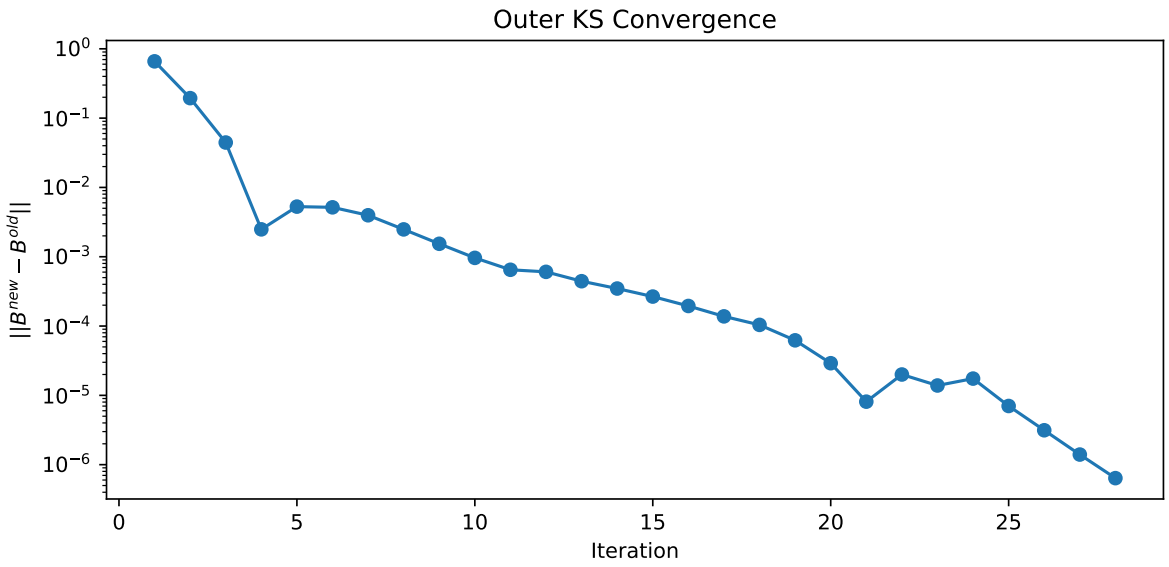
```
        ("Outer iterations", f"{len(ks_history)}"),
        ("Final diff_B", f"{ks_last['diff_B']:.3e}"),
        ("R^2 bad", f"{ks_last['R2_bad']:.6f}"),
        ("R^2 good", f"{ks_last['R2_good']:.6f}"),
        ("B (bad state)", f"{ks_solution['B'][0]}"),
        ("B (good state)", f"{ks_solution['B'][1]}"),
    ],
)
speedup = 25.3100 / ks_elapsed
print(f"Speedup: {speedup:.2f}x")
```

```
Full KS Solve Summary
---------------------
Elapsed time             7.5960 s
Outer iterations         28
Final diff_B             6.390e-07
R^2 bad                  0.999927
R^2 good                 0.999964
B (bad state)            [0.14500784 0.96016969]
B (good state)           [0.15565201 0.95866002]
Speedup: 3.33x
```

The original Maliar, Maliar, and Valli (2010) MATLAB code took 25.3 seconds to solve the model, so the JAX implementation is much faster.

4.25 Visualize the Convergence

```
plt.figure(figsize=(8, 4))
plt.semilogy(
    [row["iteration"] for row in ks_history],
    [row["diff_B"] for row in ks_history],
    marker="o",
)
plt.title("Outer KS Convergence")
plt.xlabel("Iteration")
plt.ylabel(r"$\| B^{new} - B^{old} \|$")
plt.tight_layout()
plt.show()
```



4.26 Forecast Accuracy Plot

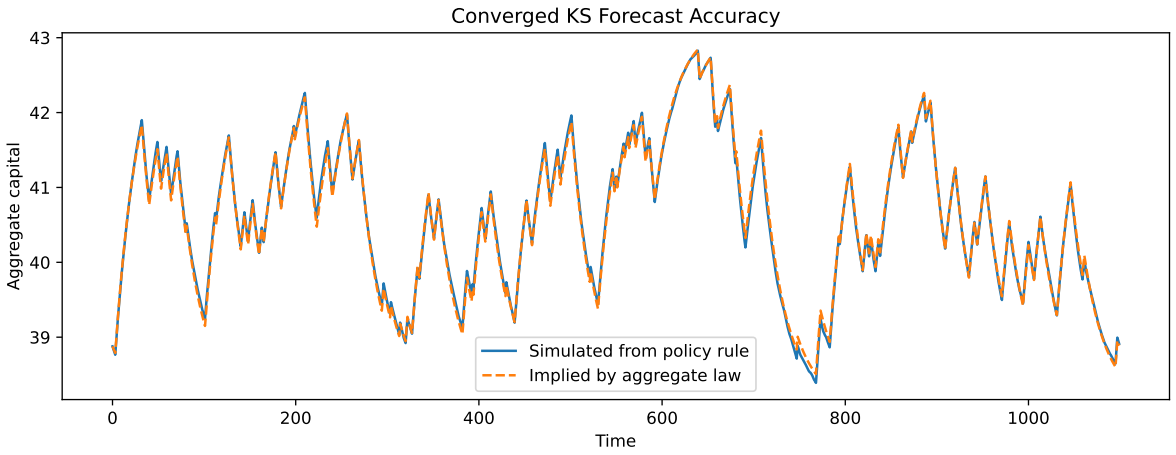
As a final diagnostic, we compare the aggregate capital path generated by the household policy with the path implied by the converged perceived law of motion itself.

```
@jax.jit
def simulate_alm_path(B, km0, agshock, grids):
    """Simulate the path implied directly by the aggregate law of motion."""

    def step(km_t, agg_t):
        log_km_next = B[agg_t, 0] + B[agg_t, 1] * jnp.log(km_t)
        km_next = jnp.clip(jnp.exp(log_km_next), grids.km_min, grids.km_max)
        return km_next, km_next

    _, km_next = lax.scan(step, km0, agshock[:-1])
    return jnp.concatenate([jnp.array([km0]), km_next])
```

```
kmalm_final = simulate_alm_path(
    ks_solution["B"], ks_solution["kmts"][0], agshock, grids
)

diff_kmt = 100 * (kmalm_final[-1] - ks_solution["kmts"][-1]) / ks_solution["kmts"][-1]
print(f"Difference in final aggregate capital: {diff_kmt:.4f}%")
```

```
Difference in final aggregate capital: -0.0613%
```

```
plt.figure(figsize=(10, 4))
plt.plot(ks_solution["kmts"], label="Simulated from policy rule")
plt.plot(kmalm_final, "--", label="Implied by aggregate law")
plt.title("Converged KS Forecast Accuracy")
plt.xlabel("Time")
plt.ylabel("Aggregate capital")
plt.legend()
plt.tight_layout()
plt.show()
```



References

Chipeniuk, Karsten, Nets Hawk Katz, and Todd Walker. 2016. "Approximate Aggregation in the Neoclassical Growth Model with Idiosyncratic Shocks."

Haan, Wouter J. Den, Kenneth L. Judd, and Michel Juillard. 2010. "Computational Suite of Models with Heterogeneous Agents: Incomplete Markets and Aggregate Uncertainty." *Journal of Economic Dynamics and Control* 34 (1): 1–3.

Maliar, Lilia, Serguei Maliar, and Fernando Valli. 2010. "Solving the Incomplete Markets Model with Aggregate Uncertainty Using the Krusell–Smith Algorithm." *Journal of Economic Dynamics and Control* 34 (1): 42–49.

Werning, Iván. 2015. "Incomplete Markets and Aggregate Demand." w. National Bureau of Economic Research.

Young, Eric R. 2010. "Solving the Incomplete Markets Model with Aggregate Uncertainty Using the Krusell–Smith Algorithm and Non-Stochastic Simulations." *Journal of Economic Dynamics and Control* 34 (1): 36–41.