Continuous Time Heterogeneous Agent Models

Lecturer: Bo Li TA: Chen Gao

2026-05-12

Table of contents

1 Continuous Time Huggett (1993) 1

2 Solve the HJB Equation using Explicit Method 11

3 Code for Solve the HJB Equation using Explicit Method 15

4 Solve the HJB Equation using Implicit Method 26

5 Code for Solve the HJB Equation using Implicit Method 29

6 Implementation of the Kolmogorov Forward Equation 41

7 Code for Solving the KFE 42

8 Equilibrium 47

1 Continuous Time Huggett (1993)

1.1 Income Process

The income process $y_{i,t}$ is defined as a **Continuous-Time Markov Chain (CTMC)** on a finite state space $\mathcal{Y} = \{y_1, y_2\}$.

Suppose that at time t, the state is $y_{i,t}$. Over an infinitesimally small time interval $\Delta t \to 0$:

- If the current state is y_1:
  - The probability of remaining in y_1 is $1 - \lambda_1 \Delta t + o(\Delta t)$.
  - The probability of jumping to y_2 is $\lambda_1 \Delta t + o(\Delta t)$.

- If the current state is y_2:
  - The probability of remaining in y_2 is $1 - \lambda_2 \Delta t + o(\Delta t)$.
  - The probability of jumping to y_1 is $\lambda_2 \Delta t + o(\Delta t)$.
- Here, $o(\Delta t)$ denotes terms that are infinitesimal of higher order than Δt.

1.2 Formal Definition

$$\begin{aligned}
\mathbb{P}(y_{i,t+\Delta t} = y_2 \mid y_{i,t} = y_1) &= \lambda_1 \Delta t + o(\Delta t) \\
\mathbb{P}(y_{i,t+\Delta t} = y_1 \mid y_{i,t} = y_1) &= 1 - \lambda_1 \Delta t + o(\Delta t) \\
\mathbb{P}(y_{i,t+\Delta t} = y_1 \mid y_{i,t} = y_2) &= \lambda_2 \Delta t + o(\Delta t) \\
\mathbb{P}(y_{i,t+\Delta t} = y_2 \mid y_{i,t} = y_2) &= 1 - \lambda_2 \Delta t + o(\Delta t)
\end{aligned}$$

Definition 1.1. The process is characterized by the intensity matrix (or generator matrix) Λ:

$$\Lambda = \begin{pmatrix} -\lambda_1 & \lambda_1 \\ \lambda_2 & -\lambda_2 \end{pmatrix}$$

Let $P(t)$ be the row vector of probabilities $[p_1(t), p_2(t)]$ where $p_k(t) = \mathbb{P}(y_{i,t} = y_k)$. The evolution of the distribution follows the Kolmogorov Forward Equation:

$$\frac{dP(t)}{dt} = P(t)\Lambda$$

1.3 Households

Definition 1.2. The problem of household $i \in [0, 1]$ (in sequence form) is

$$\max_{\{c_{i,t}\}_{t\geq 0}} \mathbb{E}_0 \int_0^\infty e^{-\rho t} u(c_{i,t}) dt$$

$$\text{s.t.} \quad \dot{a}_{i,t} = w_t y_{i,t} + r_t a_{i,t} - c_{i,t}$$

$$a_{i,t} \geq \underline{a}$$

where $y_{i,t}$ is given by Definition 1.1 and the initial condition is $(a_{i,0}, y_{i,0})$.

A solution to the household problem is a stochastic process $\{c_{i,t}, a_{i,t}\}_{t\geq 0}$

1.4 Firms

- **Production Technology:**
  A representative firm produces the homogeneous final consumption good according to

$$Y_t = A_t \ell_t$$

  where Y_t is output, A_t is productivity, and ℓ_t is labor input.

- **Firm Optimization Problem:**
  Firms are small and perfectly competitive, so each firm solves

$$\max_{\ell_t} \; Y_t - w_t \ell_t$$

  where w_t is the (endogenous) wage.

- **Firm Problem is Static:**
  The firm's problem is static (one period). If it were dynamic, we would also need to model firm ownership.

- **Market Clearing Conditions:**

  - **Goods Market:**

$$Y_t = \int_0^1 c_{i,t} \, di$$

  - **Labor Market:**

$$\ell_t = \int_0^1 y_{i,t} \, di$$

  - **Assets Market:**

$$0 = \int_0^1 a_{i,t} \, di$$

1.5 Competitive Equilibrium

Now we can define the competitive equilibrium.

Definition 1.3. Given an initial distribution of assets and individual labor productivities $\{a_{i,0}, y_{i,0}\}_i$ and an exogenous path for TFP $\{A_t\}$, a **competitive equilibrium** consists of:

- An allocation $\{Y_t, \ell_t, c_{i,t}, a_{i,t}\}$
- Prices $\{r_t, w_t\}$

such that:

1. Households optimize
2. Firms optimize
3. Markets clear

1.6 Derivation of the HJB Equation

Consider a very short time interval Δt. According to the Bellman equation, the value function at time t, $V(a_t, y_t)$, can be written as:

$$V(a_t, y_t) = \max_{c_t} \left\{ \int_t^{t+\Delta t} e^{-\rho(s-t)} u(c_s) ds + e^{-\rho \Delta t} \mathbb{E}_t \left[V(a_{t+\Delta t}, y_{t+\Delta t}) \right] \right\} \tag{1}$$

When Δt is sufficiently small, we can make the following approximations:

1. **Flow utility:** $\int_t^{t+\Delta t} e^{-\rho(s-t)} u(c_s) ds \approx u(c_t) \Delta t$
2. **Discount factor:** $e^{-\rho \Delta t} \approx 1 - \rho \Delta t$

Substituting these, we obtain the approximate equation: [1]

$$V(a, y) \approx \max_c \{ u(c) \Delta t + (1 - \rho \Delta t) \mathbb{E}_t [V(a', y')] \}$$

1.7 Expectation of the Value Function

The expectation $\mathbb{E}_t$ must account for the Poisson jumps in the state y. According to Definition 1.1, suppose we are currently in state y_1:

- **Case A (No Jump):** Probability is $1 - \lambda_1 \Delta t$.
  - The income state remains at y_1.
  - The asset evolves to $a' = a + \dot{a} \Delta t$, where $\dot{a} = w y_1 + r a - c$.
- **Case B (Jump):** Probability is $\lambda_1 \Delta t$.
  - The income state jumps to y_2.
  - The asset is approximately unchanged at a instantly (since asset is a stock variable and cannot jump immediately).

[1] Note: For simplicity, we omit the time subscript t, and use a' and y' to denote the state at time $t + \Delta t$.

Therefore, the expectation term unfolds as:

$$\mathbb{E}_t[V(a', y')] \approx (1 - \lambda_1 \Delta t) \cdot V(a + \dot{a}\Delta t, y_1) + (\lambda_1 \Delta t) \cdot V(a, y_2)$$

Now, we need to take a first-order Taylor expansion of $V(a + \dot{a}\Delta t, y_1)$:

$$V(a + \dot{a}\Delta t, y_1) \approx V(a, y_1) + \frac{\partial V(a, y_1)}{\partial a} \cdot \dot{a}\Delta t$$

Substitute this expansion into the expectation formula in Step 2, and neglect higher order terms (Δt^2 and above), since they vanish in the limit:

$$\mathbb{E}_t[V(a', y')] \approx (1 - \lambda_1 \Delta t) \left[V(a, y_1) + V_a(a, y_1)\dot{a}\Delta t\right] + \lambda_1 \Delta t V(a, y_2)$$

$$\approx V(a, y_1) + V_a(a, y_1)\dot{a}\Delta t - \lambda_1 \Delta t V(a, y_1) + \lambda_1 \Delta t V(a, y_2)$$

$$= V(a, y_1) + V_a(a, y_1)\dot{a}\Delta t + \lambda_1 \Delta t [V(a, y_2) - V(a, y_1)]$$

Substitute the result back into the approximate Bellman equation Equation 1:

$$V(a, y_1) \approx \max_c \{u(c)\Delta t + (1 - \rho\Delta t) \left[V(a, y_1) + V_a \dot{a}\Delta t + \lambda_1 \Delta t (V(a, y_2) - V(a, y_1))\right]\}$$

Expand the right-hand side and subtract $V(a, y_1)$ from both sides:

$$V(a, y_1) \approx u(c)\Delta t + V(a, y_1) - \rho V(a, y_1)\Delta t + V_a \dot{a}\Delta t + \lambda_1 (V(a, y_2) - V(a, y_1))\Delta t + o(\Delta t)$$

Subtract $V(a, y_1)$ from both sides and collect terms with Δt:

$$\rho V(a, y_1)\Delta t \approx u(c)\Delta t + V_a \dot{a}\Delta t + \lambda_1 (V(a, y_2) - V(a, y_1))\Delta t$$

Finally, divide both sides by Δt and take the limit as $\Delta t \to 0$ to obtain the HJB equation.

For state y_1, the HJB equation is:

$$\rho V(a, y_1) = \max_c \{u(c) + V_a(a, y_1) \cdot (wy_1 + ra - c) + \lambda_1[V(a, y_2) - V(a, y_1)]\}$$

Similarly, for state y_2:

$$\rho V(a, y_2) = \max_c \{u(c) + V_a(a, y_2) \cdot (wy_2 + ra - c) + \lambda_2[V(a, y_1) - V(a, y_2)]\}$$

Intuition

On the left side, ρV is the required rate of return (analogous to the risk-free rate in asset pricing). The right side gives the actual return provided by this "asset" (the agent's lifetime), which consists of three parts:

1. **Dividend**: The current utility from consumption, $u(c)$.
2. **Capital Gain due to Savings**: The value increase from accumulating wealth, $V_a \cdot \dot{a}$.
3. **Expected Jump Gain/Loss**: The expected change in value arising from sudden switches in the income state, $\lambda[V(y_{\text{new}}) - V(y_{\text{old}})]$.

1.8 Recursive Representation

The recursive form of the value function can be written as:

$$\rho V_t(a, y) = \max_c \left\{ u(c) + \mathbb{E}_t \left[\frac{dV_t(a, y)}{dt} \right] \right\}$$

Q1: What is continuation value?

For income state y_j, we have:

$$\rho V_t(a, y_j) = \max_c \left\{ u(c) + \left(r_t a + w_t y_j - c\right) \partial_a V_t(a, y_j) + \lambda_j \left[V_t(a, y_{-j}) - V_t(a, y_j) \right] + \partial_t V_t(a, y_j) \right\}$$

Resolving the maximization gives the first order condition (FOC):

$$u'\left(c_t(a, y_j)\right) = \partial_a V_t(a, y_j)$$

for all a and j. Define the savings policy function as

$$s_t(a, y_j) = r_t a + w_t y_j - c_t(a, y_j)$$

Q2: Where is the borrowing constraint $a_{i,t} \geq \underline{a}$ in the HJB?

The borrowing constraint appears in the **boundary condition** of the HJB:

- **State constraint boundary condition:**

$$\partial_a V_t(\underline{a}, y_j) \geq u'(r_t \underline{a} + w_t y_j)$$

- **Economic intuition:**
  The value of saving must be weakly larger than the value of consuming.
- **Heuristic derivation:**
  The first-order condition (FOC) continues to hold at the borrowing constraint,

$$u'(c_t(\underline{a}, y_j)) = \partial_a V_t(\underline{a}, y_j)$$

  but the borrowing constraint imposes

$$s_t(\underline{a}, y_j) = r_t \underline{a} + w_t y_j - c_t(\underline{a}, y_j) \geq 0$$

- **Advantage of continuous time:** The borrowing constraint is neatly incorporated as a boundary condition.

1.9 Summary

A solution to the household problem in recursive form consists of two functions, $V_t(a, y)$ and $c_t(a, y)$, such that:

$$\rho V_t(a, y_j) = u(c_t(a, y_j)) + \left(r_t a + w_t y_j - c_t(a, y_j)\right) \partial_a V_t(a, y_j) + \lambda_j \left[V_t(a, y_{-j}) - V_t(a, y_j)\right] + \partial_t V_t(a, y_j)$$

and

$$u'(c_t(a, y_j)) = \partial_a V_t(a, y_j)$$

subject to the HJB boundary condition:

$$\partial_a V_t(a, y_j) \geq u'(r_t a + w_t y_j)$$

To save space, we now use the savings policy function as shorthand:

$$s_t(a, y_j) \equiv r_t a + w_t y_j - c_t(a, y_j)$$

1.10 Kolmogorov Forward Equation

Proposition 1.1. *The evolution of the joint density $g_t(a, y)$ is governed by the **Kolmogorov Forward (KF) equation**:*

$$\partial_t g_t(a, y_j) = -\partial_a \left[(r_t a + w_t y_j - c_t(a, y_j))\, g_t(a, y_j) \right] - \lambda_j\, g_t(a, y_j) + \lambda_{-j}\, g_t(a, y_{-j}) \tag{2}$$

Proof: We consider, at some time t, the total mass of agents who are in income state y_j and whose assets lie in the interval $[a_1, a_2]$. Denote this total mass as $M_j(t; a_1, a_2)$:

$$M_j(t; a_1, a_2) = \int_{a_1}^{a_2} g_t(a, y_j)\, da$$

The rate of change of this total mass over time, $\frac{d}{dt} M_j$, consists of two components:

1. **Net flow across boundaries (Asset flow):** Agents cross the boundaries a_1 and a_2 due to saving or borrowing decisions.
2. **Net flow from switching (State jumps):** Agents enter or leave income state y_j due to income shocks.

For the first component, the net flow across boundaries is:

$$s_t(a_1, y_j) g_t(a_1, y_j) - s_t(a_2, y_j) g_t(a_2, y_j)$$

By the Fundamental Theorem of Calculus, we can write the difference at the boundaries as an integral of the derivative:

$$s_t(a_1, y_j) g_t(a_1, y_j) - s_t(a_2, y_j) g_t(a_2, y_j) = -\int_{a_1}^{a_2} \frac{\partial}{\partial a} \left(s_t(a, y_j) g_t(a, y_j) \right) da$$

For the second component, the net flow from switching is:

$$\int_{a_1}^{a_2} \left(\lambda_{-j} g_t(a, y_{-j}) - \lambda_j g_t(a, y_j) \right) da$$

Therefore,

$$\int_{a_1}^{a_2} \frac{\partial g_t(a, y_j)}{\partial t} da = \int_{a_1}^{a_2} \left[-\frac{\partial}{\partial a} (s_t(a, y_j) g_t(a, y_j)) - \lambda_j g_t(a, y_j) + \lambda_{-j} g_t(a, y_{-j}) \right] da$$

Since the interval $[a_1, a_2]$ is chosen arbitrarily, the integral equation above must hold for all such intervals. This implies that the integrand itself must be equal everywhere. Therefore, we can drop the integral and obtain a pointwise partial differential equation:

$$\frac{\partial g_t(a, y_j)}{\partial t} = -\frac{\partial}{\partial a}\left(s_t(a, y_j) g_t(a, y_j)\right) - \lambda_j g_t(a, y_j) + \lambda_{-j} g_t(a, y_{-j})$$

Finally, by substituting the drift term $s_t(a, y_j)$ back with its economic expression $r_t a + w_t y_j - c_t(a, y_j)$, we recover the equation as previously stated:

$$\partial_t g_t(a, y_j) = -\partial_a\left[(r_t a + w_t y_j - c_t(a, y_j)) g_t(a, y_j)\right] - \lambda_j g_t(a, y_j) + \lambda_{-j} g_t(a, y_{-j})$$

Q.E.D.

1.11 Competitive Equilibrium: Recursive Form

Definition 1.4. Given an initial joint density $g_0(a, y)$ and an exogenous path of total factor productivity (TFP) $\{A_t\}$, a **competitive equilibrium (in recursive form)** consists of functions

$$\{V_t(a, y),\ c_t(a, y),\ g_t(a, y)\} \quad \text{and} \quad \{Y_t,\ \ell_t,\ r_t,\ w_t\}$$

such that:

1. Households optimize,
2. Firms optimize,
3. Markets clear, and
4. The joint density $g_t(a, y)$ evolves consistently with household behavior.

1.12 Summary I

The HJB equation for the value function is:

$$\rho V_t(a, y_j) = u(c_t(a, y_j)) + s_t(a, y_j)\frac{\partial V_t(a, y_j)}{\partial a} + \lambda_j\left(V_t(a, y_{-j}) - V_t(a, y_j)\right) + \frac{\partial V_t(a, y_j)}{\partial t}$$

The first-order and envelope conditions are:

$$\frac{\partial V_t(a, y_j)}{\partial a} \geq u'\left(r_t a + A_t y_j\right)$$

$$u'(c_t(a, y_j)) = \frac{\partial V_t(a, y_j)}{\partial a}$$

The evolution of the density is given by:

$$\frac{\partial g_t(a, y_j)}{\partial t} = -\frac{\partial}{\partial a}\left[s_t(a, y_j) g_t(a, y_j)\right] - \lambda_j g_t(a, y_j) + \lambda_{-j} g_t(a, y_{-j})$$

1.13 Summary II

Bond Market Clearing Condition:

$$0 = \sum_j \int a \, g_t(a, y_j) \, da$$

> **Note**
>
> Here, we set the wage $w_t = A_t$ and omitted the goods market clearing condition by Walras' law.

Summary:
In continuous time, heterogeneous agent (HA) models reduce to two coupled partial differential equations (PDEs)! In mathematics, these are often called "Mean Field Games."

1.14 Stationary Competitive Equilibrium

Definition 1.5. When $A_t = A$, a *stationary competitive equilibrium* consists of functions $\{V(a, y),\ c(a, y),\ g(a, y)\}$ and $\{Y,\ \ell,\ r,\ w\}$ such that:

1. Households optimize,
2. Firms optimize,
3. Markets clear, and
4. The joint density $g(a, y)$ evolves in a way that is consistent with optimal household behavior.

- This is the natural extension of the "steady state" concept to heterogeneous agent economies.
- Macroeconomic aggregates are constant over time. The distribution $g(a, y)$ is also constant, but individual households continue to move as they receive idiosyncratic income shocks.
- The standard notion of "steady" means: if the economy starts in this state, it remains there.

1.15 Stationary Competitive Equilibrium Conditions

The stationary equilibrium is characterized by the following system of equations:

$$\begin{aligned}
\rho V(a, y_j) &= u(c(a, y_j)) + s(a, y_j)\frac{\partial V(a, y_j)}{\partial a} + \lambda_j \left[V(a, y_{-j}) - V(a, y_j)\right] \\
\frac{\partial V(a, y_j)}{\partial a} &\geq u'\left(ra + Ay_j\right) \\
u'(c(a, y_j)) &= \frac{\partial V(a, y_j)}{\partial a} \\
0 &= -\frac{\partial}{\partial a}\left[s(a, y_j)g(a, y_j)\right] - \lambda_j g(a, y_j) + \lambda_{-j} g(a, y_{-j}) \\
0 &= \sum_j \int a\, g(a, y_j)\, da
\end{aligned} \tag{3}$$

2 Solve the HJB Equation using Explicit Method

2.1 HJB Equation

We use a finite difference method to approximate the functions (v_1, v_2) at I discrete points along the asset grid, denoted a_i for $i = 1, \ldots, I$. The grid points are evenly spaced, with Δa representing the distance between each point. For simplicity, let $v_{i,j} \equiv v_j(a_i)$.

To approximate the derivative $v'_{i,j} = v'_j(a_i)$, we use either the forward or backward difference method:

$$v'_j(a_i) \approx \frac{v_{i+1,j} - v_{i,j}}{\Delta a} \equiv v'_{i,j,F} v'_j(a_i) \approx \frac{v_{i,j} - v_{i-1,j}}{\Delta a} \equiv v'_{i,j,B} \tag{4}$$

The finite difference approximation to Equation 3 becomes:

$$\rho v_{i,j} = u(c_{i,j}) + v'_{i,j}(Ay_j + ra_i - c_{i,j}) + \lambda_j(v_{i,-j} - v_{i,j}), \quad j = 1, 2 c_{i,j} = (u')^{-1}(v'_{i,j}) \tag{5}$$

where $v'_{i,j}$ is chosen as either the forward or backward difference.

There are two main complications:

1. **Difference Approximation Choice:** Deciding when to use the forward or backward difference matters for the numerical stability of the method.
2. **Nonlinearity:** The HJB equations are highly nonlinear, making the resulting system of Equation 5 nonlinear as well. This means we must use an iterative approach to solve them, rather than simply inverting a matrix.

2.2 Explicit Method

This method begins with an initial guess $v_j^0 = (v_{1,j}^0, ..., v_{I,j}^0)$ for each $j = 1, 2$, and then updates the values iteratively, so that for each iteration $n = 1, 2, ...$, the update rule is:

$$\frac{v_{i,j}^{n+1} - v_{i,j}^n}{\Delta} + \rho v_{i,j}^n = u(c_{i,j}^n) + (v_{i,j}^n)'(Ay_j + ra_i - c_{i,j}^n) + \lambda_j(v_{i,-j}^n - v_{i,j}^n) \tag{6}$$

Here, $c_{i,j}^n = (u')^{-1}\left[(v_{i,j}^n)'\right]$.

The parameter Δ is the time step size used in the explicit method.

> **Caution**
>
> It is important to note that the explicit method will only converge if the time step Δ is not too large; specifically, it must satisfy the so-called "Courant-Friedrichs-Lewy (CFL) condition" (see, for example, p.181 in Candler (1999)).

2.3 Upwind Scheme

As previously discussed, it is important to decide whether to use a forward or backward difference approximation for the derivative. The correct way to do this is by employing an **upwind scheme**.

The basic idea is:

- Use the forward difference approximation whenever the drift of the state variable (i.e., savings) is positive.
- Use the backward difference whenever it is negative.

Here, savings at iteration n are given by $s_{i,j}^n = Ay_j + ra_i - c_{i,j}^n$. In practice, proceed as follows:

First, compute savings using both the forward and backward difference approximations to the derivative $v'_{i,j,F}$ and $v'_{i,j,B}$:

$$s_{i,j,F} = Ay_j + ra_i - (u')^{-1}(v'_{i,j,F}), \quad s_{i,j,B} = Ay_j + ra_i - (u')^{-1}(v'_{i,j,B})$$

Here, for simplicity, we omit the superscript n.

Then, approximate the derivative $v'_{i,j}$ by:

$$v'_{i,j} = v'_{i,j,F} \cdot \mathbf{1}_{\{s_{i,j,F}>0\}} + v'_{i,j,B} \cdot \mathbf{1}_{\{s_{i,j,B}<0\}} + \bar{v}'_{i,j} \cdot \mathbf{1}_{\{s_{i,j,F}\leq 0\leq s_{i,j,B}\}} \tag{7}$$

where $\mathbf{1}_{\{.\}}$ denotes the indicator function.

Explanation of the last term:

- Since the value function v is concave in a, we have $v'_{i,j,F} < v'_{i,j,B}$ and therefore $s_{i,j,F} < s_{i,j,B}$. As a result, for some grid points i, it is possible that $s_{i,j,F} \leq 0 \leq s_{i,j,B}$.
- At these grid points, set savings to zero and thus set the derivative of the value function to $\bar{v}'_{i,j} = u'(Ay_j + ra_i)$.

Due to concavity, we do not need to worry about the case where both $s_{i,j,F} > 0$ and $s_{i,j,B} < 0$, because $s_{i,j,F} < s_{i,j,B}$ means this cannot happen.

However, in some cases (such as problems with non-convexities), the value function may not be concave. Then, it is possible for both $s_{i,j,F} > 0$ and $s_{i,j,B} < 0$ to occur. In practice, the following upwind scheme works well.

Define:

- The indicator for the problematic case where both $s_{i,j,F} > 0$ and $s_{i,j,B} < 0$:

$$\mathbf{1}_{i,j}^{\text{both}} := \mathbf{1}_{\{s_{i,j,B}\leq 0\leq s_{i,j,F}\}}$$

- The indicator for the non-problematic case:

$$\mathbf{1}_{i,j}^{\text{unique}} = \mathbf{1}_{\{s_{i,j,F}<0 \text{ and } s_{i,j,B}>0\}} + \mathbf{1}_{\{s_{i,j,F}>0 \text{ and } s_{i,j,B}<0\}}$$

Define the forward and backward Hamiltonians:

$$H_{i,j,F} := u(c_{i,j,F}) + v'_{i,j,F} s_{i,j,F}$$

and similarly for $H_{i,j,B}$.

Finally, the complete upwind scheme is:

$$\begin{aligned} v'_{i,j} = {} & v'_{i,j,F} \left(\mathbf{1}_{\{s_{i,j,F}>0\}} \mathbf{1}_{i,j}^{\text{unique}} + \mathbf{1}_{\{H_{i,j,F}\geq H_{i,j,B}\}} \mathbf{1}_{i,j}^{\text{both}} \right) \\ & + v'_{i,j,B} \left(\mathbf{1}_{\{s_{i,j,B}<0\}} \mathbf{1}_{i,j}^{\text{unique}} + \mathbf{1}_{\{H_{i,j,F}<H_{i,j,B}\}} \mathbf{1}_{i,j}^{\text{both}} \right) \\ & + \bar{v}'_{i,j} \mathbf{1}_{\{s_{i,j,F}\leq 0\leq s_{i,j,B}\}} \end{aligned}$$

Intuition:

- In the problematic case when both $s_{i,j,F} > 0$ and $s_{i,j,B} < 0$, the upwind scheme uses the direction that yields the larger value for the Hamiltonian (i.e., the greater gain in the value function), breaking the tie accordingly.

2.4 State Constraint

The state constraint is enforced by setting

$$v'_{1,j,B} = u'(Ay_j + ra_1), \quad \text{for } j = 1, 2$$

- According to Equation 7, the state constraint is applied whenever the forward difference approximation would result in negative savings, that is, when $s_{1,j,F} \leq 0$.
- Otherwise, if $s_{1,j,F} > 0$, the forward difference approximation $v'_{1,j,F}$ is used at the boundary, meaning that the value function does not 'see' the state constraint.
- At the upper end of the state space, the upwind scheme should ensure that a backward difference approximation is used.
- In practice, for better numerical stability, it can be helpful to impose a state constraint at the upper bound, $a \leq a_{\max}$, where $a_{\max}$ is the highest asset level used in the computation. This can be implemented by setting

  $$v'_{I,j,F} = u'(Ay_j + ra_I)$$

  where I is the index corresponding to $a_{\max}$.

2.5 Initial Guess

A natural initial guess for the value function is the value from "staying put":

$$v^0_{i,j} = \frac{u(Ay_j + ra_i)}{\rho}.$$

2.6 Algorithm Summary

To solve the HJB equations, proceed as follows:

Algorithm 2.1.

1. *Make an initial guess $v^0_{i,j}$ for $i = 1, \ldots, I$ and $j = 1, 2$.*
2. *For each iteration $n = 0, 1, 2, \ldots$, do:*
   1. *Compute $(v^n_{i,j})'$ using equations Equation 4 and Equation 7.*
   2. *Compute the consumption c^n via*

      $$c^n_{i,j} = (u')^{-1}\left[(v^n_{i,j})'\right]$$

   3. *Update the value function to v^{n+1} using equation Equation 6.*
   4. *If v^{n+1} is sufficiently close to v^n, stop; otherwise, repeat from step 2.1.*

3 Code for Solve the HJB Equation using Explicit Method

3.1 Preliminaries

We will use python with JAX to do it.

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

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

Let's first define the parameters.

```
sigma = 2.0
n_y = 2
rho = 0.05
r = 0.035
A = 1.0
y = jnp.array([0.1, 0.2])
lamb = jnp.array([1.5, 1])
```

3.2 Create Grids

```
I = 500
a_min, a_max = -0.1, 3.0
a = jnp.linspace(a_min, a_max, I)
da = (a_max - a_min) / (I - 1)
```

We can pack the elements into tuples:

```
params = (sigma, rho, r, A, y, lamb)
arrays = (a, y, lamb)
sizes = (I, n_y)
```

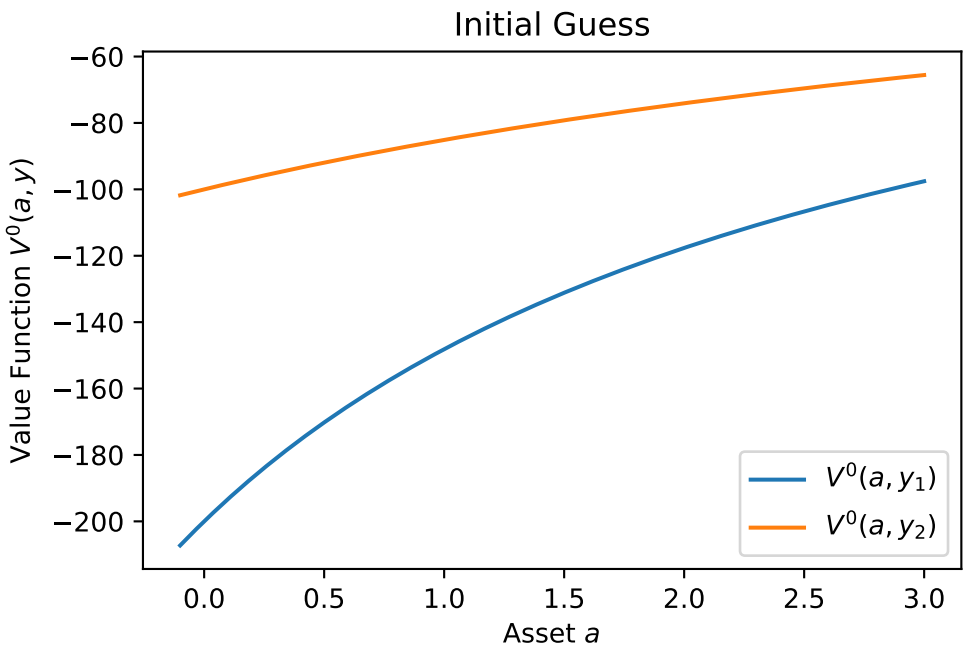
And define the utility function:

```
@jax.jit
def u(c, sigma=sigma):
    return c ** (1 - sigma) / (1 - sigma)


@jax.jit
def u_prime(c, sigma=sigma):
    return c ** (-sigma)


@jax.jit
def u_prime_inv(x, sigma=sigma):
    return x ** (-1 / sigma)
```

3.3 Initial Guess

We first define the initial guess for the value function:

```
V0 = jnp.zeros(sizes)
V0 = u(A * y[jnp.newaxis, :] + r * a[:, jnp.newaxis]) / rho


plt.plot(a, V0[:, 0], label="$V^0(a,y_1)$")
plt.plot(a, V0[:, 1], label="$V^0(a,y_2)$")
plt.title("Initial Guess")
plt.xlabel("Asset $a$")
plt.ylabel("Value Function $V^0(a,y)$")
plt.legend()
plt.show()
```



Then let's set the parameters for the solver.

```
maxit = 100_000
crit = 1e-11
# Delta is given by the CFL condition
Delta = 0.9 * da / jnp.max(y[1] + r * a)
print(f"Delta = {Delta:.4f}")
```

```
Delta = 0.0183
```

3.4 Finite Difference Method

Similar to previous lectures, we first write the scalar version of the finite difference method.

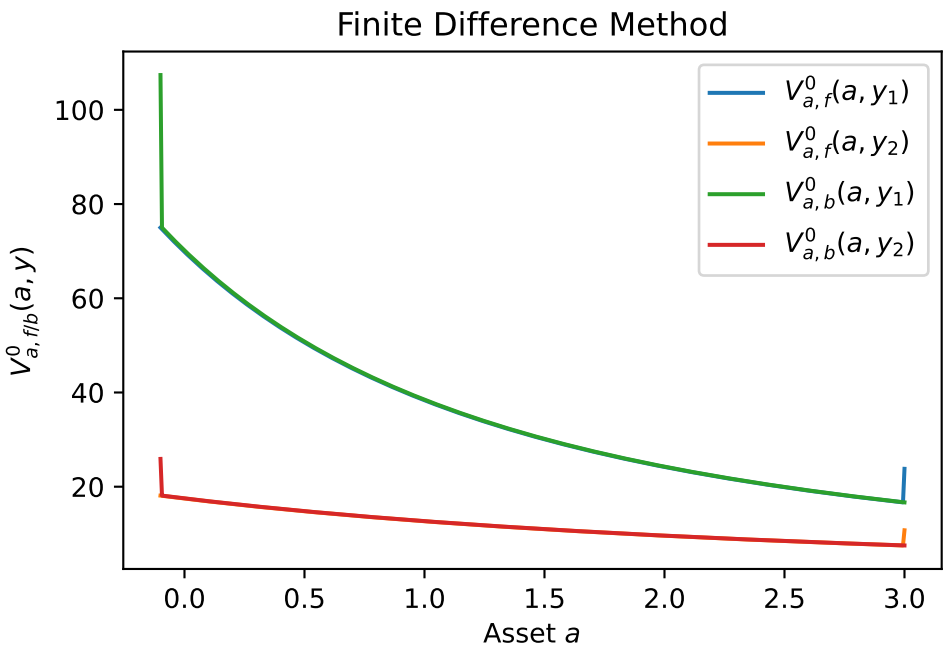
```
@jax.jit
def _FD_f(V, i_a, da, params, arrays, sizes):
    sigma, rho, r, A, y, lamb = params
    a, y, lamb = arrays
    I, n_y = V.shape
    return jax.lax.cond(
        i_a < I - 1,
        lambda _: (V[i_a + 1, :] - V[i_a, :]) / da,
```

```
        lambda _: u_prime(A * y + r * a[i_a]),
        operand=None,
    )


FD_f = jax.vmap(_FD_f, in_axes=(None, 0, None, None, None, None))


@jax.jit
def _FD_b(V, i_a, da, params, arrays, sizes):
    sigma, rho, r, A, y, lamb = params
    a, y, lamb = arrays
    I, n_y = V.shape
    return jax.lax.cond(
        i_a > 0,
        lambda _: (V[i_a, :] - V[i_a - 1, :]) / da,
        lambda _: u_prime(A * y + r * a[i_a]),
        operand=None,
    )


FD_b = jax.vmap(_FD_b, in_axes=(None, 0, None, None, None, None))
```

3.5 Test Finite Difference Method

Let's test the finite difference method.

```
a_indices = jnp.arange(I)
V0a_f = FD_f(V0, a_indices, da, params, arrays, sizes)
V0a_b = FD_b(V0, a_indices, da, params, arrays, sizes)
plt.plot(a, V0a_f[:, 0], label="$V^0_{a,f}(a,y_1)$")
plt.plot(a, V0a_f[:, 1], label="$V^0_{a,f}(a,y_2)$")
plt.plot(a, V0a_b[:, 0], label="$V^0_{a,b}(a,y_1)$")
plt.plot(a, V0a_b[:, 1], label="$V^0_{a,b}(a,y_2)$")
plt.title("Finite Difference Method")
plt.xlabel("Asset $a$")
plt.ylabel("$V^0_{a,f/b}(a,y)$")
plt.legend()
plt.show()
```



3.6 Upwind Scheme

Once we have the finite difference method, we can implement the upwind scheme.

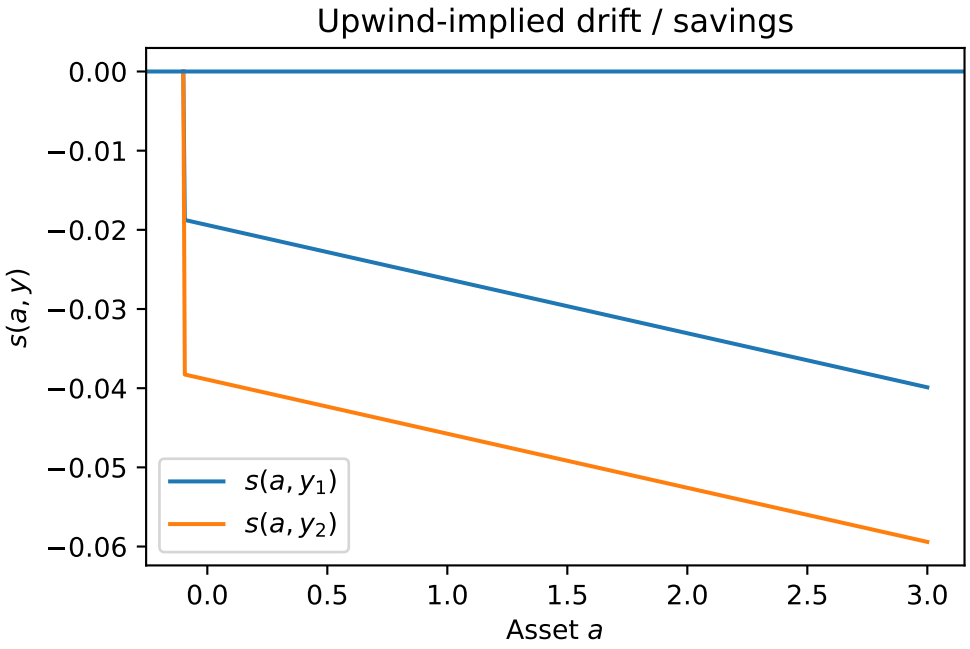
```
# Scalar (in i_a) upwind rule, vector-valued in y (shape (n_y,))
@jax.jit
def _upwind_i(V, Va_f, Va_b, i_a, params, arrays, sizes, eps=1e-12):
    sigma, rho, r, A, y, lamb = params
    a, _, _ = arrays
    I, n_y = sizes

    ai = a[i_a]
    cash = A * y + r * ai  # (n_y,)

    # --- candidates from forward/backward derivatives ---
    Va_f_i = Va_f[i_a, :]  # (n_y,)
    Va_b_i = Va_b[i_a, :]  # (n_y,)

    c_f = u_prime_inv(Va_f_i, sigma=sigma)
    s_f = cash - c_f

    c_b = u_prime_inv(Va_b_i, sigma=sigma)
```

```
    s_b = cash - c_b

    # --- "steady" / state-constraint candidate (s=0) ---
    Va_0 = u_prime(cash, sigma=sigma)  # u'(cash)

    # --- Hamiltonians for the rare "both" case ---
    H_f = u(c_f, sigma=sigma) + Va_f_i * s_f
    H_b = u(c_b, sigma=sigma) + Va_b_i * s_b

    # --- indicators (all shape (n_y,)) ---
    both = (s_f > 0) & (s_b < 0)  # non-concave / problematic case
    If = (s_f > 0) & (~both)  # unique forward
    Ib = (s_b < 0) & (~both)  # unique backward
    I0 = ~(If | Ib | both)  # steady region: s_f ≤ 0 ≤ s_b

    # --- choose Va via upwind + Hamiltonian tie-break ---
    Va_i = Va_f_i * If + Va_b_i * Ib + Va_0 * I0 + (jnp.where(H_f ≥ H_b, Va_f_i, Va_b_i)) * both

    # implied policy
    c_i = u_prime_inv(Va_i, sigma=sigma)
    s_i = cash - c_i

    return Va_i, c_i, s_i
```

3.7 Test Upwind Scheme

```
# Vectorize over asset index i_a
upwind = jax.vmap(_upwind_i, in_axes=(None, None, None, 0, None, None, None))


# --- run once to test ---
a_indices = jnp.arange(I)
Va_f = FD_f(V0, a_indices, da, params, arrays, sizes)  # (I, n_y)
Va_b = FD_b(V0, a_indices, da, params, arrays, sizes)  # (I, n_y)

Va_u, c_u, s_u = upwind(V0, Va_f, Va_b, a_indices, params, arrays, sizes)

# quick plots
plt.plot(a, s_u[:, 0], label="$s(a,y_1)$")
plt.plot(a, s_u[:, 1], label="$s(a,y_2)$")
```

```
plt.axhline(0.0)
plt.title("Upwind-implied drift / savings")
plt.xlabel("Asset $a$")
plt.ylabel("$s(a,y)$")
plt.legend()
plt.show()
```



3.8 Explicit Iteration

```
@jax.jit
def jump_term(V, params, arrays, sizes):
    """
    CTMC jump term: (Lambda V)(a, y_j) = lambda_j [V(a, y_-j) - V(a, y_j)]
    """
    sigma, rho, r, A, y, lamb = params

    # two-state special case (n_y=2)
    jump0 = lamb[0] * (V[:, 1] - V[:, 0])
    jump1 = lamb[1] * (V[:, 0] - V[:, 1])
    return jnp.array([jump0, jump1]).T
```

```
@jax.jit
def hjb_residual(V, Va_u, c_u, s_u, params, arrays, sizes):
    """
    residual = u(c) + Va*s + (Lambda V) - rho*V
    """
    sigma, rho, r, A, y, lamb = params
    jmp = jump_term(V, params, arrays, sizes)
    return u(c_u) + Va_u * s_u + jmp - rho * V


@jax.jit
def explicit_step(V, Delta, params, arrays, sizes):
    """
    One explicit time-iteration step:
      1) finite differences
      2) upwind → Va_u, c_u, s_u
      3) residual
      4) V_new = V + Delta * residual
    """
    a, y, lamb = arrays
    a_indices = jnp.arange(jnp.size(a))

    Va_f = FD_f(V, a_indices, da, params, arrays, sizes)
    Va_b = FD_b(V, a_indices, da, params, arrays, sizes)
    Va_u, c_u, s_u = upwind(V, Va_f, Va_b, a_indices, params, arrays, sizes)

    res = hjb_residual(V, Va_u, c_u, s_u, params, arrays, sizes)
    V_new = V + Delta * res
    return V_new, res, Va_u, c_u, s_u
```

3.9 Solve the HJB Equation

```
#  we use static_argnums to tell the compiler that the maximum number of iterations is fixed so tha
@jax.jit(static_argnums=(3,))
def solve_hjb(V_init, Delta, crit, maxit, params, arrays, sizes):
    """
    Solve stationary HJB by explicit time iteration
    """
```

```
    I, n_y = V_init.shape
    dist0 = jnp.zeros((maxit,), dtype=jnp.float64)
    Va0, c0, s0 = jnp.zeros((I, n_y)), jnp.zeros((I, n_y)), jnp.zeros((I, n_y))
    carry0 = (0, V_init, dist0, jnp.array(jnp.inf), Va0, c0, s0)

    def cond_fun(carry):
        n, V, dist, err, Va_u, c_u, s_u = carry
        return jnp.logical_and(n < maxit, err > crit)

    def body_fun(carry):
        n, V, dist, err, Va_u, c_u, s_u = carry
        V_new, res, Va_u_new, c_u_new, s_u_new = explicit_step(V, Delta, params, arrays, sizes)
        err_new = jnp.max(jnp.abs(res))
        dist_new = dist.at[n].set(err_new)
        return (n + 1, V_new, dist_new, err_new, Va_u_new, c_u_new, s_u_new)

    carryN = jax.lax.while_loop(cond_fun, body_fun, carry0)

    n, V, dist, err, Va_u, c_u, s_u = carryN
    return V, Va_u, c_u, s_u, dist, n
```

3.10 Test Solver

```
# warmup to get rid of the compile time
V_star, Va_star, c_star, s_star, dist, n_iter = solve_hjb(V0, Delta, crit, maxit, params, arrays, s
V_star.block_until_ready()

start_time = time.time()
V_star, Va_star, c_star, s_star, dist, n_iter = solve_hjb(V0, Delta, crit, maxit, params, arrays, s
V_star.block_until_ready()
explicit_time = time.time() - start_time

matlab_time = 4.414521
print(f"Time taken to solve for value function: {explicit_time:.4f} seconds")
print(f"Jax Speedup: {matlab_time / explicit_time:.2f}x")
print(f"Iterations: {int(n_iter)}, Residual: {dist[n_iter - 1]:.3e}")
```

```
Time taken to solve for value function: 1.1383 seconds
Jax Speedup: 3.88x
Iterations: 27216, Residual: 9.997e-12
```

- The code is so fast that the matlab version (provided by Achdou et al. (2022)) takes 4.414521 seconds to solve the stationary HJB equation under the same conditions.
- We can see that one advantage is that we can write the code in a more "dumb" way,[2] but still get better performance.

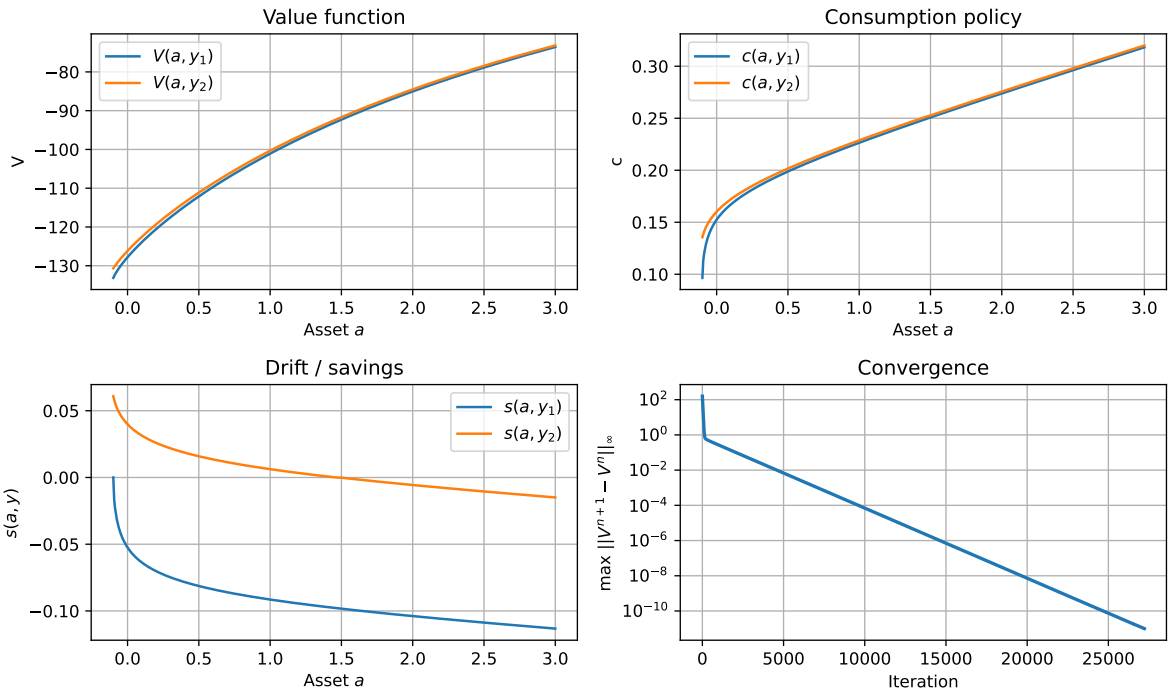
3.11 Visualize the Solution

```
n_it = int(n_iter)
dist_used = jnp.array(dist[:n_it])

V_np = jnp.array(V_star)
c_np = jnp.array(c_star)
s_np = jnp.array(s_star)
Va_np = jnp.array(Va_star)
a_np = jnp.array(a)

fig, axs = plt.subplots(2, 2, figsize=(10, 6))

# 1) Value function
axs[0, 0].plot(a_np, V_np[:, 0], label="$V(a,y_1)$")
axs[0, 0].plot(a_np, V_np[:, 1], label="$V(a,y_2)$")
axs[0, 0].set_title("Value function")
axs[0, 0].set_ylabel("V")
axs[0, 0].set_xlabel("Asset $a$")
axs[0, 0].grid(True)
axs[0, 0].legend()

# 2) Consumption policy
axs[0, 1].plot(a_np, c_np[:, 0], label="$c(a,y_1)$")
axs[0, 1].plot(a_np, c_np[:, 1], label="$c(a,y_2)$")
axs[0, 1].set_title("Consumption policy")
axs[0, 1].set_ylabel("c")
axs[0, 1].set_xlabel("Asset $a$")
axs[0, 1].grid(True)
axs[0, 1].legend()

# 3) Drift / savings
axs[1, 0].plot(a_np, s_np[:, 0], label="$s(a,y_1)$")
axs[1, 0].plot(a_np, s_np[:, 1], label="$s(a,y_2)$")
```

[2] Less vectorized and more explicit.

```
axs[1, 0].set_title("Drift / savings")
axs[1, 0].set_xlabel("Asset $a$")
axs[1, 0].set_ylabel("$s(a,y)$")
axs[1, 0].grid(True)
axs[1, 0].legend()

# 4) Convergence (max residual)
axs[1, 1].plot(jnp.arange(1, n_it + 1), dist_used, linewidth=2)
axs[1, 1].set_title("Convergence")
axs[1, 1].set_xlabel("Iteration")
axs[1, 1].set_ylabel("max $||V^{n+1} - V^n||_\infty$")
axs[1, 1].set_yscale("log")
axs[1, 1].grid(True)

plt.tight_layout()
plt.show()
```

```
<>:43: SyntaxWarning: invalid escape sequence '\i'
<>:43: SyntaxWarning: invalid escape sequence '\i'
/var/folders/xs/q8dt4q7926×0qg5_d_1_29br0000gn/T/ipykernel_96641/1045662454.py:43: SyntaxWarning: i
  axs[1, 1].set_ylabel("max $||V^{n+1} - V^n||_\infty$")
```



4 Solve the HJB Equation using Implicit Method

4.1 Implicit Method

Relative to the explicit scheme, the implicit method differs in how v^n is updated. In particular, v^{n+1} is now implicitly defined by the equation:

$$\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta} + \rho v_{i,j}^{n+1} = u(c_{i,j}^{n}) + (v_{i,j}^{n+1})'(Ay_j + ra_i - c_{i,j}^{n}) + \lambda_j(v_{i,-j}^{n+1} - v_{i,j}^{n+1})$$

Note the $n+1$ superscripts on the right-hand side of the equation[3]. The main advantage of the implicit scheme is that the step size Δ can be arbitrarily large.

4.2 Upwind Scheme

As with the explicit method, we apply an **upwind scheme** to the implicit method.

The key idea is to use the forward difference approximation when the drift of the state variable is positive, and the backward difference when the drift is negative.

The following finite difference approximation to Equation 5 is used:

$$\begin{aligned}\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta} + \rho v_{i,j}^{n+1} = u(c_{i,j}^{n}) \\ + (v_{i,j,F}^{n+1})' \left[Ay_j + ra_i - c_{i,j,F}^{n}\right]^{+} \\ + (v_{i,j,B}^{n+1})' \left[Ay_j + ra_i - c_{i,j,B}^{n}\right]^{-} \\ + \lambda_j \left[v_{i,-j}^{n+1} - v_{i,j}^{n+1}\right]\end{aligned} \tag{8}$$

where $c_{i,j}^{n} = (u')^{-1}[(v_{i,j}^{n})']$ and $(v_{i,j}^{n})'$ is given by Equation 7. [4]

Equation 8 forms a system of $2 \times I$ linear equations. We can rewrite this in matrix notation by introducing

- $s_{i,j,F}^{n} = Ay_j + ra_i - c_{i,j,F}^{n}$

[3] Strictly speaking, the present method is a "semi-implicit method." A fully implicit method would feature $n+1$ superscripts also on $c_{i,j}$. Such a fully implicit scheme can be solved using a Newton method, which ends up looking very similar to the iterative scheme outlined here.

[4] For any real number x, the notation x^{+} means the positive part, $x^{+} = \max\{x, 0\}$, and x^{-} denotes the negative part, $x^{-} = \min\{x, 0\}$.

- $s^n_{i,j,B} = Ay_j + ra_i - c^n_{i,j,B}$

and by recognizing that the finite difference derivatives (see Equation 4) lead to

$$\frac{v^{n+1}_{i,j} - v^n_{i,j}}{\Delta} + \rho v^{n+1}_{i,j} = u(c^n_{i,j}) + \frac{v^{n+1}_{i+1,j} - v^{n+1}_{i,j}}{\Delta a}(s^n_{i,j,F})^+ + \frac{v^{n+1}_{i,j} - v^{n+1}_{i-1,j}}{\Delta a}(s^n_{i,j,B})^- + \lambda_j \left[v^{n+1}_{i,-j} - v^{n+1}_{i,j} \right]$$

Collecting like terms in $v^{n+1}_{i-1,j}$, $v^{n+1}_{i,j}$, $v^{n+1}_{i+1,j}$, and $v^{n+1}_{i,-j}$, we obtain

$$\frac{v^{n+1}_{i,j} - v^n_{i,j}}{\Delta} + \rho v^{n+1}_{i,j} = u(c^n_{i,j}) + x_{i,j} v^{n+1}_{i-1,j} + y_{i,j} v^{n+1}_{i,j} + z_{i,j} v^{n+1}_{i+1,j} + \lambda_j v^{n+1}_{i,-j}$$

---

where

$$\begin{aligned} x_{i,j} &= -\frac{(s^n_{i,j,B})^-}{\Delta a} \\ y_{i,j} &= -\frac{(s^n_{i,j,F})^+}{\Delta a} + \frac{(s^n_{i,j,B})^-}{\Delta a} - \lambda_j \\ z_{i,j} &= \frac{(s^n_{i,j,F})^+}{\Delta a} \end{aligned}$$

> **Note**
>
> We have $x_{1,j} = z_{I,j} = 0$ for $j = 1, 2$, so $v^{n+1}_{0,j}$ and $v^{n+1}_{I+1,j}$ are never used (due to boundary conditions).

This system of $2 \times I$ equations can be written compactly in matrix notation as

$$\frac{1}{\Delta}(v^{n+1} - v^n) + \rho v^{n+1} = u^n + \mathbf{A}^n v^{n+1}$$

---

where

$$\mathbf{A}^n = \begin{bmatrix} y_{1,1} & z_{1,1} & 0 & \cdots & 0 & \lambda_1 & 0 & 0 & \cdots & 0 \\ x_{2,1} & y_{2,1} & z_{2,1} & 0 & \cdots & 0 & \lambda_1 & 0 & 0 & \cdots \\ 0 & x_{3,1} & y_{3,1} & z_{3,1} & 0 & \cdots & 0 & \lambda_1 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & x_{I,1} & y_{I,1} & 0 & 0 & 0 & \lambda_1 & \\ \lambda_2 & 0 & 0 & 0 & 0 & y_{1,2} & z_{1,2} & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & 0 & x_{2,2} & y_{2,2} & z_{2,2} & 0 & 0 \\ 0 & 0 & \lambda_2 & 0 & 0 & 0 & x_{3,2} & y_{3,2} & z_{3,2} & 0 \\ \vdots & & & & & & & & & \\ 0 & \cdots & \cdots & 0 & \lambda_2 & 0 & \cdots & 0 & x_{I,2} & y_{I,2} \end{bmatrix}, \qquad u^n = \begin{bmatrix} u(c^n_{1,1}) \\ \vdots \\ u(c^n_{I,1}) \\ u(c^n_{1,2}) \\ \vdots \\ u(c^n_{I,2}) \end{bmatrix}$$

This can be rearranged into a sparse linear system of the form:

$$\mathbf{B}^n v^{n+1} = b^n \qquad \text{where} \qquad \mathbf{B}^n = \left(\frac{1}{\Delta} + \rho\right)\mathbf{I} - \mathbf{A}^n \quad \text{and} \quad b^n = u^n + \frac{1}{\Delta}v^n \tag{9}$$

Equation 9 can be efficiently solved using sparse matrix routines.

If the step size goes to infinity ($(1/\Delta) = 0$), then the linear system reduces to:

$$\rho v^{n+1} = u^n + \mathbf{A}^n v^{n+1} \tag{10}$$

This demonstrates that Equation 10 is simply the discretized HJB equation in matrix form.

The matrix $\mathbf{A}^n$ encodes the evolution of the stochastic process (a_t, z_t): its structure captures the transition intensities of the Poisson approximation.

Notably, $\mathbf{A}^n$ has all rows summing to zero, non-positive diagonal entries, and non-negative off-diagonal entries, making it a valid Poisson transition/intensity matrix. [5]

4.3 Summary of the Algorithm

The algorithm below proceeds exactly as in the explicit method, except for the implicit update equation:

Algorithm 4.1.

1. *Compute* $(v^n_{i,j})'$ *using Equation 4 and Equation 7.*
2. *Compute* c^n *from* $c^n_{i,j} = (u')^{-1}[(v^n_{i,j})']$.
3. *Solve for* v^{n+1} *using Equation 9.*
4. *If* v^{n+1} *is close enough to* v^n*, stop. Otherwise, repeat from step 1.*

[5] This property will be applied in the next section for the Kolmogorov Forward equation.

5 Code for Solve the HJB Equation using Implicit Method

5.1 Initial Guess

We adopt the same code structure as the explicit method, but with the implicit update equation.

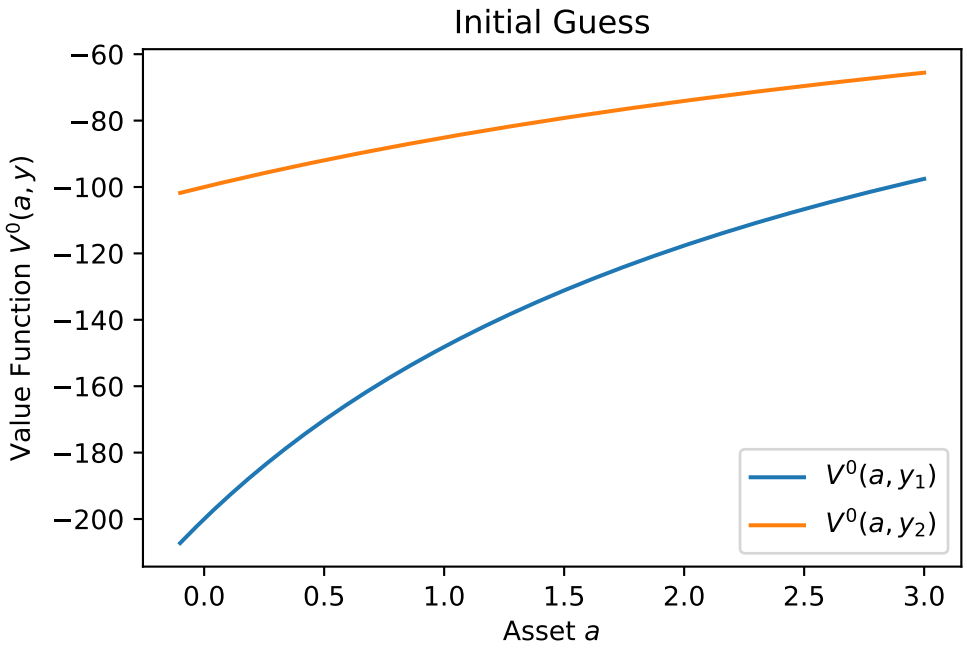
We start with the same initial guess as the explicit method:

$$v_{i,j}^0 = \frac{u(Ay_j + ra_i)}{\rho}.$$

```
V0 = u(A * y[jnp.newaxis, :] + r * a[:, jnp.newaxis]) / rho
Delta = 1000
```

We visualize the initial guess:

```
plt.plot(a, V0[:, 0], label="$V^0(a,y_1)$")
plt.plot(a, V0[:, 1], label="$V^0(a,y_2)$")
plt.title("Initial Guess")
plt.xlabel("Asset $a$")
plt.ylabel("Value Function $V^0(a,y)$")
plt.legend()
plt.show()
```



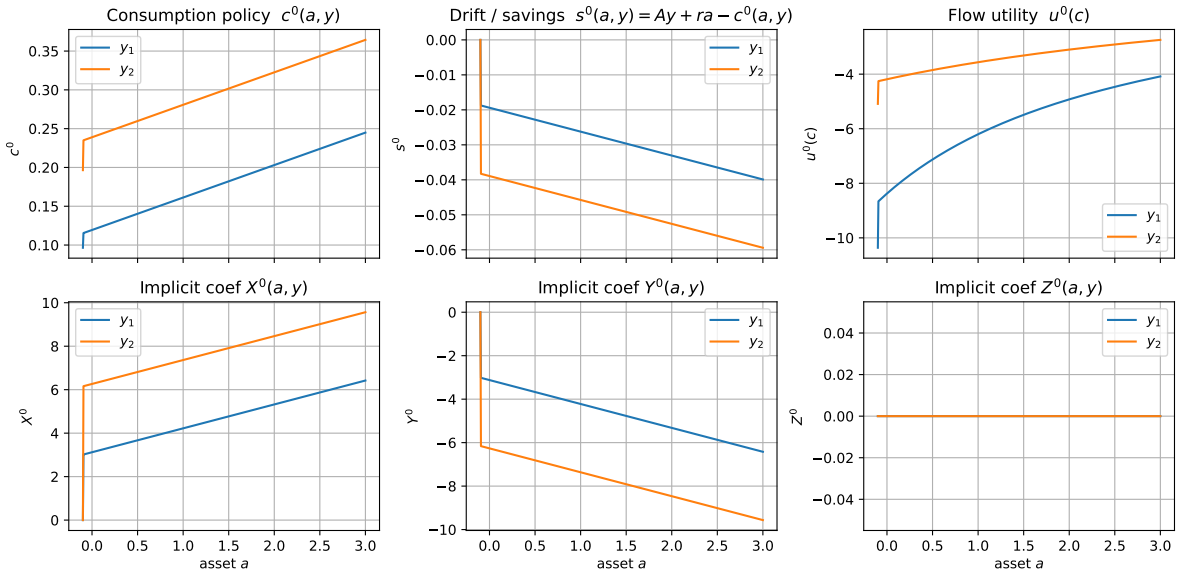
5.2 Get the Coefficients

```
@jax.jit
def get_coef(V, da, params, arrays, sizes):
    sigma, rho, r, A, y, lamb = params
    a, y, lamb = arrays

    I, n_y = V.shape
    a_indices = jnp.arange(I)
    dVf = FD_f(V, a_indices, da, params, arrays, sizes)  # (I,2)
    dVb = FD_b(V, a_indices, da, params, arrays, sizes)  # (I,2)

    # cash-on-hand at each (a_i, y_j)
    cash = A * y[jnp.newaxis, :] + r * a[:, jnp.newaxis]  # (I,2)

    cf, cb = u_prime_inv(dVf), u_prime_inv(dVb)
    ssf, ssb = cash - cf, cash - cb

    # steady (s=0) candidate
    c0 = cash
    dV0 = u_prime(c0)

    # upwind choice for marginal value → final c^n
    If = ssf > 0
    Ib = ssb < 0
    I0 = ~(If | Ib)

    dV_up = dVf * If + dVb * Ib + dV0 * I0
    c = u_prime_inv(dV_up)
    u_flow = u(c)

    ssb_neg = jnp.minimum(ssb, 0.0)
    ssf_pos = jnp.maximum(ssf, 0.0)

    X = -ssb_neg / da
    Y = -ssf_pos / da + ssb_neg / da
    Z = ssf_pos / da

    return u_flow, X, Y, Z, c
```

5.3 Visualize the Coefficients

It's useful to visualize the coefficients.

```
u_flow, X, Y, Z, c = get_coef(V0, da, params, arrays, sizes)
cash = A * y[jnp.newaxis, :] + r * a[:, jnp.newaxis]
s = cash - c
fig, axs = plt.subplots(2, 3, figsize=(12, 6), sharex=True)


def plot_two_states(ax, x, title, ylabel):
    ax.plot(a_np, x[:, 0], label=r"$y_1$")
    ax.plot(a_np, x[:, 1], label=r"$y_2$")
    ax.set_title(title)
    ax.set_ylabel(ylabel)
    ax.grid(True)
    ax.legend()


plot_two_states(axs[0, 0], c, "Consumption policy  $c^0(a,y)$", "$c^0$  ")
plot_two_states(axs[0, 1], s, "Drift / savings  $s^0(a,y)=A y + r a - c^0(a,y)$", "$s^0$")
plot_two_states(axs[0, 2], u_flow, "Flow utility  $u^0(c)$", "$u^0(c)$")

plot_two_states(axs[1, 0], X, "Implicit coef $X^0(a,y)$", "$X^0$")
plot_two_states(axs[1, 1], Y, "Implicit coef $Y^0(a,y)$", "$Y^0$")
plot_two_states(axs[1, 2], Z, "Implicit coef $Z^0(a,y)$", "$Z^0$")

for ax in axs[1, :]:
    ax.set_xlabel("asset $a$")

plt.tight_layout()
plt.show()
```



5.4 Solve the Sparse Linear System

According to the form of $\mathbf{A}$, it's easy to see that it's a sparse matrix.

The best practice is to construct the sparse matrix using JAX.

Unfortunately, doing so is not so straightforward as in matlab. And according to the documentation[6]

> The submodule is no longer being actively developed, but the team will continue supporting existing features as best we can.[7]

This forces us to use a different approach to solve Equation 9.

Note that we need to solve

$$\mathbf{B}^n v^{n+1} = b^n$$

The real issue lies in the size of the matrix $\mathbf{B}^n$, which is $(2 \times I)^2 = 4I^2$.

5.5 Solve Equation 9 using Iterative Method

Instead of explicitly forming the $(2I)\times(2I)$ matrix $\mathbf{B}^n$, we can solve the linear system using an **iterative (Krylov) method**.

Key idea: matrix-free linear solver

[6]Unfortunately, there's no sparse matrix primitives in XLA.
[7]Reference: jax.experimental.sparse

The only operation required by many iterative solvers is the ability to compute the **matrix–vector product**

$$x \mapsto \mathbf{B}^n x.$$

Hence we do **not** need to store $\mathbf{B}^n$ explicitly (dense or sparse). We only need a function that implements the linear operator $\mathbf{B}^n(\cdot)$.

This is often called a **matrix-free** approach.

5.6 Why is this possible?

Consider solving

$$\mathbf{B}^n v^{n+1} = b^n, \qquad \mathbf{B}^n \in \mathbb{R}^{N\times N}, \quad N = 2I.$$

Direct methods (e.g., Gaussian elimination) require forming and factorizing $\mathbf{B}^n$, with computational cost roughly $O(N^3)$ for dense matrices.

However, in our application, $\mathbf{B}^n$ comes from a **finite-difference discretization** of a PDE/CTMC generator. Therefore:

- Each grid point interacts only with its neighbors $(i-1, i, i+1)$ in assets, and
- Each income state interacts with the other state through **switching**.

So $\mathbf{B}^n$ is extremely **structured and sparse**, and the action $\mathbf{B}^n x$ can be computed in $O(N)$ time.

5.7 Mathematical intuition: fixed-point

Rewrite the linear system as a root-finding problem:

$$F(v) := \mathbf{B}^n v - b^n = 0.$$

Given an initial guess $v^{(0)}$, an iterative solver generates a sequence $\{v^{(k)}\}$ such that the **residual**

$$r^{(k)} := b^n - \mathbf{B}^n v^{(k)}$$

converges to zero in norm:

$$\|r^{(k)}\| \to 0.$$

The practical stopping rule is typically

$$\frac{\|r^{(k)}\|}{\|b^n\|} \le \varepsilon,$$

for some tolerance $\varepsilon > 0$.

5.8 Krylov subspace methods

A widely used family of iterative solvers is the **Krylov subspace methods**.
The simplest way to motivate them is:

Starting from $v^{(0)}$, define $r^{(0)} = b^n - \mathbf{B}^n v^{(0)}$.
Then search for corrections δv in the **Krylov subspace**

$$\mathscr{K}_k(\mathbf{B}^n, r^{(0)}) := \operatorname{span}\left\{r^{(0)},\ \mathbf{B}^n r^{(0)},\ (\mathbf{B}^n)^2 r^{(0)},\ \dots,\ (\mathbf{B}^n)^{k-1} r^{(0)}\right\}.$$

That is, we approximate

$$v^{(k)} = v^{(0)} + \delta v^{(k)}, \qquad \delta v^{(k)} \in \mathscr{K}_k(\mathbf{B}^n, r^{(0)}).$$

GMRES (Generalized Minimal Residual) chooses $\delta v^{(k)}$ to minimize the residual norm:

$$v^{(k)} = \arg \min_{v \in v^{(0)} + \mathscr{K}_k(\mathbf{B}^n, r^{(0)})} \left\| b^n - \mathbf{B}^n v \right\|.$$

Interpretation: GMRES constructs increasingly rich approximations using only repeated applications of $\mathbf{B}^n$ to vectors, and at each step chooses the best approximation in that span.

5.9 Why this is attractive for our HJB discretization

- **No need to construct sparse matrices:** only implement $\mathbf{B}^n x$ (a local stencil).
- **Computational cost scales linearly:** each matvec uses neighbor operations, so $O(N) = O(I)$.
- **Naturally compatible with JAX:** the matrix-free matvec is just array operations and can be `jit`-compiled.

In short, the implicit scheme reduces to repeatedly applying a **discrete linear operator** (a finite-difference stencil + switching) and letting an iterative solver drive the residual to zero.

5.10 Code

So the next step is to implement the `matvec` function for $\mathbf{B}^n$.

```
@jax.jit
def matvec_B(x, X, Y, Z, params, arrays, sizes,Delta):
    """
    Matrix-free y = B x, B = (rho + 1/Delta)I - A
    x is stacked as [x(:,0); x(:,1)], shape (2I,)
    """
```

```
    sigma, rho, r, A, y, lamb = params

    I = X.shape[0]
    x0 = x[:I]
    x1 = x[I:]

    coef = rho + 1.0 / Delta

    # diag of B: (rho+1/Delta) - (Y - lamb) = (rho+1/Delta) - Y + lamb
    d0 = coef - Y[:, 0] + lamb[0]
    d1 = coef - Y[:, 1] + lamb[1]

    # start with diagonal + switching terms
    y0 = d0 * x0 - lamb[0] * x1
    y1 = d1 * x1 - lamb[1] * x0

    # lower: -X[i]*x[i-1] for i=1..I-1
    y0 = y0.at[1:].add((-X[1:, 0]) * x0[:-1])
    y1 = y1.at[1:].add((-X[1:, 1]) * x1[:-1])

    # upper: -Z[i]*x[i+1] for i=0..I-2
    y0 = y0.at[:-1].add((-Z[:-1, 0]) * x0[1:])
    y1 = y1.at[:-1].add((-Z[:-1, 1]) * x1[1:])

    return jnp.concatenate([y0, y1])
```

5.11 Solve the Sparse Linear System

It's lucky that we can then directly use the `jax.scipy.sparse.linalg.gmres` function to solve the sparse linear system.

```
from jax.scipy.sparse.linalg import gmres


@jax.jit
def solve_gmres(V, da, params, arrays, sizes, Delta, tol=1e-10, maxiter=50):
    """
    Given V^n, build coefficients and solve B^n v^{n+1} = b^n using GMRES.
    Returns V^{n+1}, along with (c^n) and a residual norm for diagnostics.
    """
    sigma, rho, r, A, y, lamb = params
```

```
    I, n_y = V.shape

    u_flow, X, Y, Z, c = get_coef(V, da, params, arrays, sizes)

    v = jnp.concatenate([V[:, 0], V[:, 1]])
    b = jnp.concatenate([u_flow[:, 0], u_flow[:, 1]]) + v / Delta

    # linear operator A(x) = Bx
    def A_mv(x):
        return matvec_B(x, X, Y, Z, params, arrays, sizes, Delta)

    v_new, info = gmres(
        A_mv,
        b,
        x0=v,  # warm start helps
        tol=tol,
        maxiter=maxiter,
    )

    # reshape back
    V_new = jnp.stack([v_new[:I], v_new[I:]], axis=1)

    # Since JAX gmres currently returns info=None (placeholder), we compute residual ourselves
    res = A_mv(v_new) - b
    res_norm = jnp.linalg.norm(res) / (jnp.linalg.norm(b) + 1e-32)

    return V_new, c, res_norm
```

5.12 Solve the HJB Equation

Now we can implement the whole alg. 4.1.

```
@jax.jit(static_argnums=(3,))
def solve_hjb_implicit_gmres(V_init, Delta, crit, maxit, params, arrays, sizes, tol=1e-12, maxiter=
    I, n_y = sizes

    res0 = jnp.zeros((maxit,), dtype=jnp.float64)
    dist0 = jnp.zeros((maxit,), dtype=jnp.float64)
    carry0 = (0, V_init, dist0, res0, jnp.array(jnp.inf))

    def cond_fun(carry):
```

```
        n, V, dist, res, err = carry
        return jnp.logical_and(n < maxit, err > crit)

    def body_fun(carry):
        n, V, dist, res, err = carry

        V_new, c, res_norm = solve_gmres(V, da, params, arrays, sizes, Delta, tol=tol, maxiter=maxi

        # outer convergence metric: sup norm change in V
        err_new = jnp.max(jnp.abs(V_new - V))
        dist_new = dist.at[n].set(err_new)
        res_new = res.at[n].set(res_norm)
        return (n + 1, V_new, dist_new, res_new, err_new)

    n, V_star, dist, res, err = jax.lax.while_loop(cond_fun, body_fun, carry0)
    return V_star, dist, res, n, err
```

5.13 Test Solver

Now let's solve the HJB equation using the implicit method!

```
crit = 1e-11
Delta = 1
maxit = 1000
# warmup compile
V_imp, dist, res, n_iter, err = solve_hjb_implicit_gmres(V0, Delta, crit, maxit, params, arrays, si
V_imp.block_until_ready()

start = time.time()
V_imp, dist, res, n_iter, err = solve_hjb_implicit_gmres(V0, Delta, crit, maxit, params, arrays, si
V_imp.block_until_ready()

implicit_time = time.time() - start
print(f"Time taken to solve for value function: {implicit_time:.4f} seconds, iters: {int(n_iter)},

_, _, _, _, c_imp = get_coef(V_imp, da, params, arrays, sizes)
cash = A * y[jnp.newaxis, :] + r * a[:, jnp.newaxis]
s_imp = cash - c_imp
```

```
Time taken to solve for value function: 0.1578 seconds, iters: 466, err: 0.000e+00
```

It's crazy that the code is so fast![8]

```
matlab_imp = 0.339512
print(f"Implicit Speedup: {explicit_time / implicit_time:.2f}x")
print(f"Matlab Speedup (Explicit): {matlab_time / implicit_time:.2f}x")
print(f"Matlab Speedup (Implicit): {matlab_imp / implicit_time:.2f}x")
print(
    f"Check for implicit and explicit methods: relative error V: {jnp.max(jnp.abs(V_star - V_imp))/
)
```

```
Implicit Speedup: 7.21x
Matlab Speedup (Explicit): 27.97x
Matlab Speedup (Implicit): 2.15x
Check for implicit and explicit methods: relative error V: 1.311e-11 c: 2.370e-
13 s: 6.698e-13
```

> **Warning**
>
> **Numerical warning: large-Δ implicit steps and boundary outliers**[9]: When using the semi-implicit scheme with a **large time step** Δ , the linear system
>
> $$B(\Delta)v^{n+1} = b(\Delta), \qquad B(\Delta) = \left(\rho + \frac{1}{\Delta}\right) I - A$$
>
> can become **harder to solve accurately**. With **unpreconditioned GMRES** and/or **insufficient Krylov iterations**, the solver may stop with a non-negligible residual. In practice, this residual often concentrates near the **boundary rows** of the discretization, showing up as a small **outlier / spike** in the value function close to the borrowing constraint.

5.14 Visualize the Solution

```
n_it = int(n_iter)
dist_used = jnp.array(dist[:n_it])
res_used = jnp.array(res[:n_it])
```

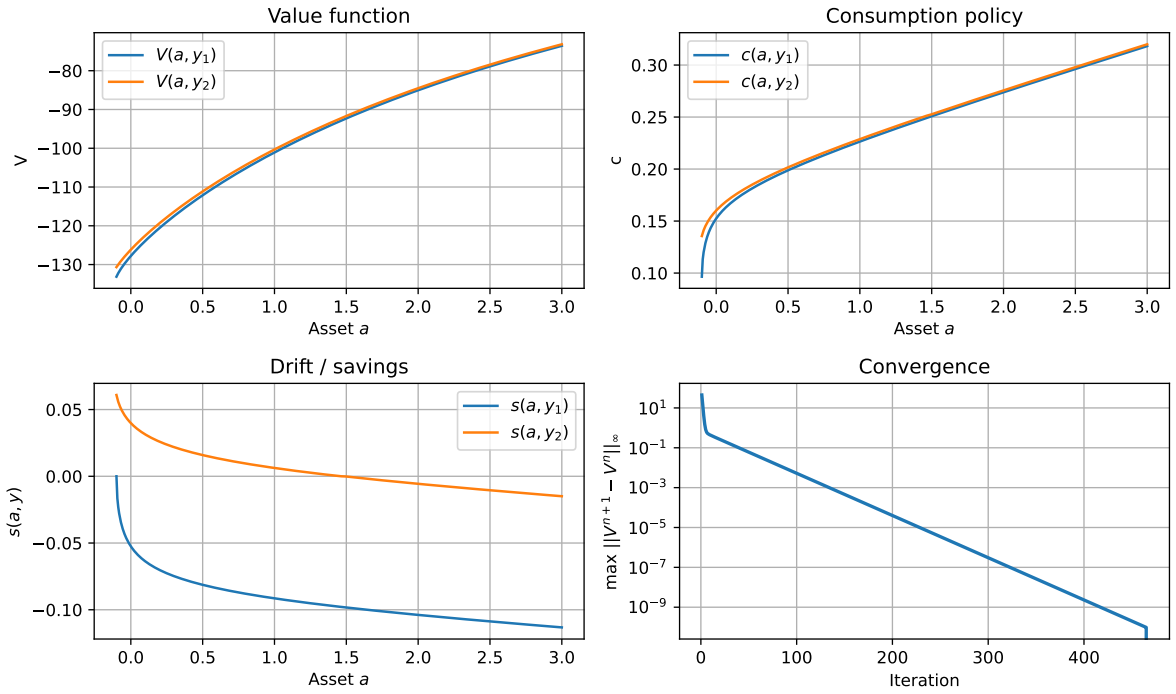
[8] The matlab runtime is still from Achdou et al. (2022) 's code.

[9] This is where sparse matrices methods really win, if we set Δ to be larger, the matlab version will take less time to solve the HJB equation.

```
V_np = jnp.array(V_star)
c_np = jnp.array(c_star)
s_np = jnp.array(s_star)
a_np = jnp.array(a)

fig, axs = plt.subplots(2, 2, figsize=(10, 6))

# 1) Value function
axs[0, 0].plot(a_np, V_np[:, 0], label="$V(a,y_1)$")
axs[0, 0].plot(a_np, V_np[:, 1], label="$V(a,y_2)$")
axs[0, 0].set_title("Value function")
axs[0, 0].set_ylabel("V")
axs[0, 0].set_xlabel("Asset $a$")
axs[0, 0].grid(True)
axs[0, 0].legend()

# 2) Consumption policy
axs[0, 1].plot(a_np, c_np[:, 0], label="$c(a,y_1)$")
axs[0, 1].plot(a_np, c_np[:, 1], label="$c(a,y_2)$")
axs[0, 1].set_title("Consumption policy")
axs[0, 1].set_ylabel("c")
axs[0, 1].set_xlabel("Asset $a$")
axs[0, 1].grid(True)
axs[0, 1].legend()

# 3) Drift / savings
axs[1, 0].plot(a_np, s_np[:, 0], label="$s(a,y_1)$")
axs[1, 0].plot(a_np, s_np[:, 1], label="$s(a,y_2)$")
axs[1, 0].set_title("Drift / savings")
axs[1, 0].set_xlabel("Asset $a$")
axs[1, 0].set_ylabel("$s(a,y)$")
axs[1, 0].grid(True)
axs[1, 0].legend()

# 4) Convergence (max residual)
axs[1, 1].plot(jnp.arange(1, n_it + 1), dist_used, linewidth=2)
axs[1, 1].set_title("Convergence")
axs[1, 1].set_xlabel("Iteration")
axs[1, 1].set_ylabel("max $||V^{n+1} - V^n||_\infty$")
axs[1, 1].set_yscale("log")
axs[1, 1].grid(True)
```

```
# axs[1, 1].plot(jnp.arange(1, n_it + 1), res_used, linewidth=2)
# axs[1, 1].set_title("Convergence")
# axs[1, 1].set_xlabel("Iteration")
# axs[1, 1].set_ylabel("max $||B^n v^{n+1} - b^n||_\infty$")
# axs[1, 1].set_yscale("log")
# axs[1, 1].grid(True)

plt.tight_layout()
plt.show()
```

```
<>:42: SyntaxWarning: invalid escape sequence '\i'
<>:42: SyntaxWarning: invalid escape sequence '\i'
/var/folders/xs/q8dt4q7926×0qg5_d_1_29br0000gn/T/ipykernel_96641/3269399479.py:42: SyntaxWarning: i
  axs[1, 1].set_ylabel("max $||V^{n+1} - V^n||_\infty$")
```



6 Implementation of the Kolmogorov Forward Equation

6.1 Discretize the KFE

We now turn to solve for the KFE as discussed in Proposition 1.1.

The rough idea is to discretize these as [10]

$$0 = -[s_{i,j} g_{i,j}]' - \lambda_j g_{i,j} + \lambda_{-j} g_{i,-j}$$
$$1 = \sum_{i=1}^{I} g_{i,1} \Delta a + \sum_{i=1}^{I} g_{i,2} \Delta a$$

- Because Equation 2 is linear in g_1 and g_2 so is the finite difference approximation.
- As a result, no iterative procedure like the one for the HJB equation is needed
- and the KFE can be solved in one step.

6.2 Upwind Scheme

We use an **upwind scheme** to approximate the derivative $[s_{i,j} g_{i,j}]'$.

There is a choice between forward and backward differences, depending on the sign of the drift term.

The upwind discretization ensures stability and correctness:

$$-\frac{(s^n_{i,j,F})^+ g_{i,j} - (s^n_{i-1,j,F})^+ g_{i-1,j}}{\Delta a} - \frac{(s^n_{i+1,j,B})^- g_{i+1,j} - (s^n_{i,j,B})^- g_{i,j}}{\Delta a} - g_{i,j}\lambda_j + g_{i,-j}\lambda_{-j} = 0$$

> **Note**
>
> Because $g_{0,j}$ and $g_{I+1,j}$ are outside the state space, their densities are zero, and $(s_{0,j,F})^+$ and $(s_{I+1,j,B})^-$ are never used.

We can collect terms to write the equation more compactly:

$$g_{i-1,j}\, z_{i-1,j} + g_{i,j}\, y_{i,j} + g_{i+1,j}\, x_{i+1,j} + g_{i,-j}\, \lambda_{-j} = 0$$

where

[10] One could also use a slightly more accurate trapezoidal rule, but results are virtually identical given the fine grid size.

$$x_{i+1,j} = -\frac{(s^n_{i,j+1,B})^-}{\Delta a}$$
$$y_{i,j} = -\frac{(s^n_{i,j,F})^+}{\Delta a} + \frac{(s^n_{i,j,B})^-}{\Delta a} - \lambda_j$$
$$z_{i-1,j} = \frac{(s^n_{i,j-1,F})^+}{\Delta a}$$

6.3 Solving the KFE

The main reason for preferring this approximation is that it allows us to write the finite difference scheme in matrix form, closely paralleling the structure used in the HJB equation:

$$\mathbf{A}^{\mathrm{T}} g = 0 \tag{11}$$

- Here, $\mathbf{A}^{\mathrm{T}}$ denotes the transpose of the intensity matrix $\mathbf{A}$ from the HJB Equation 9.
- *More precisely, $\mathbf{A}$ is the matrix built from the final HJB iteration.*
- This makes intuitive sense: this operation is exactly the same as that needed to find the steady-state (stationary) distribution of a discrete Poisson process, or a continuous-time Markov chain.

7 Code for Solving the KFE

7.1 Implement $\mathbf{A}^{\mathrm{T}} p$

```
@jax.jit
def ATp(p: jnp.ndarray, X: jnp.ndarray, Y: jnp.ndarray, Z: jnp.ndarray, lamb: jnp.ndarray) → jnp.n
    """
    Computes the product of the transposed intensity matrix A (A^T) and a stacked vector p,
    for use in the Kolmogorov Forward Equation (KFE) under a two-state Markov process.

    Parameters
    ----------
    p : jnp.ndarray
        Stacked probability vector of shape (2*I,), where p = [p0, p1] and each p0, p1 has length I
    X : jnp.ndarray
        Matrix of shape (I, 2). Coefficients for drift to lower asset neighbor (i-1) for each incom
    Y : jnp.ndarray
        Matrix of shape (I, 2). Diagonal coefficients for each asset/income state.
    Z : jnp.ndarray
```

```
        Matrix of shape (I, 2). Coefficients for drift to upper asset neighbor (i+1) for each incom
    lamb : jnp.ndarray
        Length-2 vector (lamb[0], lamb[1]) representing transition intensities between income state

    Returns
    -------
    jnp.ndarray
        Stacked vector (length 2*I) corresponding to A^T p.
        This encodes the time derivative of the distribution over (a, y) states.
    """
    I = X.shape[0]
    p0 = p[:I]
    p1 = p[I:]

    # diagonal + switching inflow
    y0 = (Y[:, 0] - lamb[0]) * p0 + lamb[1] * p1
    y1 = (Y[:, 1] - lamb[1]) * p1 + lamb[0] * p0

    # neighbor inflows in A^T:
    # from i-1 via Z[i-1]*p[i-1]
    y0 = y0.at[1:].add(Z[:-1, 0] * p0[:-1])
    y1 = y1.at[1:].add(Z[:-1, 1] * p1[:-1])

    # from i+1 via X[i+1]*p[i+1]
    y0 = y0.at[:-1].add(X[1:, 0] * p0[1:])
    y1 = y1.at[:-1].add(X[1:, 1] * p1[1:])

    return jnp.concatenate([y0, y1])
```

7.2 Uniformization for the Stationary KFE

Let A be the **generator** (intensity) matrix implied by the upwind discretization, and let $p \in \mathbb{R}^N$ be the **probability mass** on the grid ($N = I \times n_y$).
The stationary KFE is

$$A^\top p = 0 \quad \mathbf{1}^\top p = 1 \quad p \geq 0.$$

Key idea: uniformization (Jensen's method)

Choose a scalar q such that it dominates all **outflow rates**:

$$q \geq \max_i(-A_{ii})$$

Then define a **discrete-time** transition matrix

$$P \equiv I + \frac{1}{q} A$$

Because A is a valid generator (rows sum to zero, off-diagonals nonnegative), this choice implies:

- P has **nonnegative** entries,
- rows of P sum to **one** (so P is stochastic).

Moreover, the stationary distribution of the continuous-time chain is also stationary for P:

$$A^\top p = 0 \quad \iff \quad P^\top p = p.$$

So we can compute p by a simple **power iteration**:

$$p^{(k+1)} = P^\top p^{(k)} = p^{(k)} + \frac{1}{q} A^\top p^{(k)}.$$

> **Implementation notes**
>
> - In code, `ATp(p, ...)` implements the matrix-free product $A^\top p$.
> - We initialize with a uniform probability vector $p^{(0)}$.
> - Finally, convert probability mass to **density** via $g = p/\Delta a$.

7.3 Solve the KFE

```
@jax.jit(static_argnames=("maxit"))
def solve_kfe(
    X: jnp.ndarray, Y: jnp.ndarray, Z: jnp.ndarray, params: tuple, sizes: tuple, da: float, tol: fl
) → tuple:
    """
    Solves the Kolmogorov Forward Equation (KFE) for the stationary distribution.

    Parameters
    ----------
    X, Y, Z : array-like
        Coefficient matrices (I x 2 each) from the HJB solver that encode drift and generator struc
```

```
    tol : float, optional
        Convergence tolerance for fixed-point iteration (default: 1e-14).
    maxit : int, optional
        Maximum number of iterations (default: 200_000).

    Returns
    -------
    g_star : ndarray
        Stationary density, stacked as [g1, g2], length I * n_y.
    k : int
        Number of iterations used.
    diff : float
        Final difference (sup norm of probability vector update).
    mass : float
        Total mass of stationary density (should be ≈ 1).
    stat_inf : float
        Stationarity residual (should be nearly zero if converged).
    min_g : float
        Minimum value in stationary density array.
    """
    I, n_y = X.shape
    sigma, rho, r, A, y, lamb = params

    # outflow rate at each node = X + Z + lamb  (because -diag = -Y + lamb = X+Z+lamb when Y=-X-Z)
    out0 = X[:, 0] + Z[:, 0] + lamb[0]
    out1 = X[:, 1] + Z[:, 1] + lamb[1]
    q = jnp.max(jnp.concatenate([out0, out1])) + 1e-12  # safety

    N = I * n_y
    p0 = jnp.ones(N) / N  # probability mass init (sum=1)

    def cond_fun(carry):
        k, p, diff = carry
        return jnp.logical_and(k < maxit, diff > tol)

    def body_fun(carry):
        k, p, diff = carry
        p_new = p + ATp(p, X, Y, Z, lamb) / q
        diff_new = jnp.max(jnp.abs(p_new - p))
        return (k + 1, p_new, diff_new)

    k, p_star, diff = jax.lax.while_loop(cond_fun, body_fun, (0, p0, jnp.array(jnp.inf)))
```

```
    g_star = p_star / da
    mass = jnp.sum(g_star) * da
    stat_inf = jnp.max(jnp.abs(ATp(p_star, X, Y, Z, lamb)))  # should be ~0
    min_g = jnp.min(g_star)

    return g_star, k, diff, mass, stat_inf, min_g
```

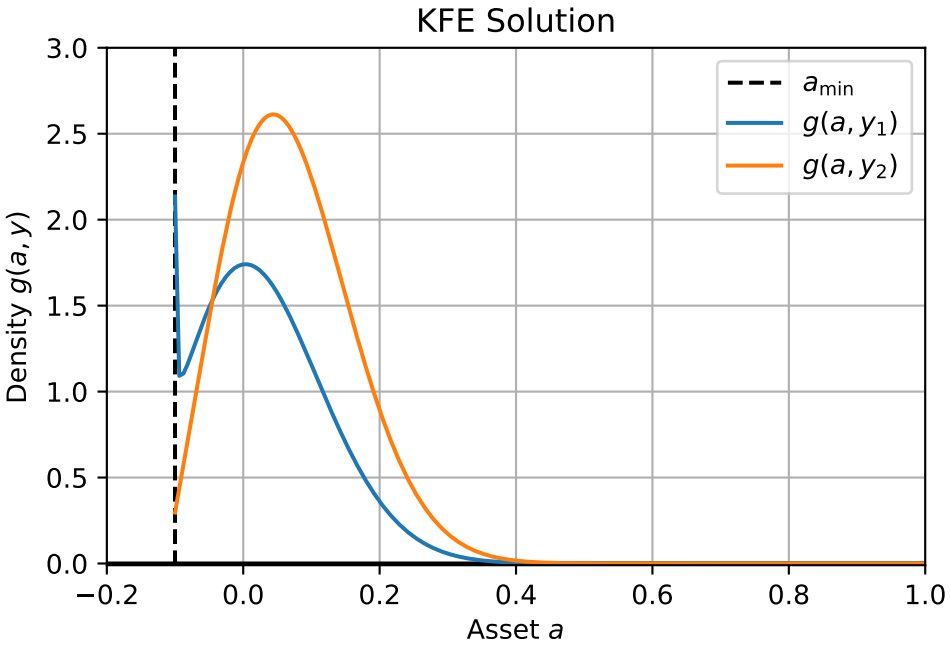
7.4 Results

```
# Build X,Y,Z from the final HJB iteration (same as you already do)
u_flow, X, Y, Z, c_imp = get_coef(V_imp, da, params, arrays, sizes)
g_u, iters, diff, mass, stat_inf, min_g = solve_kfe(X, Y, Z, params, sizes, da)  # warmup

start = time.time()
g_u, iters, diff, mass, stat_inf, min_g = solve_kfe(X, Y, Z, params, sizes, da)
g_u.block_until_ready()
uniformization_time = time.time() - start
print(f"KFE solved in {uniformization_time:.4f} s in {iters} iterations with diff = {diff:.4e}")
print(f"Total mass = {float(mass):.4e}, stationarity inf (on p) = {float(stat_inf):.4e}")
```

```
KFE solved in 0.0133 s in 6339 iterations with diff = 9.9747e-15
Total mass = 1.0000e+00, stationarity inf (on p) = 1.9582e-13
```

```
I, n_y = sizes
g1, g2 = g_u[:I], g_u[I:]

plt.figure()
plt.axhline(y=0, color="black")
plt.axvline(x=float(jnp.min(a)), color="black", linestyle="--", label="$a_{\min}$")
plt.plot(a, g1, label="$g(a,y_1)$")
plt.plot(a, g2, label="$g(a,y_2)$")
plt.title("KFE Solution")
plt.xlabel("Asset $a$")
plt.ylabel("Density $g(a,y)$")
plt.ylim([-0.01, 3])
plt.xlim([jnp.min(a) - 0.1, 1])
plt.grid(True)
plt.legend()
plt.show()
```

```
<>:6: SyntaxWarning: invalid escape sequence '\m'
<>:6: SyntaxWarning: invalid escape sequence '\m'
/var/folders/xs/q8dt4q7926×0qg5_d_1_29br0000gn/T/ipykernel_96641/3737332922.py:6: SyntaxWarning: in
  plt.axvline(x=float(jnp.min(a)), color="black", linestyle="--", label="$a_{\min}$")
```



8 Equilibrium

8.1 Aggregate asset supply

We now try to find the aggregate asset supply.

- We approximate the aggregate asset supply by:

$$S(r) \approx \sum_{i=1}^{I} a_i g_{i,1} \Delta a + \sum_{i=1}^{I} a_i g_{i,2} \Delta a$$

- It looks as expected:
  - supply is bounded below by the borrowing constraint
  - supply is increasing in the interest rate
  - $\lim_{r \to \rho} S(r) = \infty$

8.2 Calculate the Agg. Supply

```
@jax.jit
def agg_asset_supply(g_u: jnp.ndarray, a: jnp.ndarray, da: float, sizes: tuple) → jnp.ndarray:
    """
    Computes the aggregate asset supply given the density array, asset grid, grid step, and grid si

    Parameters
    ----------
    g_u : jnp.ndarray
        Flattened array of densities, stacked as [g(a, y_1), g(a, y_2)].
    a : jnp.ndarray
        Asset grid, shape (I,).
    da : float
        Asset grid spacing (Δa).
    sizes : tuple
        Tuple containing the grid sizes, where sizes[0] = I (number of asset grid points)
        and sizes[1] = n_y (number of discrete y states).

    Returns
    -------
    jnp.ndarray
        The total aggregate asset supply S, computed as:
            S = ∫ a * [g(a, y_1) + g(a, y_2)] da
        (approximated by a sum with weight da).
    """
    I = a.shape[0]
    g1, g2 = g_u[:I], g_u[I:]
    return da * jnp.sum(a * (g1 + g2))


def S_of_r(
    r_new: float,
    V_guess: jnp.ndarray,
    Delta_hjb: float,
    crit: float,
    maxit: int,
    params_base: tuple,
    arrays: tuple,
    sizes: tuple,
    da: float,
    tol: float = 1e-12,
```

```
    maxiter: int = 200,
):
    """
    Computes the aggregate asset supply S(r) for a given interest rate by solving the HJB and KFE e

    This function follows these steps for the provided interest rate `r_new`:
      1. Updates model parameters with the new interest rate.
      2. Solves the HJB (Hamilton-Jacobi-Bellman) equation via an implicit GMRES routine, warm-star
      3. Obtains necessary coefficients from the computed value function.
      4. Solves the KFE (Kolmogorov Forward Equation) using the previously obtained coefficients.
      5. Aggregates the steady-state distribution to compute the aggregate asset supply.

    Parameters
    ----------
    r_new : float
        The new interest rate at which to evaluate the asset supply.
    V_guess : jnp.ndarray
        Initial guess for the value function to warm-start the HJB solver.
    Delta_hjb : float
        Step size parameter for the HJB solver.
    crit : float
        Convergence criterion for the HJB solver.
    maxit : int
        Maximum number of iterations for the HJB solver.
    params_base : tuple
        Tuple of baseline economic parameters (sigma, rho, r, A, y, lamb),
        of which r will be replaced by `r_new`.
    arrays : tuple
        Tuple containing model arrays, such as the asset grid.
    sizes : tuple
        Tuple of grid sizes (number of grid points etc.).
    da : float
        Asset grid spacing (Δa).
    tol : float, optional
        Tolerance for solvers (default: 1e-12).
    maxiter : int, optional
        Maximum number of iterations for inner solvers (default: 200).

    Returns
    -------
    S : jnp.ndarray
        The aggregate asset supply at the given interest rate.
```

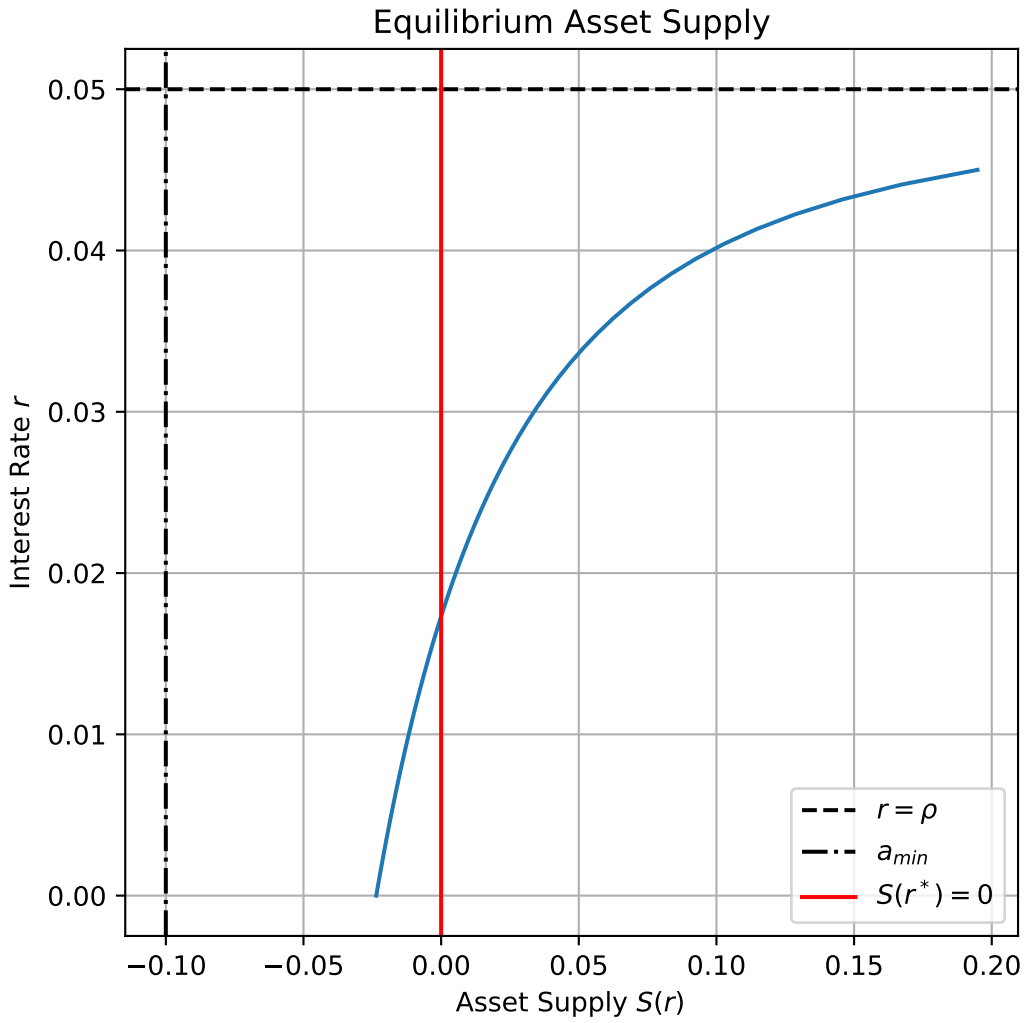
```
    V_star : jnp.ndarray
        The converged value function from the HJB solver.
    g_u : jnp.ndarray
        The stationary distribution from the KFE solver.
    err : float
        Final error or residual from the HJB solver.
    n_iter : int
        Number of iterations taken by the HJB solver.
    """
    sigma, rho, _, A, y, lamb = params_base
    params_r = (sigma, rho, r_new, A, y, lamb)

    V_star, dist, res, n_iter, err = solve_hjb_implicit_gmres(V_guess, Delta_hjb, crit, maxit, para

    u_flow, X, Y, Z, c = get_coef(V_star, da, params_r, arrays, sizes)
    g_u, *_ = solve_kfe(X, Y, Z, params_r, sizes, da)

    S = agg_asset_supply(g_u, arrays[0], da, sizes)  # arrays[0] is a
    return S, V_star, g_u, err, n_iter
```

8.3 Visualize the Equilibrium

```
r_grid = jnp.linspace(0.0, 0.045, 50)
def Sr(r,args):
    ans, *_ = S_of_r(r, *args)
    return ans
Sr_vmap = jax.vmap(Sr, in_axes=(0,None))
args = (V_imp,1.0,1e-11,5000,params,arrays,sizes,da)
%time S_grid = Sr_vmap(r_grid,args).block_until_ready()

plt.figure(figsize=(6,6))
plt.plot(S_grid, r_grid)
plt.axhline(y=0.05, color = "black", linestyle="--", label="$r=\\rho$")
plt.axvline(x=a_min, color = "black", linestyle="-.", label="$a_{min}$")
plt.axvline(x=0, color="red",  label="$S(r^*)=0$")
plt.title("Equilibrium Asset Supply")
plt.xlabel("Asset Supply $S(r)$")
plt.ylabel("Interest Rate $r$")
plt.legend()
```

```
plt.grid()
plt.show()
```

```
CPU times: user 31.1 s, sys: 2.5 s, total: 33.6 s
Wall time: 15.8 s
```



8.4 Solve for the Equilibrium Interest Rate

We'll just use `optimistix` to solve for the equilibrium interest rate which is completely compatible with JAX.

```
import optimistix as optx

solver = optx.Bisection(rtol=1e-5, atol=1e-5)
%time sol = optx.root_find(Sr, solver, y0=0.04,args=args,options=dict(lower=0,upper=0.045))
r_star = sol.value
print("-"*60)
print(f"Equilibrium interest rate: {r_star:.4f} and S(r^*) = {float(Sr(r_star,args)):.4f}")
```

```
CPU times: user 3.82 s, sys: 73 ms, total: 3.89 s
Wall time: 2.75 s
------------------------------------------------------------
Equilibrium interest rate: 0.0173 and S(r^*) = -0.0000
```

> **`optimistix` (OptX) in one paragraph**
>
> `optimistix` (often imported as `optx`) is a lightweight, JAX-native numerical optimization library.
>
> - It provides common solvers for **root finding** (e.g. Newton, Bisection), **fixed points**, and related problems, designed to work cleanly with **JAX transformations** (e.g. `jit`, `vmap`) and JAX arrays.
> - In this lecture we use `optx.root_find` with **Bisection**, which only requires the function to change sign on `[lower, upper]` and is robust even when derivatives are noisy or unavailable.

References

Achdou, Yves, Jiequn Han, Jean-Michel Lasry, Pierre-Louis Lions, and Benjamin Moll. 2022. "Income and Wealth Distribution in Macroeconomics: A Continuous-Time Approach." *Review of Economic Studies* 89: 45–86. https://doi.org/10.1093/restud/rdab002.

Candler, Graham V. 1999. "Finite-Difference Methods for Dynamic Programming Problems." In *Computational Methods for the Study of Dynamic Economies*. Cambridge, England: Cambridge University Press.

Huggett, Mark. 1993. "The Risk-Free Rate in Heterogeneous-Agent Incomplete-Insurance Economies." *Journal of Economic Dynamics and Control* 17 (5–6): 953–69. https://doi.org/10.1016/0165-1889(93)90024-M.