Kydland & Prescott (1982): Real Business Cycles

David Evans

2026-11-13

Introduction

From Micro to Macro

Where We've Been

- **McCall (1970)**: an unemployed worker searches for a job
  - solve for a reservation wage
- **Rust (1987)**: a bus superintendent decides when to replace engines
  - solve for replacement probabilities

- Both models feature a **single agent** making decisions in a **micro** environment
- Today we scale up: a **representative agent** in a **macroeconomy**

Kydland & Prescott (1982)

- **"Time to Build and Aggregate Fluctuations"**
  - *Econometrica* and 10000+ citations
- Won the **2004 Nobel Prize** in Economics

- Key question: Can a **competitive equilibrium** model with **technology shocks** explain business cycles?

The RBC Revolution

Before KP (1982), business cycles were thought to require market failures (sticky prices, coordination failures). KP showed that a **perfectly competitive** economy hit by productivity shocks could generate realistic fluctuations.

The Big Idea

- Business cycles as the **optimal response** of a competitive economy to random productivity shocks
- No market failures, no government intervention needed to explain fluctuations

- The core model: a **stochastic neoclassical growth model**
  - Households save and consume optimally
  - Firms produce using capital and labor
  - Productivity fluctuates randomly

> **Connection to Previous Lectures**
>
> The RBC model combines: **dynamic programming** (McCall, Rust), **Markov chains** (productivity shocks), and **linear algebra** (matrix operations for expectations).

A New Computational Challenge

- In McCall and Rust, the state space was **discrete** (wage offers, mileage bins)
- We stored the value function as a **vector** and looked up values directly
- In the RBC model, capital k is **continuous** — we can't just index into a vector!
- We'll introduce **linear interpolation**
  - a tool for evaluating functions at arbitrary points, not just grid points
- This unlocks a whole class of models with **continuous state spaces**

Roadmap

What We'll Cover

1. **The Growth Model** — households, firms, competitive equilibrium, and the steady state, first **without** shocks
2. **The Planner's Problem** — the First Welfare Theorem and the Bellman equation
3. **Linear Interpolation** — a new tool for continuous state spaces
4. **Solving the Deterministic Model** — value function iteration and the transition to the steady state
5. **Adding Productivity Shocks** — the stochastic growth model, reusing the same solver
6. **Calibration and Simulation** — generating artificial business cycles with `DataFrames`
7. **Linearized Solution** — differentiating the Euler equation for a fast, analytical policy rule

> **Deterministic First**
>
> We first understand the economy **without** shocks: where it is heading (the steady state) and how it gets there (the transition). Business cycles are then what happens when productivity shocks keep knocking it off that path.

The Neoclassical Growth Model

The Household

The Household's Problem

- A representative household lives forever and maximizes:

$$\max_{\{c_t, k_{t+1}\}} \sum_{t=0}^{\infty} \beta^t \log(c_t)$$

- c_t: consumption in period t
- $\beta \in (0, 1)$: discount factor
- $\log(c_t)$: log utility
  - more consumption is better, but with **diminishing returns**

> **No Uncertainty (Yet)**
>
> For now there are **no shocks**: the household knows all future prices. We add random productivity once we understand the deterministic model.

Income and Ownership

- The household **owns the capital stock** k_t and supplies **one unit of labor** each period (inelastically)
- Capital is the household's **savings vehicle**: each unit earns the **real interest rate** r_t

1. **Interest income**: $r_t k_t$ from renting its capital to firms
2. **Wage income**: w_t from supplying labor

- The household takes prices (r_t, w_t) as given: it is a **price taker** in competitive markets

> **Simplification**
>
> We fix labor supply at $\ell_t = 1$ (inelastic). The full KP (1982) model includes labor-leisure choice and time-to-build for investment, which generate additional dynamics. The core insights carry through without these.

The Budget Constraint

- Each period, the household faces the budget constraint:

$$c_t + k_{t+1} = (1 + r_t)\, k_t + w_t$$

- **Left side**: uses of funds
  - consumption today + capital carried into tomorrow
- **Right side**: sources of funds
  - capital plus interest + wages
- Capital works like a **bank account**: put k_{t+1} in today, get $(1 + r_{t+1})\, k_{t+1}$ back tomorrow
- Depreciation does **not** appear here: it lives in the **firm's** problem (coming up)

The Household's Euler Equation

- Should the household save **one extra unit** in capital at t?
- **Cost**: one less unit of consumption today reduces utility by $\frac{1}{c_t}$
- **Benefit**: $1 + r_{t+1}$ extra units of consumption tomorrow, each worth $\frac{1}{c_{t+1}}$, discounted by β
- At the optimum, cost = benefit:

$$\boxed{\frac{1}{c_t} = \beta \frac{1 + r_{t+1}}{c_{t+1}}}$$

- Rearranged: $\frac{c_{t+1}}{c_t} = \beta \left(1 + r_{t+1}\right)$
  - consumption **grows** when the interest rate beats impatience: $1 + r_{t+1} > 1/\beta$

The Firm

The Firm's Problem

- A representative firm produces output using capital and labor:

$$y_t = z_t \, k_t^{\alpha} \, \ell_t^{1-\alpha}$$

- z_t: **total factor productivity** (TFP)
  - for now **constant** at $z_t = 1$
  - later it becomes the random shock driving business cycles
- $\alpha \in (0, 1)$: capital's share of output (typically $\approx 1/3$)
- **Constant returns to scale** (CRS): doubling both inputs doubles output

The Cost of Capital

- The firm **rents** capital from households:
  - it pays the household the **interest rate** r_t per unit
  - it must return the capital **intact**, so it also replaces the fraction δ that **wears out** in production
- The **rental rate** (user cost) of capital is therefore $r_t + \delta$

Depreciation Lives in the Firm's Problem

Households earn the **interest rate** r_t on their capital; firms pay the **rental rate** $r_t + \delta$. The gap δ covers the wear and tear that happens inside the firm.

Profit Maximization

- The firm rents capital at $r_t + \delta$ and hires labor at wage w_t:

$$\pi_t = z_t \, k_t^{\alpha} \, \ell_t^{1-\alpha} - (r_t + \delta) \, k_t - w_t \, \ell_t$$

- First-order conditions (marginal product = factor cost):

$$\underbrace{\alpha \, z_t \, k_t^{\alpha-1} \, \ell_t^{1-\alpha}}_{\text{MPK}} = r_t + \delta \qquad \underbrace{(1-\alpha) \, z_t \, k_t^{\alpha} \, \ell_t^{-\alpha}}_{\text{MPL}} = w_t$$

Zero Profits

With CRS, paying each factor its marginal product **exactly exhausts output**: $(r_t + \delta)k_t + w_t\ell_t = y_t$. The firm earns zero profits in equilibrium

Factor Prices in Equilibrium

- Since the household supplies $\ell_t = 1$ unit of labor, the factor prices simplify to:

$$r_t = \alpha \, z_t \, k_t^{\alpha-1} - \delta \qquad\qquad w_t = (1-\alpha) \, z_t \, k_t^{\alpha}$$

- The interest rate **falls** with capital (diminishing returns)
- The wage **rises** with capital (more capital per worker $\rightarrow$ higher productivity)
- Both move with z_t: once productivity varies, high z_t raises **all** factor prices

Competitive Equilibrium

Definition

Competitive Equilibrium

A **competitive equilibrium** is a sequence of prices $\{r_t, w_t\}_{t=0}^{\infty}$ and allocations $\{c_t, k_{t+1}\}_{t=0}^{\infty}$ such that:

1. **Household optimizes**: given prices, the household maximizes lifetime utility subject to $c_t + k_{t+1} = (1 + r_t)k_t + w_t$
2. **Firm optimizes**: given prices, the firm's factor demands satisfy $r_t + \delta = \text{MPK}$ and $w_t = \text{MPL}$
3. **Markets clear**: labor ($\ell_t = 1$), capital (the firm rents the k_t households own), and goods ($c_t + k_{t+1} - (1 - \delta)k_t = y_t$)

From Budget Constraint to Resource Constraint

- Substituting the firm's factor prices into the household's budget constraint:

$$c_t + k_{t+1} = \left(1 + \underbrace{\alpha\, z_t\, k_t^{\alpha-1} - \delta}_{r_t}\right) k_t + \underbrace{(1-\alpha)\, z_t\, k_t^{\alpha}}_{w_t}$$

$$c_t + k_{t+1} = z_t\, k_t^{\alpha} + (1-\delta)\, k_t$$

- The budget constraint **collapses to the aggregate resource constraint**!
- Rental payments + wages = total output (zero profits under CRS)
  - the δk_t the firm pays covers exactly the capital that wore out

The Equilibrium Euler Equation

- Substituting the firm's interest rate into the household's Euler equation:

$$\frac{1}{c_t} = \beta \frac{1 + \alpha \, z_{t+1} \, k_{t+1}^{\alpha-1} - \delta}{c_{t+1}}$$

- Together with the resource constraint: **two equations** in $\{c_t, k_{t+1}\}$
  - the household's saving decision meets the firm's demand for capital
- Given k_0, these pin down the whole equilibrium path
  - but solving them directly means finding an **infinite sequence**
- Let's start with the easiest case: the **steady state**

The Steady State

What Happens in the Long Run?

- With productivity constant at $z = 1$, capital settles at a **steady state** k_{ss} where $k_{t+1} = k_t = k_{ss}$
  - consumption, the interest rate, and the wage are constant too

- At the steady state, investment exactly replaces worn-out capital:
  - Investment: $i_{ss} = \delta\, k_{ss}$
  - Consumption: $c_{ss} = k_{ss}^{\alpha} - \delta\, k_{ss}$

- Understanding the steady state helps us in **three** ways:
  1. It tells us **where capital should be** on average
  2. It guides our **grid construction** (center the grid around k_{ss})
  3. It provides a **starting point** for simulation

Step 1: The Household Pins Down r_{ss}

- In steady state $c_t = c_{t+1} = c_{ss}$, so the Euler equation becomes:

$$\frac{1}{c_{ss}} = \beta \cdot \frac{1 + r_{ss}}{c_{ss}} \quad \Longrightarrow \quad 1 = \beta\,(1 + r_{ss})$$

$$\boxed{r_{ss} = \frac{1}{\beta} - 1}$$

- The household keeps consumption **flat** only if the interest rate exactly compensates for impatience
  - $r > r_{ss}$: save more, consumption grows
  - $r < r_{ss}$: save less, consumption falls
- The **more patient** the household (β closer to 1), the **lower** the steady-state interest rate

Step 2: The Firm Pins Down k_{ss}

- Given r_{ss}, the firm's capital demand: MPK = rental rate

$$\alpha\, k_{ss}^{\alpha-1} = r_{ss} + \delta$$

- Solving for k_{ss} and substituting $r_{ss} = 1/\beta - 1$:

$$k_{ss} = \left(\frac{\alpha}{r_{ss}+\delta}\right)^{\frac{1}{1-\alpha}} \quad \Longrightarrow \quad \boxed{k_{ss} = \left(\frac{\alpha\beta}{1-\beta(1-\delta)}\right)^{\frac{1}{1-\alpha}}}$$

- A higher rental rate $r_{ss} + \delta$ means the firm demands **less** capital
  - more impatience (higher r_{ss}) or faster depreciation (higher δ) $\rightarrow$ lower k_{ss}

Interpreting the Steady State

- In steady state the household **consumes its income** (interest + wages) and leaves its capital untouched:

$$c_{ss} + k_{ss} = (1 + r_{ss})\, k_{ss} + w_{ss} \quad \Longrightarrow \quad c_{ss} = r_{ss}\, k_{ss} + w_{ss}$$

- With $\beta = 0.96$: $r_{ss} = 1/0.96 - 1 \approx 4.2\%$ per year

The Logic of the Steady State

Households keep accumulating capital until the **interest rate** it earns, $r_{ss} = \alpha k_{ss}^{\alpha-1} - \delta$ (the marginal product net of depreciation), falls to the household's **rate of impatience** $1/\beta - 1$. At that point, saving one more unit costs exactly as much as it benefits

The Social Planner's Problem

The First Welfare Theorem

Beyond the Steady State

- The steady state was easy: one household condition, one firm condition
- But what if the economy **starts away** from k_{ss}?

- To trace out the equilibrium path we would need to track, every period:
  - Household optimization (Euler equation)
  - Firm optimization (factor prices)
  - Market clearing conditions

- That's a lot of moving parts! Is there a shortcut?

> **Yes! The First Welfare Theorem!**

From Equilibrium to Planning

- The **First Welfare Theorem** states: under standard conditions (complete markets, no externalities, price-taking behavior), a competitive equilibrium allocation is **Pareto optimal**

- A Pareto optimal allocation is exactly what a **benevolent social planner** would choose

> **The Shortcut**
>
> The CE allocation = the planner's solution. So instead of tracking prices and market clearing, we can solve the **much simpler planner's problem** and recover the competitive equilibrium allocations directly.

- To find equilibrium **prices**, we can always back them out: $r_t = \alpha z_t k_t^{\alpha-1} - \delta$ and $w_t = (1-\alpha) z_t k_t^{\alpha}$

Check: The Planner's Euler Equation

- The planner ignores prices and faces only the resource constraint $c_t + k_{t+1} = z_t k_t^{\alpha} + (1-\delta)k_t$
- Saving **one extra unit** at t:
  - **Cost**: $\frac{1}{c_t}$ in utility today
  - **Benefit**: the unit produces $\alpha z_{t+1} k_{t+1}^{\alpha-1}$ extra output tomorrow, and $1-\delta$ of it survives

$$\frac{1}{c_t} = \beta \frac{\alpha\, z_{t+1}\, k_{t+1}^{\alpha-1} + 1 - \delta}{c_{t+1}}$$

- **Identical** to the equilibrium Euler equation, since $1 + r_{t+1} = \alpha z_{t+1} k_{t+1}^{\alpha-1} + 1 - \delta$
  - the household's interest rate is the planner's return on capital **net of depreciation**

The Planner's Problem

The Sequential Problem

- Given initial capital k_0, the planner chooses **entire sequences** of consumption and capital to maximize lifetime utility:

$$\max_{\{c_t,\, k_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t \log(c_t)$$

subject to, for every $t \geq 0$ (with $z_t = 1$),

$$c_t + k_{t+1} = k_t^{\alpha} + (1-\delta)\, k_t, \qquad c_t \geq 0, \quad k_{t+1} \geq 0$$

- This is an **infinite-dimensional** optimization problem
  - one choice per period, forever
- Bellman equations help us break this giant problem into something manageable!

Splitting: Today vs. the Future

- Pull the $t = 0$ term out of the sum and factor β from the rest:

$$\sum_{t=0}^{\infty} \beta^t \log(c_t) = \log(c_0) + \beta \sum_{t=1}^{\infty} \beta^{t-1} \log(c_t)$$

- The planner's problem becomes a **choice today** plus a **continuation problem** starting tomorrow:

$$\max_{c_0,\, k_1} \left\{ \log(c_0) + \beta \left[\max_{\{c_t,\, k_{t+1}\}_{t=1}^{\infty}} \sum_{t=1}^{\infty} \beta^{t-1} \log(c_t) \right] \right\}$$

- The inner problem has the **same form** as the original, just starting from k_1 instead of k_0
- Define $V(k)$ as the value of the planner's problem starting with capital k
  - then the inner max equals $V(k_1)$

The Bellman Equation

- Substituting $V(k_1)$ for the continuation problem and dropping time subscripts gives a **recursive** problem in one variable:
- State variable: k — current capital
- Control variable: k' (which pins down $c = k^\alpha + (1-\delta)k - k'$)

$$V(k) = \max_{k'} \left\{ \log(c) + \beta V(k') \right\}$$

where $c = k^\alpha + (1-\delta)k - k' > 0$

- V appears on **both sides**: we are looking for the **function** V that solves this equation

Setup in Julia

Loading Packages

```
using Plots
using LinearAlgebra
using Statistics
using Distributions
using Random
using Optim
using Interpolations
using DataFrames
```

The `GrowthModel` Struct

- A `mutable struct` holds the model parameters and the objects we'll compute from them:

```
@kwdef mutable struct GrowthModel
    β::Float64 = 0.96                    # discount factor
    α::Float64 = 0.33                    # capital share
    δ::Float64 = 0.08                    # depreciation rate
    N_k::Int = 1000                      # number of capital grid points
    k_grid::Vector{Float64} = Float64[]  # filled by setup!
end
```

```
GrowthModel
```

- The economic parameters (β, α, δ) have sensible defaults (we'll calibrate them later)
- The capital grid starts empty
  - we fill it with a `setup!` function
- With no shocks the model is cheap to solve, so we can afford a fine grid

The Steady State in Julia

- A helper that follows our derivation step by step:

```julia
function steady_state(m)
    (; α, β, δ) = m
    r_ss = 1/β - 1                          # household: β(1 + r) = 1
    k_ss = (α / (r_ss + δ))^(1/(1-α))       # firm: α k^(α-1) = r + δ
    y_ss = k_ss^α
    w_ss = (1-α) * k_ss^α                   # firm: w = MPL
    i_ss = δ * k_ss                         # replace worn-out capital
    c_ss = y_ss - i_ss
    return (k=k_ss, y=y_ss, c=c_ss, i=i_ss, r=r_ss, w=w_ss)
end
```

```
steady_state (generic function with 1 method)
```

- No type annotation on `m`: it works for **any** model with fields α, β, δ
  - we'll reuse it when we add shocks

The `setup!` Function

- Build the capital grid around the steady state:

```
function setup!(m::GrowthModel)
    k_ss = steady_state(m).k
    m.k_grid = collect(LinRange(0.2k_ss, 1.8k_ss, m.N_k))   # centered on k_ss
    return m
end
```

```
setup! (generic function with 1 method)
```

- `setup!` **mutates** the struct (note the `!` convention)
  - it fills in `k_grid`
- Calling `setup!` again after changing a parameter rebuilds the grid consistently

Creating the Model and Checking the Steady State

```
m_det = setup!(GrowthModel())                        # deterministic model
ss = steady_state(m_det)
println("k_ss = $(round(ss.k, digits=3)),  r_ss = $(round(ss.r, digits=4))")
println("w_ss = $(round(ss.w, digits=3)),  c_ss = $(round(ss.c, digits=3))")
```

```
k_ss = 4.434,  r_ss = 0.0417
w_ss = 1.095,  c_ss = 1.28
```

- Does the steady state satisfy the household's and firm's conditions? Each residual should be zero:

```
[m_det.β * (1 + ss.r) - 1,                           # household Euler: β(1 + r) = 1
 m_det.α * ss.k^(m_det.α - 1) - (ss.r + m_det.δ),    # firm: MPK = r + δ
 ss.c - (ss.r * ss.k + ss.w)]                        # budget: c = r k + w
```

```
3-element Vector{Float64}:
 0.0
 0.0
 0.0
```

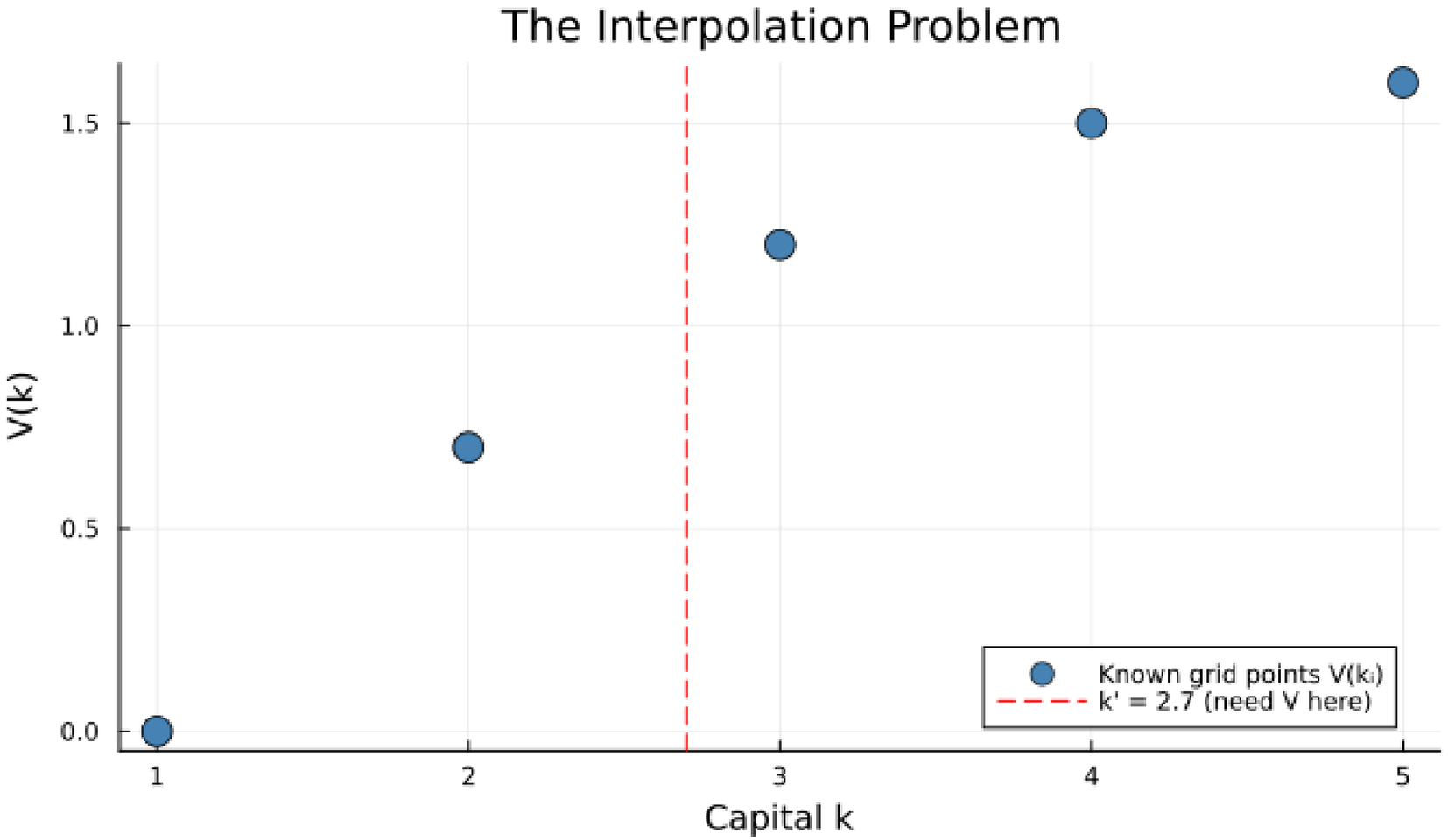
Linear Interpolation

Why We Need Interpolation

The Problem

- In McCall and Rust, the state space was **discrete**
  - we stored V as a vector and looked up values by index
- In the growth model, capital k is **continuous**
- We store V on a grid: $V(k_1), V(k_2), \ldots, V(k_N)$
- But when we optimize, the optimal k' will **not** land on a grid point!
- We need to evaluate $V(k')$ for **any** k', not just grid points
- **Solution**: **interpolation**
  - use known grid values to approximate V at off-grid points

Visual Intuition



- We know V at the blue dots
  - but we need $V(2.7)$!

How Linear Interpolation Works

The Idea

- Given two adjacent grid points k_i and k_{i+1} with known values V_i and V_{i+1}:

$$V(k') \approx V_i + \frac{k' - k_i}{k_{i+1} - k_i} \cdot (V_{i+1} - V_i)$$

- Using the **interpolation weight** $w = \frac{k' - k_i}{k_{i+1} - k_i}$:

$$V(k') \approx (1 - w) \cdot V_i + w \cdot V_{i+1}$$

Intuition

Linear interpolation draws a **straight line** between adjacent known points and reads off the value at k'. The weight w measures how far k' is from k_i toward k_{i+1}.

Step by Step

1. **Find the interval**: locate i such that $k_i \leq k' \leq k_{i+1}$
2. **Compute the weight**: $w = \frac{k' - k_i}{k_{i+1} - k_i} \in [0, 1]$
3. **Interpolate**: $\hat{V} = (1 - w)V_i + wV_{i+1}$

- If $w = 0$: k' is exactly at k_i $\rightarrow$ $\hat{V} = V_i$
- If $w = 1$: k' is exactly at k_{i+1} $\rightarrow$ $\hat{V} = V_{i+1}$
- If $w = 0.5$: k' is midway $\rightarrow$ $\hat{V} = \frac{V_i + V_{i+1}}{2}$

`Interpolations.jl`

- Julia's `Interpolations.jl` package does all of this for us with clean **call syntax**:

```
# Known grid points
k_demo = [1.0, 2.0, 3.0, 4.0, 5.0]
V_demo = [0.0, 0.7, 1.2, 1.5, 1.6]

V_itp = linear_interpolation(k_demo, V_demo)       # build interpolation object
```

```
5-element extrapolate(interpolate((::Vector{Float64},), ::Vector{Float64}, Gridded(Linear())), Throw()) with element type Float64:
 0.0
 0.7
 1.2
 1.5
 1.6
```

- Now `V_itp` behaves like a **function** — just call it with any k':

```
V_itp(2.0)         # exact grid point → 0.7
```

```
0.7
```

```
V_itp(2.5)         # midpoint → (0.7 + 1.2)/2 = 0.95
```

```
0.95
```

```
V_itp(2.7)         # arbitrary point → 0.3×0.7 + 0.7×1.2 = 1.05
```

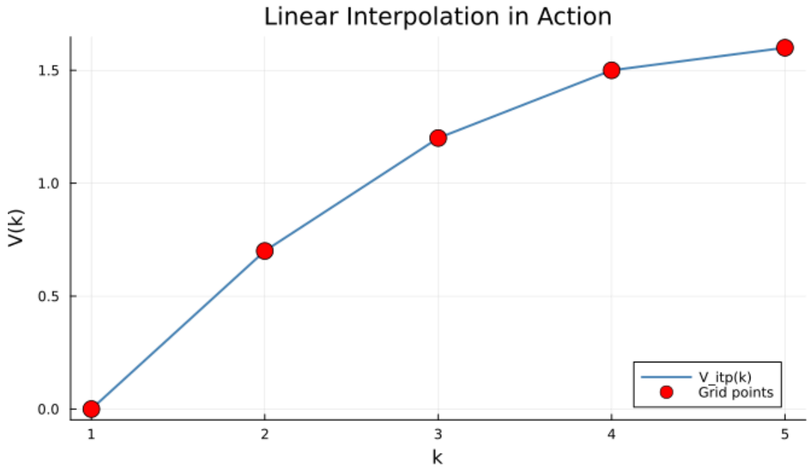
```
1.05
```

Call Syntax

`linear_interpolation(xs, ys)` returns a callable object. Writing `V_itp(k')` is just like calling a function — clean, readable, and exactly what we need inside an optimizer.

Visualizing the Interpolation

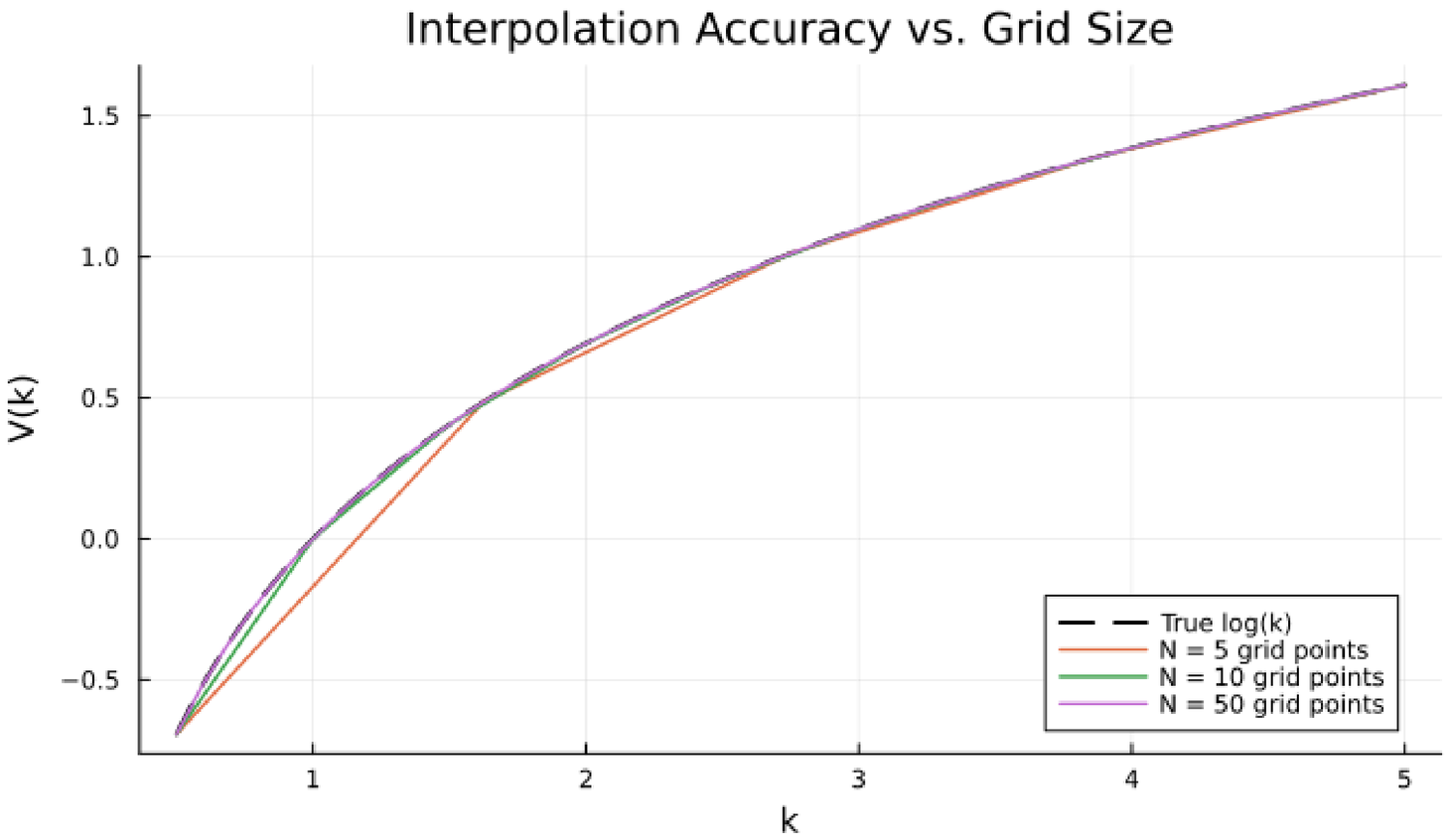
```
k_fine = LinRange(1.0, 5.0, 200)

plot(k_fine, V_itp.(k_fine), linewidth=2, color=:steelblue,
     label="V_itp(k)", legend=:bottomright, size=(700, 400))
scatter!(k_demo, V_demo, markersize=8, color=:red, label="Grid points")
xlabel!("k")
ylabel!("V(k)")
title!("Linear Interpolation in Action")
```



- The interpolation connects the dots with **straight line segments**
- Note the **broadcasting**: `V_itp.(k_fine)` evaluates the interpolant at every point in the vector

Accuracy vs. Grid Size

- With more grid points, the piecewise-linear approximation better captures curvature:



Rule of Thumb

For smooth economic functions, 100–500 grid points typically give excellent accuracy with linear interpolation.

Solving the Deterministic Model

The Planning Problem

Recall: The Bellman Equation

$$V(k) = \max_{k'} \left\{ \log(c) + \beta V(k') \right\}, \qquad c = k^{\alpha} + (1-\delta)k - k'$$

- An **infinite-horizon, recursive** problem — it looks intimidating
- But at its heart is a **simple one-period trade-off**: consume now vs. save for later

> **The plan**
>
> Boil this down to a problem you can solve in a few lines of Julia, then **iterate** it to recover V.

A Simple Problem

- Strip away the complexity: a planner has **resources** R to split between consumption c and savings k':

$$c + k' = R$$

- It values consumption today, $\log(c)$, and future capital, $\beta\, v^f(k')$:

$$\max_{k'} \quad \log(\underbrace{R - k'}_{=c}) \; + \; \beta\, v^f(k')$$

- $v^f(k')$ is the (**known**) value of having capital k' into the future
  - we'll have to find this eventually, but for now we'll take it as given
- Assume that planner must choose k' inside the range of the grid

> **This is ALL you need**
>
> Learn to solve this two-variable problem we can solve the **entire** infinite-horizon Bellman equation.

Step 1: Solve It in Julia

Numerical Optimization with `Optim.jl`

- The problem rarely has a closed form
- so we maximize $\log(R - k') + \beta\, v^f(k')$ with `Optim.jl`:

```
1 obj(kp) = -(log(resources - kp) + β * vf(kp))     # negative — we minimize
2 res = optimize(obj, k_min, k_max)
```

- `resources` is R; `vf` is the future value v^f
- We minimize the **negative** payoff (`Optim` minimizes by convention)

Example: $v^f(k') = 0$

- The planner places **no value** on the future, assuming $R = 5$

```
#inputs
resources = 5.
vf_zero(kp) = 0.0

#solve for max
obj(kp) = -(log(resources - kp) + m_det.β * vf_zero(kp))
k_max = min(m_det.k_grid[end], resources - 1e-8)
k_min = m_det.k_grid[1]
res = optimize(obj, k_min, k_max, Brent())

(kp = Optim.minimizer(res), c = resources - Optim.minimizer(res))
```

```
(kp = 0.8867671586152306, c = 4.11323284138477)
```

- Compare to

```
k_min
```

```
0.8867671427443421
```

- With no future value, the planner sets k' at the **lower bound** and consumes nearly everything

Example: $v^f(k') = \log(k')$

- Now give capital a future value of $\log(k')$:

```
 1 #inputs
 2 vf_log(kp) = log(kp)
 3 resources = 5.
 4
 5 #solve for max
 6 obj(kp) = -(log(resources - kp) + m_det.β * vf_log(kp))   # same code, new future value
 7 k_max = min(m_det.k_grid[end], resources - 1e-8)
 8 k_min = m_det.k_grid[1]
 9 res = optimize(obj, k_min, k_max)
10
11 (kp = Optim.minimizer(res), c = resources - Optim.minimizer(res))
```

```
(kp = 2.4489795934593364, c = 2.5510204065406636)
```

- The planner now **saves** since there is a future value that rewards capital
  - The optimum is **interior** (away from the bounds)

Step 2: Wrap It in a Function

`optimal_policy`

- Both examples ran the **same code**: we can wrap it in a function
  - taking **resources** R and the future value `vf` as inputs

```
function optimal_policy(m, R, vf)
    (; β, k_grid) = m
    obj(kp) = -(log(R - kp) + β * vf(kp))

    k_max = min(k_grid[end], R - 1e-8) #avoid negative consumption
    k_min = k_grid[1]
    res = optimize(obj, k_min, k_max)

    return (V  = -Optim.minimum(res),
            kp = Optim.minimizer(res),
            c  = R - Optim.minimizer(res))
end
```

```
optimal_policy (generic function with 1 method)
```

- Same body as the examples, now taking **resources** R and the future value `vf` directly
- Returns a named tuple `(V, kp, c)`
  - everything we need: values and choices
- Nothing here is specific to the deterministic model: we'll reuse it **unchanged** once we add shocks

Step 3: Iterate the Bellman Equation

The Bellman Equation *is* the Simple Problem

- The Bellman equation at capital k

$$V(k) = \max_{k'} \left\{ \log(c) + \beta V(k') \right\}, \qquad c = k^\alpha + (1-\delta)k - k'$$

  is exactly our simple problem with two substitutions:

$$R = \underbrace{k^\alpha + (1-\delta)k}_{\text{resources at } k}, \qquad v^f(k') = \underbrace{V(k')}_{\text{continuation value}}$$

- So `optimal_policy` already solves the Bellman equation at one k **if** we can supply v^f
- But v^f is V, which is what we're solving for!
- We need to find the fixed point: V that we put into the bellman equation is the V we get out.

> **The iteration loop**
>
> Guess $V \to$ interpolate it to get $v^f \to$ run `optimal_policy` at every $k \to$ get $V_{\text{new}} \to$ repeat until convergence.

From V to v^f

- `optimal_policy` needs $v^f(\cdot)$ as a callable function of k'
- But V is stored on a **grid**: $V[i] = V(k_i)$
- **Interpolate** it! One line turns the grid values into the callable $v^f(k')$:

```
vf = linear_interpolation(k_grid, V)        # vf(k') ≈ V(k') for any k'
```

- Without shocks, the future value is **just** the interpolated V
  - with shocks, we'll need an expectation first

`bellman_operator`: Pseudocode

```
function bellman_operator(m::GrowthModel, V)
    # vf = interpolate V                    # continuation value
    # For each capital k_i:
    #   R = resources at k_i
    #   optimal_policy(m, R, vf)            # optimize
    # Return V_new, kp_policy, c_policy
end
```

`bellman_operator`: Full Function

```
function bellman_operator(m::GrowthModel, V)
    (; N_k, α, δ, k_grid) = m
    V_new     = similar(V)
    kp_policy = similar(V)
    c_policy  = similar(V)
    vf = linear_interpolation(k_grid, V)            # V(k') as a function
    for ik in 1:N_k
        k = k_grid[ik]
        R = k^α + (1 - δ) * k                       # resources at k
        result = optimal_policy(m, R, vf)
        V_new[ik]     = result.V
        kp_policy[ik] = result.kp
        c_policy[ik]  = result.c
    end
    return V_new, kp_policy, c_policy
end
```

```
bellman_operator (generic function with 1 method)
```

- We build v^f **once**, then reuse it across all k
- For each k, we form resources R and hand (R, v^f) to `optimal_policy`

Iterating to Convergence

The VFI Algorithm

1. Start with an initial guess $V^0(k) = 0$
2. Apply `bellman_operator` to get V^{n+1} and policies
3. Repeat until $\|V^{n+1} - V^n\|_\infty < \text{tol}$

Convergence

The Bellman operator is a **contraction mapping** with modulus β. By the Banach fixed-point theorem, iterating converges to the unique fixed point V^* at a geometric rate.

`solveGrowth`: Pseudocode

```
function solveGrowth(m::GrowthModel; tol=1e-6, maxiter=1000, verbose=true)
    # Initial guess V = 0
    # Iterate Bellman operator until convergence
    # Build interpolated policy functions
    # Return solution (grids + interpolated functions)
end
```

`solveGrowth`: Full Function

```
function solveGrowth(m::GrowthModel; tol=1e-6, maxiter=1000, verbose=true)
    (; k_grid, N_k) = m
    V = zeros(N_k)                                  # initial guess: V = 0
    V_new, kp_policy, c_policy = bellman_operator(m, V)
    dist = 1.0; iter = 0
    while dist > tol && iter < maxiter
        V_new, kp_policy, c_policy = bellman_operator(m, V)
        dist = maximum(abs.(V_new - V))
        V = V_new
        iter += 1
    end
    if dist < tol && verbose
        println("Converged in $iter iterations (dist = $(round(dist,digits=8)))")
    end
    kp_itp = linear_interpolation(k_grid, kp_policy)  # k'(k) at any k
    c_itp  = linear_interpolation(k_grid, c_policy)   # c(k) at any k
    return (V=V, kp=kp_policy, c=c_policy, kp_itp=kp_itp, c_itp=c_itp)
end
```

```
solveGrowth (generic function with 1 method)
```

Running the Solver

```
1  sol_det = solveGrowth(m_det)
```

```
Converged in 293 iterations (dist = 1.0e-6)
```

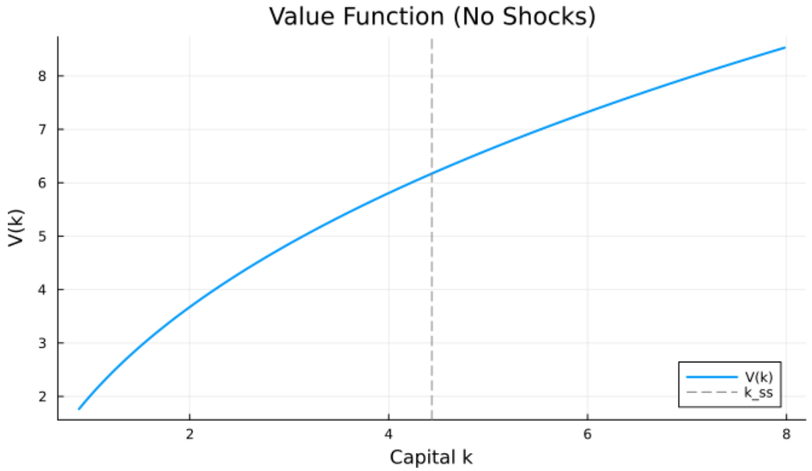
```
(V = [1.7644644786633954, 1.7806230227257094, 1.7966983524539102, 1.8126914692020115, 1.8286034134759044, 1.8444351494130145, 1.86018776242375,
1.8758624428397455, 1.8914604921888594, 1.9069830571931121  …  8.492197698384393, 8.496161882399209, 8.50012371402122, 8.504083202019583,
8.50804042794461, 8.511995355017795, 8.51594781028582, 8.519897892590805, 8.523845746500088, 8.527791355826585], kp = [1.2168720890963585,
1.223840428893597, 1.2307883964616213, 1.237730429314422, 1.2446550429276024, 1.251572206964698, 1.258485091789756, 1.265406909983036,
1.2723324962090006, 1.279252053282327  …  7.467952613392386, 7.473984301767737, 7.479897416403173, 7.485912349777228, 7.491896670422614,
7.498020107706739, 7.505023864160718, 7.511049506161238, 7.5170709978344, 7.523075912791381], c = [0.56007268867594, 0.5621705987426344,
0.5642754335791873, 0.5663729322117435, 0.5684747520282569, 0.570571092656603, 0.5726589495526839, 0.5747252725818295, 0.576775386244476,
0.5788192437005444  …  1.7950415263091521, 1.7961286866509178, 1.7973340691631243, 1.798437281892765, 1.799570756831395, 1.8005647651346264,
1.800678104793116, 1.8017692099507086, 1.802864117000592, 1.8039752528496464], kp_itp = 1000-element
extrapolate(interpolate((::Vector{Float64},), ::Vector{Float64}, Gridded(Linear())), Throw()) with element type Float64:
 1.2168720890963585
 1.223840428893597
 1.2307883964616213
 1.237730429314422
```

- Fast: one optimization per grid point per iteration
- Now let’s see what the planner does

Results

The Value Function $V(k)$

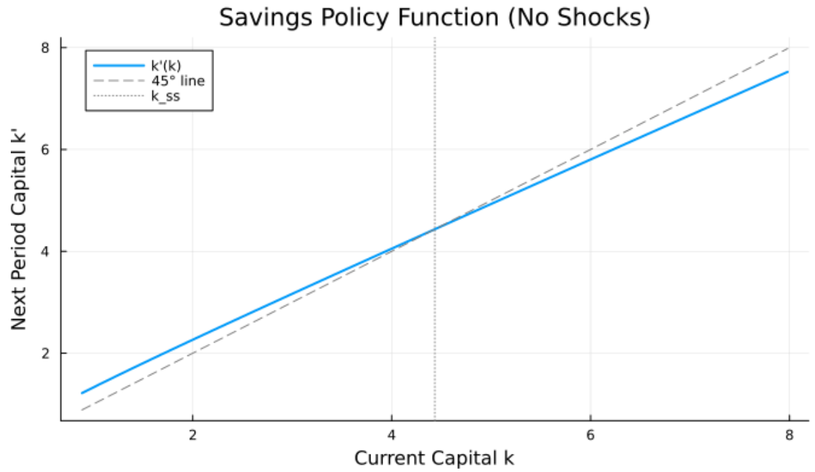
```
plot(m_det.k_grid, sol_det.V, linewidth=2, label="V(k)",
     legend=:bottomright, size=(700, 400))
vline!([ss.k], color=:gray, linestyle=:dash, label="k_ss")
xlabel!("Capital k")
ylabel!("V(k)")
title!("Value Function (No Shocks)")
```



- **Increasing** in k: more capital → more output → more consumption, now and later
- **Concave** in k (diminishing returns to capital)

The Savings Policy $k'(k)$

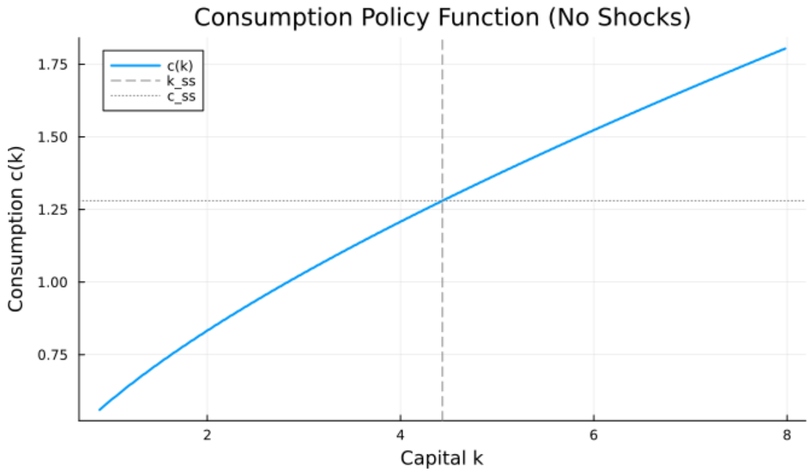
```
plot(m_det.k_grid, sol_det.kp, linewidth=2, label="k'(k)",
     legend=:topleft, size=(700, 400))
plot!(m_det.k_grid, m_det.k_grid, color=:gray, linestyle=:dash, label="45° line")
vline!([ss.k], color=:gray, linestyle=:dot, label="k_ss")
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
title!("Savings Policy Function (No Shocks)")
```



- **Above the 45° line** ($k < k_{ss}$): capital is growing ($k' > k$)
- **Below the 45° line** ($k > k_{ss}$): capital is shrinking ($k' < k$)
- The crossing point is **exactly** our analytical k_{ss}: the numerical and pencil-and-paper solutions agree!

The Consumption Policy $c(k)$

```
plot(m_det.k_grid, sol_det.c, linewidth=2, label="c(k)",
     legend=:topleft, size=(700, 400))
vline!([ss.k], color=:gray, linestyle=:dash, label="k_ss")
hline!([ss.c], color=:gray, linestyle=:dot, label="c_ss")
xlabel!("Capital k")
ylabel!("Consumption c(k)")
title!("Consumption Policy Function (No Shocks)")
```



- Consumption is **increasing** in k: richer economies consume more
- At k_{ss}, the planner consumes exactly c_{ss}

Transition to the Steady State

Simulating the Transition

```julia
function simulateGrowth(m::GrowthModel, sol, k_0, T)
    # Start from capital k_0
    # For each period:
        # use the policy to get k' = k'(k_t), then c, i
        # back out prices from the firm: r_t = MPK - δ, w_t = MPL
        # update state k_{t+1} = k'
    # Return a DataFrame with columns t, k, y, c, i, r, w
end
```

Simulating the Transition

```julia
function simulateGrowth(m::GrowthModel, sol, k_0, T)
    (; α, δ) = m
    k_sim = zeros(T); y_sim = zeros(T); c_sim = zeros(T); i_sim = zeros(T)
    r_sim = zeros(T); w_sim = zeros(T)
    k_sim[1] = k_0                              # initial capital
    for t in 1:T
        kp = sol.kp_itp(k_sim[t])               # interpolated policy!
        y_sim[t] = k_sim[t]^α
        c_sim[t] = y_sim[t] + (1-δ)*k_sim[t] - kp
        i_sim[t] = kp - (1-δ)*k_sim[t]
        r_sim[t] = α * k_sim[t]^(α-1) - δ       # firm: r = MPK - δ
        w_sim[t] = (1-α) * k_sim[t]^α           # firm: w = MPL
        if t < T
            k_sim[t+1] = kp
        end
    end
    return DataFrame(t=0:T-1, k=k_sim, y=y_sim, c=c_sim, i=i_sim, r=r_sim, w=w_sim)
end
```

```
simulateGrowth (generic function with 1 method)
```

Running the Transition

- Start the economy with **half** its steady-state capital (e.g., after a war destroys capital)

```
1 sim_det = simulateGrowth(m_det, sol_det, 0.5 * ss.k, 60)
2 first(sim_det, 4)
```

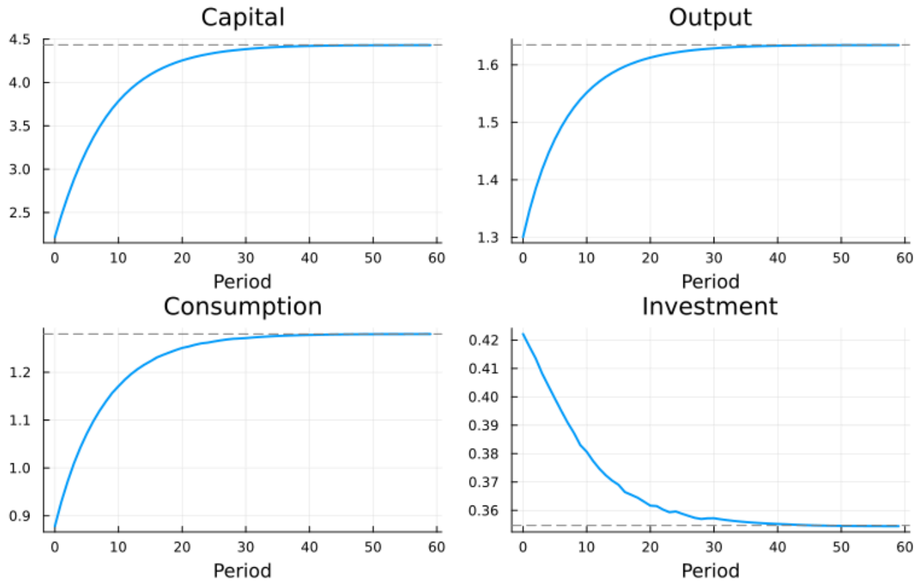
4×7 DataFrame

Row	t	k	y	c	i	r	w
	Int64	Float64	Float64	Float64	Float64	Float64	Float64
1	0	2.21692	1.30046	0.878326	0.422136	0.113581	0.871309
2	1	2.4617	1.34619	0.928461	0.417733	0.100462	0.90195
3	2	2.6825	1.3849	0.971419	0.413479	0.0903698	0.927882
4	3	2.88138	1.41797	1.00975	0.40822	0.0823985	0.950042

- A `DataFrame` stores the path as a table: one row per period, one column per variable

Quantities Converge to the Steady State

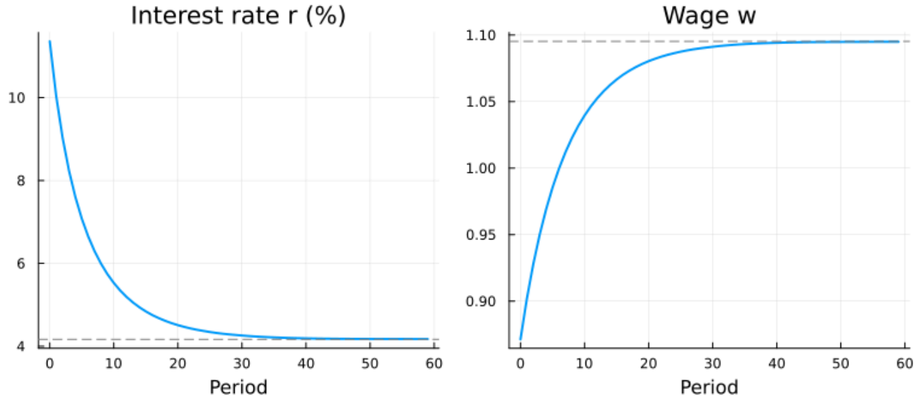
```
p1 = plot(sim_det.t, sim_det.k, linewidth=2, label="", title="Capital")
hline!(p1, [ss.k], color=:gray, linestyle=:dash, label="")
p2 = plot(sim_det.t, sim_det.y, linewidth=2, label="", title="Output")
hline!(p2, [ss.y], color=:gray, linestyle=:dash, label="")
p3 = plot(sim_det.t, sim_det.c, linewidth=2, label="", title="Consumption")
hline!(p3, [ss.c], color=:gray, linestyle=:dash, label="")
p4 = plot(sim_det.t, sim_det.i, linewidth=2, label="", title="Investment")
hline!(p4, [ss.i], color=:gray, linestyle=:dash, label="")
plot(p1, p2, p3, p4, layout=(2, 2), size=(800, 500), xlabel="Period")
```



- Capital climbs **gradually** to k_{ss}: fastest when capital is scarce
- Consumption rises **smoothly**; investment starts high and falls back to δk_{ss}

Prices Along the Transition

```
p1 = plot(sim_det.t, 100 .* sim_det.r, linewidth=2, label="", title="Interest rate r (%)")
hline!(p1, [100 * ss.r], color=:gray, linestyle=:dash, label="")
p2 = plot(sim_det.t, sim_det.w, linewidth=2, label="", title="Wage w")
hline!(p2, [ss.w], color=:gray, linestyle=:dash, label="")
plot(p1, p2, layout=(1, 2), size=(800, 350), xlabel="Period", bottom_margin=5Plots.mm)
```

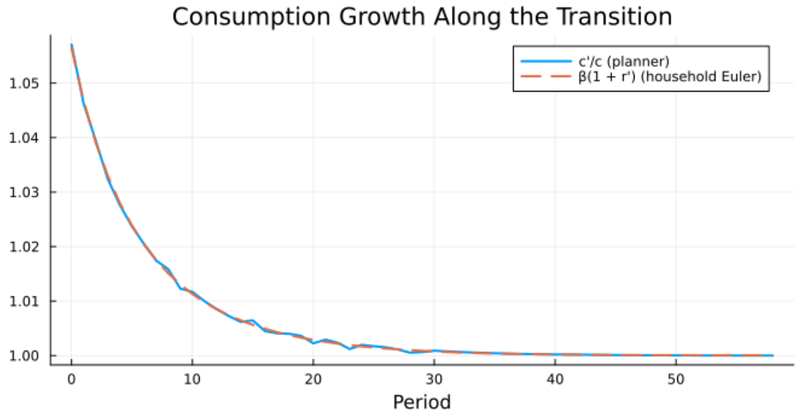


- Capital is scarce early on → high MPK → **high interest rate**
- As capital accumulates, r_t falls toward $r_{ss} = 1/\beta - 1$ and w_t rises toward w_{ss}
- The high interest rate is what makes the household **willing to save** and postpone consumption

Checking the Household's Euler Equation

- The planner never saw a price. Does its solution satisfy the household's Euler equation at the firm's interest rate, $\frac{c_{t+1}}{c_t} = \beta(1 + r_{t+1})$?

```
growth_c     = sim_det.c[2:end] ./ sim_det.c[1:end-1]   # c_{t+1}/c_t from the planner
growth_euler = m_det.β .* (1 .+ sim_det.r[2:end])      # β(1 + r_{t+1}) from the firm
plot(sim_det.t[1:end-1], growth_c, linewidth=2, label="c'/c (planner)",
     size=(700, 350), legend=:topright)
plot!(sim_det.t[1:end-1], growth_euler, linewidth=2, linestyle=:dash,
      label="β(1 + r') (household Euler)")
xlabel!("Period")
title!("Consumption Growth Along the Transition")
```



- The two lines coincide (up to small grid error): the planner's allocation **is** the competitive equilibrium
- Consumption grows while $r_{t+1} > r_{ss}$ and stops growing as $r_{t+1} \to r_{ss}$

Adding Productivity Shocks

Technology Shocks

The Driving Force

- Without shocks, the economy glides smoothly to its steady state: **no business cycles**!
- Now let productivity follow an **AR(1) process** in logs:

$$\log z_{t+1} = \rho \log z_t + \sigma_\varepsilon \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \sim N(0,1)$$

- $\rho \in (0,1)$: persistence of technology shocks (typically 0.9–0.95)
- σ_ε: volatility of technology innovations
- High ρ means a good shock today predicts good productivity tomorrow: **persistence** drives business cycle dynamics

Connecting to Markov Chains

We'll **discretize** this continuous AR(1) process into a finite-state Markov chain. This lets us use our Markov chain tools!

Discretizing the AR(1): Rouwenhorst Method

- We approximate the continuous AR(1) with an N-state Markov chain
- The **Rouwenhorst (1995)** method:
  1. Create an evenly spaced grid for $\log z$ spanning $\pm\sqrt{N-1} \cdot \sigma_z$
  2. Build the transition matrix **recursively**

> **Why Rouwenhorst?**
>
> Rouwenhorst is simpler to implement than Tauchen (1986) (an alternative method) and performs **better for highly persistent** processes (ρ close to 1): exactly the case in RBC models.

The `rouwenhorst` Function

```
function rouwenhorst(N, ρ, σ_ε)
    σ_z = σ_ε / sqrt(1 - ρ^2)                    # unconditional std dev
    p = (1 + ρ) / 2                               # persistence parameter
    P = [p 1-p; 1-p p]                           # base case: N = 2
    for n in 3:N                                  # recursively expand
        z = zeros(n - 1)
        P = p     * [P z; z' 0] + (1-p) * [z P; 0 z'] +
            (1-p) * [z' 0; P z] + p     * [0 z'; z P]
        P[2:end-1, :] ./= 2                      # normalize interior rows
    end
    z_grid = collect(LinRange(-sqrt(N-1) * σ_z,     # evenly spaced grid
                              sqrt(N-1) * σ_z, N))
    return z_grid, P
end
```

```
rouwenhorst (generic function with 1 method)
```

Testing the Rouwenhorst Approximation

```
log_z_grid, P_z = rouwenhorst(7, 0.90, 0.02)
z_grid = exp.(log_z_grid)                          # convert to levels
round.(z_grid, digits=4)
```

```
7-element Vector{Float64}:
 0.8937
 0.9278
 0.9632
 1.0
 1.0382
 1.0778
 1.1189
```

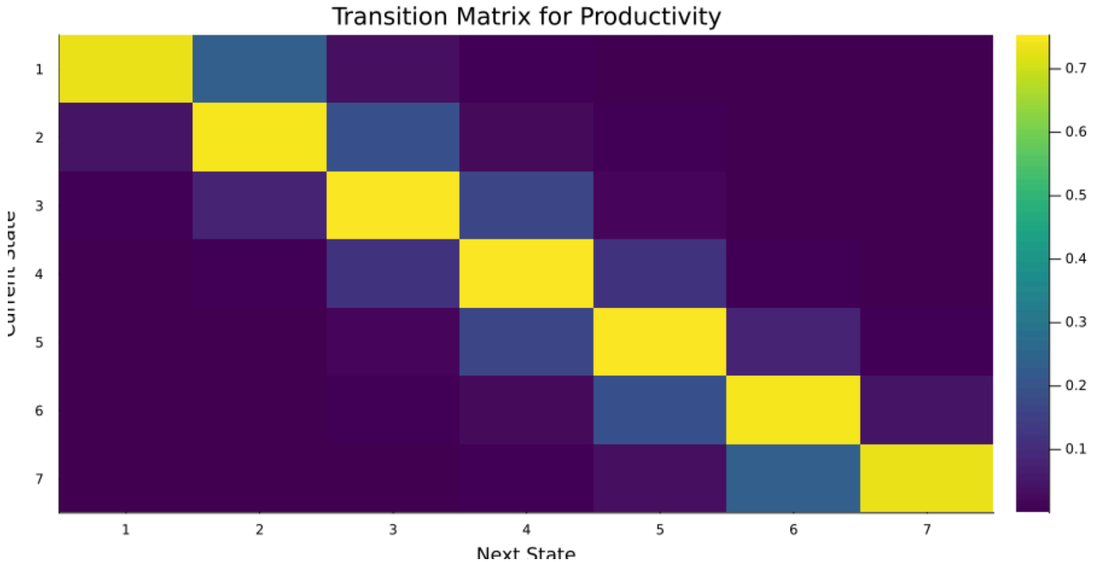
```
round.(P_z, digits=3)
```

```
7×7 Matrix{Float64}:
 0.735  0.232  0.031  0.002  0.0    0.0    0.0
 0.039  0.745  0.195  0.02   0.001  0.0    0.0
 0.002  0.078  0.751  0.156  0.012  0.0    0.0
 0.0    0.006  0.117  0.753  0.117  0.006  0.0
 0.0    0.0    0.012  0.156  0.751  0.078  0.002
 0.0    0.0    0.001  0.02   0.195  0.745  0.039
 0.0    0.0    0.0    0.002  0.031  0.232  0.735
```

- High diagonal values: the economy tends to stay near its current productivity level

Visualizing the Transition Matrix

```
1 heatmap(P_z, xlabel="Next State", ylabel="Current State",
2         title="Transition Matrix for Productivity", color=:viridis, yflip=true)
```



The Stochastic Growth Model

What Changes with Shocks?

- The household now maximizes **expected** lifetime utility: $\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \log(c_t)$
- The budget constraint and the firm's conditions are **unchanged** — they hold period by period:

$$c_t + k_{t+1} = (1 + r_t)k_t + w_t, \qquad r_t = \alpha z_t k_t^{\alpha-1} - \delta, \qquad w_t = (1-\alpha) z_t k_t^{\alpha}$$

- What's new: when saving at t, the household does **not know** r_{t+1} (it depends on z_{t+1}):

$$\frac{1}{c_t} = \beta \, \mathbb{E}_t \left[\frac{1 + r_{t+1}}{c_{t+1}} \right]$$

- The First Welfare Theorem still applies, so we again solve the **planner's problem**
  - same resource constraint, now with random z_t: $c_t + k_{t+1} = z_t k_t^{\alpha} + (1-\delta)k_t$

The Sequential Problem with Shocks

- Given initial conditions (k_0, z_0), the planner maximizes **expected** lifetime utility:

$$\max_{\{c_t,\, k_{t+1}\}_{t=0}^{\infty}} \; \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \log(c_t)$$

subject to, for every $t \geq 0$,

$$c_t + k_{t+1} = z_t\, k_t^{\alpha} + (1-\delta)\, k_t, \qquad c_t \geq 0, \quad k_{t+1} \geq 0$$

- Productivity follows the **finite-state Markov chain** from Rouwenhorst:
  - let $\bar{z}_1, \ldots, \bar{z}_S$ denote the S possible productivity values
  - let $s_t \in \{1, \ldots, S\}$ be the index, so $z_t = \bar{z}_{s_t}$
  - transition matrix $P_{s,s'} = \Pr(s_{t+1} = s' \mid s_t = s)$
- Now one choice per period, forever, across every possible realization of $\{s_t\}$!

The Bellman Equation with Shocks

- The **same splitting argument** works: today's choice plus a continuation problem starting from (k_1, s_1)
  - the **Markov property** is key: tomorrow's distribution depends only on s_1, not the whole history $\{s_0, s_1\}$
- State variables: (k, s) — current capital and productivity index
- Control variable: k' (which pins down $c = \bar{z}_s \, k^{\alpha} + (1-\delta)k - k'$)

$$V(k, s) = \max_{k'} \left\{ \log(c) + \beta \, \mathbb{E}\left[V(k', s') \mid s \right] \right\}$$

where $c = \bar{z}_s \, k^{\alpha} + (1-\delta)k - k' > 0$

- The conditional expectation is a **sum over the Markov chain**:

$$\mathbb{E}\left[V(k', s') \mid s \right] = \sum_{s'=1}^{S} P_{s,s'} \, V(k', s')$$

Solving the Stochastic Model

Reusing the Deterministic Solver

- The stochastic Bellman equation is **still** our simple problem, with:

$$R = \underbrace{\bar{z}_s \, k^{\alpha} + (1-\delta)k}_{\text{resources at } (k,\, s)}, \qquad v^f(k') = \underbrace{\mathbb{E}\left[V(k', s') \mid s\right]}_{\text{expected continuation value}}$$

- `optimal_policy(m, R, vf)` works **unchanged** — only its inputs change

What's New

1. The state is (k, s), so V becomes a **matrix**: $V[i, s] = V(k_i, s)$
2. The future value v^f is an **expectation** over s', not just the interpolated V
3. The model must store the productivity grid and transition matrix

The `RBCModel` Struct

- A new struct adds the shock parameters and the Markov chain to the growth model:

```
@kwdef mutable struct RBCModel
    β::Float64 = 0.96                   # discount factor
    α::Float64 = 0.33                   # capital share
    δ::Float64 = 0.08                   # depreciation rate
    ρ::Float64 = 0.90                   # persistence of TFP
    σ_ε::Float64 = 0.02                 # std dev of TFP innovation
    N_z::Int = 7                        # number of productivity states
    N_k::Int = 200                      # number of capital grid points

    z_grid::Vector{Float64} = Float64[] # filled by setup!
    P_z::Matrix{Float64} = zeros(0,0)   # filled by setup!
    k_grid::Vector{Float64} = Float64[] # filled by setup!
end
```

```
RBCModel
```

- New: shock parameters (ρ, σ_ε, N_z) and the Markov chain (`z_grid`, `P_z`)
- Coarser capital grid ($N_k = 200$): with 7 productivity states, each iteration solves 7× as many problems

The `setup!` Function

- Compute everything the model needs from the primitives:

```
function setup!(m::RBCModel)
    (; ρ, σ_ε, N_z, N_k) = m
    # Discretize productivity
    log_z, P = rouwenhorst(N_z, ρ, σ_ε)
    m.z_grid = exp.(log_z)
    m.P_z = P
    # Capital grid centered on (deterministic) steady state
    k_ss = steady_state(m).k
    m.k_grid = collect(LinRange(0.2k_ss, 1.8k_ss, N_k))
    return m
end
```

```
setup! (generic function with 2 methods)
```

> **Multiple Dispatch**
>
> We now have **two** `setup!` functions: one for `GrowthModel`, one for `RBCModel`. Julia picks the right **method** based on the type of `m`. We'll use the same trick for `bellman_operator`.

Creating and Inspecting the Model

```
m = setup!(RBCModel())                                  # create with defaults and set up
```

```
RBCModel(0.96, 0.33, 0.08, 0.9, 0.02, 7, 200, [0.8936953824472262, 0.927811339397771, 0.9632296400120643, 1.0, 1.0381740329206182,
1.0778053226306605, 1.118949498498781], [0.7350918906249998 0.23213428125000016 … 1.781250000000008e-6 1.5625000000000085e-8;
0.03868904687500002 0.74527321875 … 2.82187500000001e-5 2.968750000000013e-7; … ; 2.968750000000013e-7 2.8218750000000093e-5 … 0.74527321875
0.03868904687500002; 1.5625000000000085e-8 1.781250000000008e-6 … 0.23213428125000016 0.7350918906249998], [0.8867671427443421,
0.9224160731059237, 0.9580650034675052, 0.9937139338290869, 1.0293628641906685, 1.06501179455225, 1.1006607249138316, 1.1363096552754133,
1.171958585636995, 1.2076075159985764  …  7.660063911444845, 7.695712841806427, 7.7313617721680075, 7.767010702529589, 7.802659632891172,
7.838308563252753, 7.873957493614335, 7.909606423975916, 7.9452553543374975, 7.980904284699079])
```

```
println("z_grid: ", round.(m.z_grid, digits=4))
println("k_grid: [$(round(m.k_grid[1], digits=3)), ..., $(round(m.k_grid[end], digits=3))]")
println("N_k = $(m.N_k) grid points")
```

```
z_grid: [0.8937, 0.9278, 0.9632, 1.0, 1.0382, 1.0778, 1.1189]
k_grid: [0.887, ..., 7.981]
N_k = 200 grid points
```

From V to v^f with Shocks

- `optimal_policy` needs $v^f(\cdot)$ as a callable function of k'
- But V is stored on a **grid**: $V[i, s'] = V(k_i, s')$
- **Compute** the expected continuation value on the grid (rows = k, columns = current state s):

$$\mathbb{E}V[i', s] \;=\; \sum_{s'=1}^{S} P_{s,s'}\, V[i', s']$$

- **Interpolate** each column $\mathbb{E}V[:, s]$ into the callable $v^f(\cdot, s)$

`compute_vf`: The Future Value for State s

- `compute_vf(m, V, s)` builds the future value $v^f(\cdot)$ for a single current state s:

```
 1 function compute_vf(m, V, s)
 2     (; k_grid, P_z) = m
 3     Nk = length(k_grid)
 4     S = length(P_z[s, :])
 5     EV_s = zeros(Nk)                        # E[V(k', s') | s] on the grid
 6     for i' in 1:Nk
 7         EV_s[i'] = sum(P_z[s, s'] * V[i', s'] for s' in 1:S)
 8     end
 9     #construct the interpolated callable function
10     return linear_interpolation(k_grid, EV_s)
11 end
```

```
compute_vf (generic function with 1 method)
```

- Interpolating it returns the callable $v^f(k')$
  - exactly what `optimal_policy` expects as input
- The deterministic version was just `linear_interpolation(k_grid, V)`: now we take the expectation first

`bellman_operator`: Pseudocode

```
function bellman_operator(m::RBCModel, V)
    # For each state s:
    #   vf = compute_vf(m, V, s)            # expected continuation value
    #   For each capital k_i:
    #     R = resources at (k_i, s)
    #     optimal_policy(m, R, vf)          # optimize
    # Return V_new, kp_policy, c_policy
end
```

`bellman_operator`: Full Function

```julia
function bellman_operator(m::RBCModel, V)
    (; N_k, N_z, α, δ, z_grid, k_grid) = m
    V_new     = similar(V)
    kp_policy = similar(V)
    c_policy  = similar(V)
    for s in 1:N_z
        vf = compute_vf(m, V, s)                    # E[V(k', s')|s] as a function
        z  = z_grid[s]
        for ik in 1:N_k
            k = k_grid[ik]
            R = z * k^α + (1 - δ) * k               # resources at (k, s)
            result = optimal_policy(m, R, vf)
            V_new[ik, s]     = result.V
            kp_policy[ik, s] = result.kp
            c_policy[ik, s]  = result.c
        end
    end
    return V_new, kp_policy, c_policy
```

```
bellman_operator (generic function with 2 methods)
```

- Compared to the deterministic version: an extra loop over s, and `compute_vf` in place of a plain interpolation
- For each state s, `compute_vf` builds v^f **once**, then we reuse it across all k

`solveRBC`: Pseudocode

```
function solveRBC(m::RBCModel; tol=1e-6, maxiter=1000, verbose=true)
    # Initial guess V = 0
    # Iterate Bellman operator until convergence
    # Build interpolated policy functions
    # Return solution (grids + interpolated functions)
end
```

`solveRBC`: Full Function

```
function solveRBC(m::RBCModel; tol=1e-6, maxiter=1000, verbose=true)
    (; k_grid, N_k, N_z) = m
    V = zeros(N_k, N_z)                          # initial guess: V = 0
    V_new, kp_policy, c_policy = bellman_operator(m, V)
    dist = 1.0; iter = 0
    while dist > tol && iter < maxiter
        V_new, kp_policy, c_policy = bellman_operator(m, V)
        dist = maximum(abs.(V_new - V))
        V = V_new
        iter += 1
    end
    if dist < tol && verbose
        println("Converged in $iter iterations (dist = $(round(dist,digits=8)))")
    end
    kp_itp = [linear_interpolation(k_grid, kp_policy[:, s]) for s in 1:N_z]
    c_itp  = [linear_interpolation(k_grid, c_policy[:, s])  for s in 1:N_z]
    return (V=V, kp=kp_policy, c=c_policy, kp_itp=kp_itp, c_itp=c_itp)
end
```

```
solveRBC (generic function with 1 method)
```

- Same loop as `solveGrowth`; V is now an $N_k \times N_z$ matrix with one interpolated policy per state

Running the Solver

```
m = setup!(RBCModel())
sol = solveRBC(m)
```

```
Converged in 294 iterations (dist = 9.6e-7)
```

```
(V = [0.5450767096793305 0.9524145298018536 … 2.601594644353457 3.019041640967936; 0.6288520459017412 1.0350077981944352 … 2.6796968957245433
3.096067287082055; … ; 7.561499679633554 7.877213571940094 … 9.168728441903228 9.498811277963835; 7.582033994564341 7.897513180113204 …
9.188095953883254 9.51794445504028], kp = [1.1539249754881118 1.171958597827226 … 1.2641278395321915 1.285496326920585; 1.1875771745562833
1.207607520388135 … 1.2998968026283584 1.3217391807554197; … ; 7.358439432225009 7.4025190720995875 … 7.589068507764866 7.644303444974232;
7.388480155618007 7.432574853941241 … 7.624414964838906 7.674711618118873], c = [0.5208484138810829 0.5356042861902821 … 0.5875971126232149
0.6057730746662906; 0.5312381980186944 0.544426612586032 … 0.5981867053510661 0.6164064753257412; … ; 1.7222221898789316 1.745749781405082 …
1.8564418373926106 1.8827418787488632; 1.7275968224578087 1.7512093081341567 … 1.8570501398025003 1.8884090088944356], kp_itp =
Interpolations.Extrapolation{Float64, 1, Interpolations.GriddedInterpolation{Float64, 1, Vector{Float64}, Gridded{Linear{Throw{OnGrid}}},
Tuple{Vector{Float64}}}, Gridded{Linear{Throw{OnGrid}}}, Throw{Nothing}}[200-element extrapolate(interpolate((::Vector{Float64},),
::Vector{Float64}, Gridded(Linear())), Throw()) with element type Float64:
 1.1539249754881118
 1.1875771745562833
 1.2210546041465316
 1.2543945213699295
```

> **Patience**
>
> This takes a moment because each Bellman iteration solves $N_k \times N_z$ optimization problems. More efficient methods exist (policy iteration, endogenous grid method), but VFI with interpolation is the clearest to understand.

Calibration

Why Calibrate?

- We need values for five parameters: $(\alpha, \beta, \delta, \rho, \sigma_\varepsilon)$
- Rather than **estimating** them econometrically, the RBC literature **calibrates**: pick values so that the model's **steady state** matches long-run averages in the data

Calibration vs. Estimation

Calibration uses **different data moments** than the ones we're trying to explain. We match long-run averages (growth accounting, interest rates) and then ask: can the model reproduce **business cycle** fluctuations?

Matching Long-Run Moments

Parameter	Target	Data Source	Value
α	Capital's share of income	National accounts: $(r+\delta)K/Y \approx 1/3$	0.33
β	Real interest rate	$r = 1/\beta - 1 \approx 4\%$/yr	0.96
δ	Investment–capital ratio	$I/K \approx 8\%$/yr	0.08

- $\alpha = 0.33$: from NIPA data, labor's share of income $\approx 2/3$, so capital's share $\approx 1/3$
- $\beta = 0.96$: household's steady-state Euler equation $r_{ss} = 1/\beta - 1$; with $r \approx 4\%$ we get $\beta \approx 0.96$
- $\delta = 0.08$: annual depreciation of the capital stock is roughly 8%

The Productivity Process

- KP model the **Solow residual** (total factor productivity) as an AR(1):

$$\log z_t = \rho \log z_{t-1} + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma_\varepsilon^2)$$

- The Solow residual is computed from data:

$$\log z_t = \log Y_t - \alpha \log K_t - (1-\alpha) \log L_t$$

- Estimate the AR(1) by OLS regression of $\log z_t$ on $\log z_{t-1}$:
  - $\hat{\rho} \approx 0.90$ — productivity shocks are **very persistent**
  - $\hat{\sigma}_\varepsilon \approx 0.02$ — innovations are small (≈ 2% of output)

Our Calibration

```
println("Calibrated Parameters:")
println("  α  = $(m.α)   (capital share)")
println("  β  = $(m.β)   (discount factor → r ≈ $(round(1/m.β - 1, digits=3)))")
println("  δ  = $(m.δ)   (depreciation rate)")
println("  ρ  = $(m.ρ)   (TFP persistence)")
println("  σ_ε = $(m.σ_ε)  (TFP innovation std dev)")
```

```
Calibrated Parameters:
  α  = 0.33   (capital share)
  β  = 0.96   (discount factor → r ≈ 0.042)
  δ  = 0.08   (depreciation rate)
  ρ  = 0.9   (TFP persistence)
  σ_ε = 0.02  (TFP innovation std dev)
```

```
println("Implied Steady State:")
println("  K/Y  = $(round(ss.k / ss.y, digits=2))    (data: ≈ 2.8)")
println("  I/Y  = $(round(ss.i / ss.y, digits=2))    (data: ≈ 0.25-0.30)")
println("  C/Y  = $(round(ss.c / ss.y, digits=2))    (data: ≈ 0.65-0.70)")
```

```
Implied Steady State:
  K/Y  = 2.71    (data: ≈ 2.8)
  I/Y  = 0.22    (data: ≈ 0.25-0.30)
  C/Y  = 0.78    (data: ≈ 0.65-0.70)
```

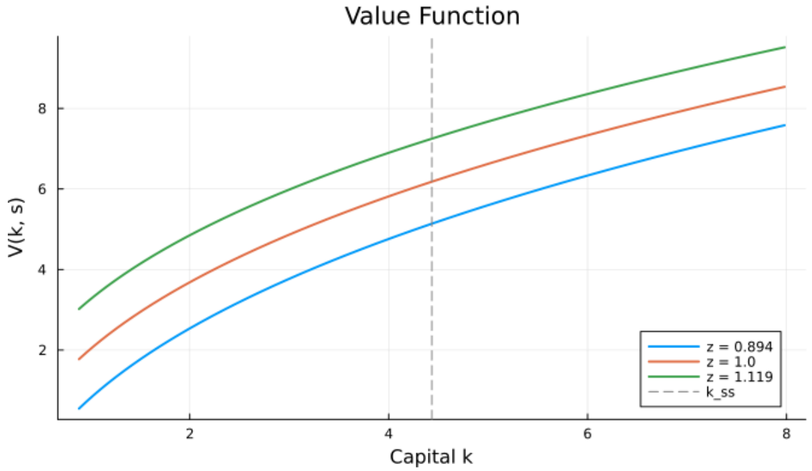
The Calibration Strategy

1. Match α from **factor shares** in national income accounts
2. Match β and δ from **long-run interest rates** and **investment rates**
3. Match ρ and σ_ε from the **estimated Solow residual** process
4. Then ask: does the model reproduce **business cycle statistics** it was *not* calibrated to match?

Results

The Value Function

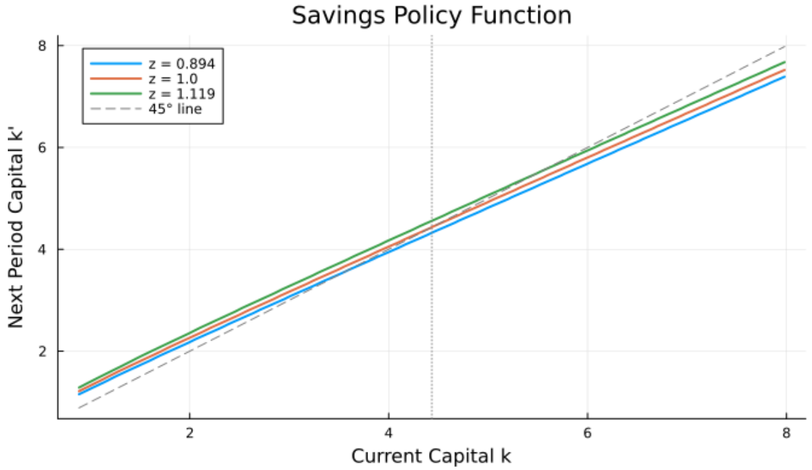
```
plot(legend=:bottomright, size=(700, 400))
for s in [1, 4, 7]
    plot!(m.k_grid, sol.V[:, s], linewidth=2,
          label="z = $(round(m.z_grid[s], digits=3))")
end
xlabel!("Capital k")
ylabel!("V(k, s)")
title!("Value Function")
vline!([ss.k], color=:gray, linestyle=:dash, label="k_ss")
```



- Higher productivity $z \rightarrow$ higher value (more output at every k)
- Value is **concave** in k (diminishing returns to capital)

The Savings Policy $k'(k, s)$

```
plot(legend=:topleft, size=(700, 400))
for s in [1, 4, 7]
    plot!(m.k_grid, sol.kp[:, s], linewidth=2,
          label="z = $(round(m.z_grid[s], digits=3))")
end
plot!(m.k_grid, m.k_grid, color=:gray, linestyle=:dash, label="45° line")
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
title!("Savings Policy Function")
vline!([ss.k], color=:gray, linestyle=:dot, label="")
```

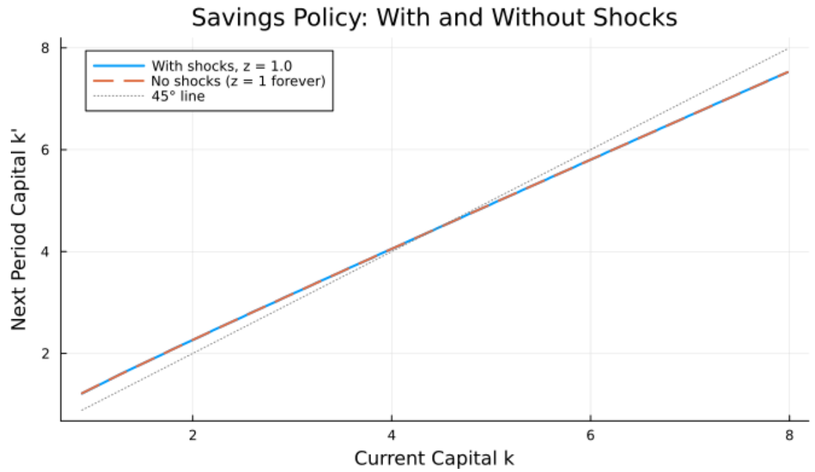


- **Above the 45° line**: capital is growing ($k' > k$)
- **Below the 45° line**: capital is shrinking ($k' < k$)
- High z shifts the policy **up**: the economy accumulates capital in booms and runs it down in busts

Shocks vs. No Shocks

- How does the stochastic policy at average productivity ($z = 1$) compare to the deterministic policy?

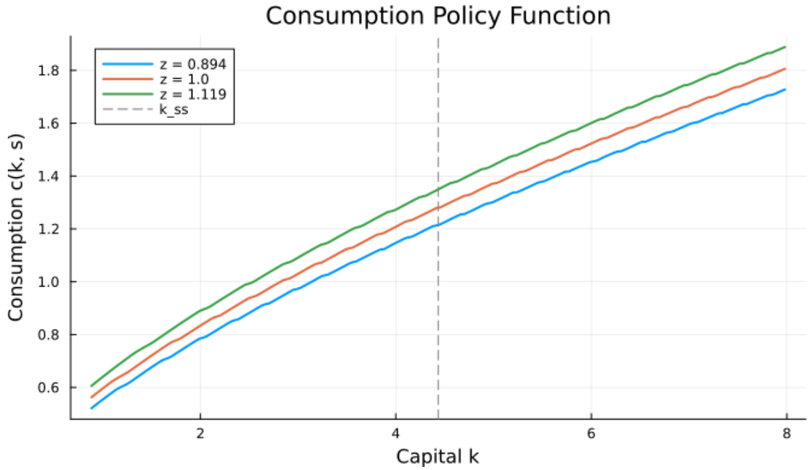
```
plot(m.k_grid, sol.kp[:, 4], linewidth=2,
     label="With shocks, z = $(round(m.z_grid[4], digits=3))", legend=:topleft, size=(700, 400))
plot!(m.k_grid, sol_det.kp_itp.(m.k_grid), linewidth=2, linestyle=:dash,
      label="No shocks (z = 1 forever)")
plot!(m.k_grid, m.k_grid, color=:gray, linestyle=:dot, label="45° line")
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
title!("Savings Policy: With and Without Shocks")
```



- Nearly identical: with log utility and small shocks, **uncertainty itself** barely changes saving
- What matters is the **realized** z, which moves the policy up and down

The Consumption Policy $c(k, s)$

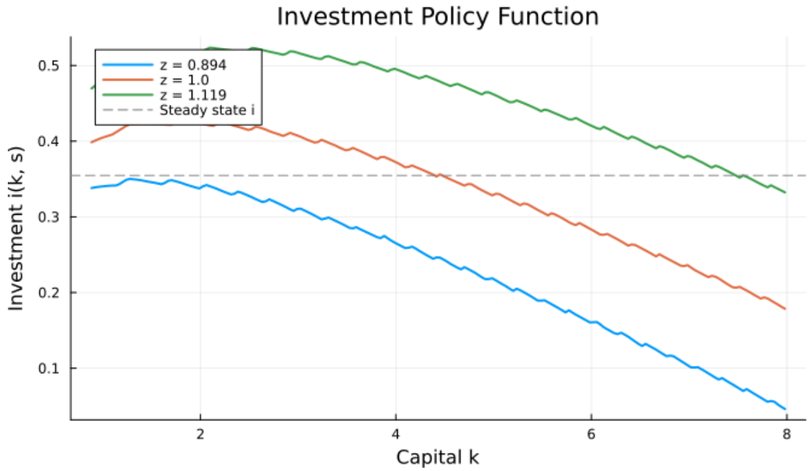
```
plot(legend=:topleft, size=(700, 400))
for s in [1, 4, 7]
    plot!(m.k_grid, sol.c[:, s], linewidth=2,
          label="z = $(round(m.z_grid[s], digits=3))")
end
xlabel!("Capital k")
ylabel!("Consumption c(k, s)")
title!("Consumption Policy Function")
vline!([ss.k], color=:gray, linestyle=:dash, label="k_ss")
```



- Consumption is **increasing** in both k and z
- High productivity → more output → consume more **and** save more

The Investment Policy

```
i_policy = sol.kp .- (1 - m.δ) .* m.k_grid         # i = k' - (1-δ)k

plot(legend=:topleft, size=(700, 400))
for s in [1, 4, 7]
    plot!(m.k_grid, i_policy[:, s], linewidth=2,
          label="z = $(round(m.z_grid[s], digits=3))")
end
xlabel!("Capital k")
ylabel!("Investment i(k, s)")
title!("Investment Policy Function")
hline!([m.δ * ss.k], color=:gray, linestyle=:dash, label="Steady state i")
```



- Investment responds **strongly** to productivity
  - when z is high, the agent invests heavily
- This foreshadows a key business cycle fact: **investment is volatile**

Simulation

Simulating the Economy

The Simulation Function

```
function simulateRBC(m::RBCModel, sol, T; seed=42)
    # Initialize at steady state
    # For each period:
        # use policies to get k', c, i
        # draw next period's productivity from the Markov chain
        # update state k_{t+1} = k'
    # Return a DataFrame with columns t, k, z, y, c, i
end
```

The Simulation Function

```julia
function simulateRBC(m::RBCModel, sol, T; seed=42)
    Random.seed!(seed)
    (; α, δ, z_grid, P_z, N_z) = m
    k_sim = zeros(T); z_sim = zeros(T); s_sim = zeros(Int, T)
    y_sim = zeros(T); c_sim = zeros(T); i_sim = zeros(T)
    k_sim[1] = steady_state(m).k                    # start at steady state
    s_sim[1] = (N_z + 1) ÷ 2                        # start at median z
    for t in 1:T
        s = s_sim[t]
        z = z_grid[s]
        kp = sol.kp_itp[s](k_sim[t])               # interpolated policy!
        z_sim[t] = z
        y_sim[t] = z * k_sim[t]^α
        c_sim[t] = y_sim[t] + (1-δ)*k_sim[t] - kp
        i_sim[t] = kp - (1-δ)*k_sim[t]
        if t < T
            k_sim[t+1] = kp
            s_sim[t+1] = rand(Categorical(P_z[s, :]))
```

```
simulateRBC (generic function with 1 method)
```

Using Interpolated Policies

- Just like `simulateGrowth`, simulation just **calls** the interpolated policy, now one per state:

```
kp = sol.kp_itp[s](k_sim[t])        # reads like math: k'(k_t, s_t)
```

- The current capital k_t may not be on our grid (because last period's optimal k' was found by continuous optimization)
- The interpolated policy function evaluates $k'(k_t, s_t)$ at **any** k_t

> **Interpolation Everywhere**
>
> Linear interpolation is used in **two places**: (1) inside VFI to evaluate the continuation value, and (2) in simulation to evaluate the policy function. Both are essential for handling the continuous state space.

Running the Simulation

```
1 sim = simulateRBC(m, sol, 200)
2 first(sim, 6)
```

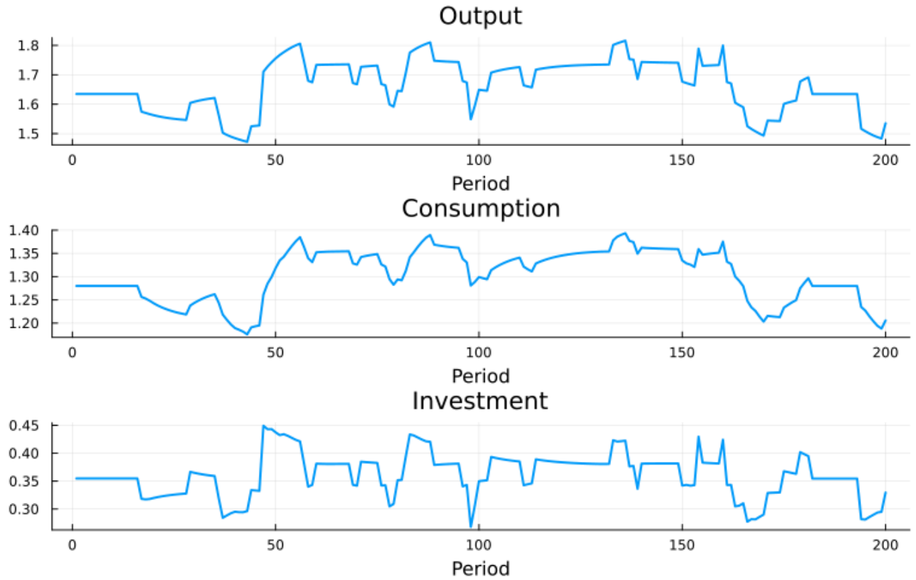
6×6 DataFrame

Row	t	k	z	y	c	i
	Int64	Float64	Float64	Float64	Float64	Float64
1	1	4.43384	1.0	1.6347	1.27999	0.354707
2	2	4.43384	1.0	1.6347	1.27999	0.354707
3	3	4.43384	1.0	1.6347	1.27999	0.354707
4	4	4.43384	1.0	1.6347	1.27999	0.354707
5	5	4.43384	1.0	1.6347	1.27999	0.354707
6	6	4.43384	1.0	1.6347	1.27999	0.354707

- Each row is one period; columns are the macro aggregates
- DataFrames give us a clean, tabular view of the simulation and powerful tools for computing statistics

Output, Consumption, and Investment

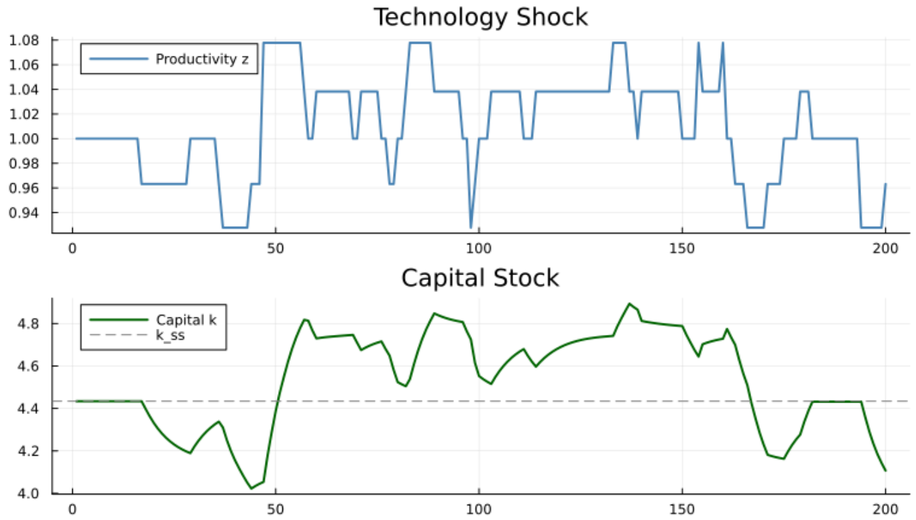
```
p1 = plot(sim.y, linewidth=2, label="", title="Output", size=(800, 150))
p2 = plot(sim.c, linewidth=2, label="", title="Consumption", size=(800, 150))
p3 = plot(sim.i, linewidth=2, label="", title="Investment", size=(800, 150))
plot(p1, p2, p3, layout=(3, 1), size=(800, 500), xlabel="Period")
```



- Consumption is **smoother** than output
  - households smooth consumption over time
- Investment is **more volatile** than output
  - the main adjustment margin

Productivity and Capital

```
p1 = plot(sim.z, linewidth=2, color=:steelblue, label="Productivity z",
          title="Technology Shock", size=(800, 200))
p2 = plot(sim.k, linewidth=2, color=:darkgreen, label="Capital k",
          title="Capital Stock", size=(800, 200))
hline!(p2, [ss.k], color=:gray, linestyle=:dash, label="k_ss")
plot(p1, p2, layout=(2,1), size=(800, 450))
```



- Capital responds **sluggishly** to productivity shocks
  - it takes time to build (or deplete) capital, just like the deterministic transition
- This is the **propagation mechanism** that turns short-lived shocks into persistent cycles

Business Cycle Statistics

Computing Moments

- Simulate a long series and construct **log-deviation** :

```
sim_long = simulateRBC(m, sol, 10_000)

sim_long.y_dev = log.(sim_long.y) .- mean(log.(sim_long.y))
sim_long.c_dev = log.(sim_long.c) .- mean(log.(sim_long.c))
sim_long.i_dev = log.(sim_long.i) .- mean(log.(sim_long.i))
sim_long.k_dev = log.(sim_long.k) .- mean(log.(sim_long.k))

first(sim_long, 6) #report the first 6 rows
```

6×10 DataFrame

Row	t	k	z	y	c	i	y_dev	c_de
	Int64	Float64	Float64	Float64	Float64	Float64	Float64	Float
1	1	4.43384	1.0	1.6347	1.27999	0.354707	-0.00359466	-0.0(
2	2	4.43384	1.0	1.6347	1.27999	0.354707	-0.00359466	-0.0(
3	3	4.43384	1.0	1.6347	1.27999	0.354707	-0.00359466	-0.0(
4	4	4.43384	1.0	1.6347	1.27999	0.354707	-0.00359466	-0.0(
5	5	4.43384	1.0	1.6347	1.27999	0.354707	-0.00359466	-0.0(

Standard Deviations

```
stats = DataFrame(
    Variable   = ["Output", "Consumption", "Investment", "Capital"],
    σ_pct      = [100*std(sim_long.y_dev), 100*std(sim_long.c_dev), 100*std(sim_long.i_dev), 100*std(sim_long.k_dev)],
    σ_relative = [1.0, std(sim_long.c_dev)/std(sim_long.y_dev), std(sim_long.i_dev)/std(sim_long.y_dev), std(sim_long.k_dev)/std(sim_long.y_d
    corr_y     = [1.0, cor(sim_long.y_dev, sim_long.c_dev), cor(sim_long.y_dev, sim_long.i_dev), cor(sim_long.y_dev, sim_long.k_dev)]
)
stats
```

4×4 DataFrame

Row	Variable	σ_pct	σ_relative	corr_y
	String	Float64	Float64	Float64
1	Output	6.15173	1.0	1.0
2	Consumption	5.29062	0.860021	0.965192
3	Investment	11.1569	1.81362	0.892204
4	Capital	6.4229	1.04408	0.853609

Business Cycle Observations

Key Business Cycle Facts

The model reproduces two central stylized facts:

1. **Consumption is less volatile than output** ($\sigma_c/\sigma_y < 1$) — consumption smoothing
2. **Investment is more volatile than output** — investment absorbs the shocks
3. All variables are **procyclical** (positively correlated with output)

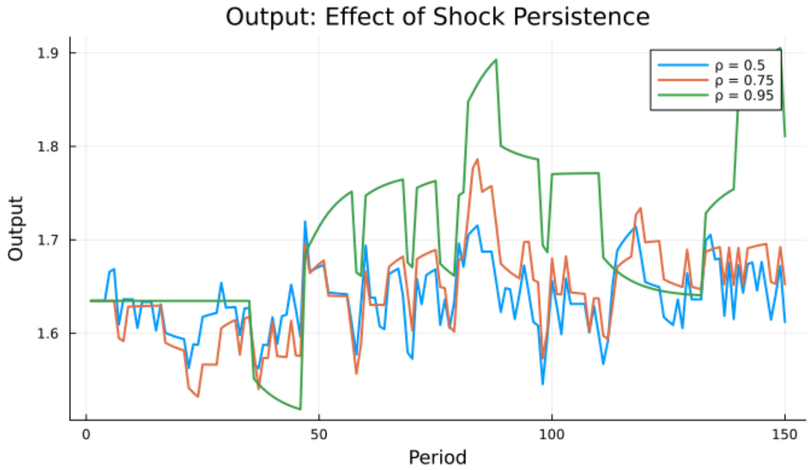
The RBC Achievement

With just **one source of randomness** (technology shocks), the model generates realistic comovement among macro aggregates. This was revolutionary in the early 1980s.

Comparative Statics

Effect of Persistence (ρ)

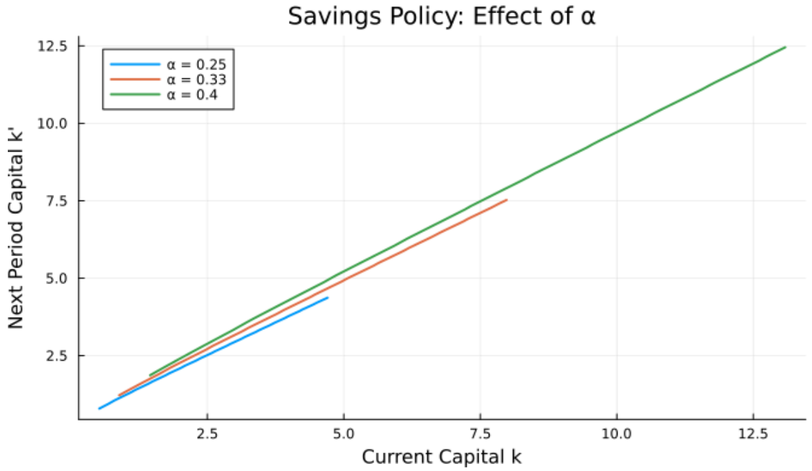
```julia
plot(legend=:topright, size=(700, 400), title="Output: Effect of Shock Persistence")
for ρ_val in [0.5, 0.75, 0.95]
    m_trial = setup!(RBCModel(ρ=ρ_val, N_k=100))
    sol_trial = solveRBC(m_trial, tol=1e-5, verbose=false)
    sim_trial = simulateRBC(m_trial, sol_trial, 150)
    plot!(sim_trial.y, linewidth=2, label="ρ = $ρ_val")
end
xlabel!("Period")
ylabel!("Output")
```



- Higher persistence → longer, more pronounced business cycles
- Low persistence → output fluctuates rapidly but reverts quickly

Effect of Capital Share (α)

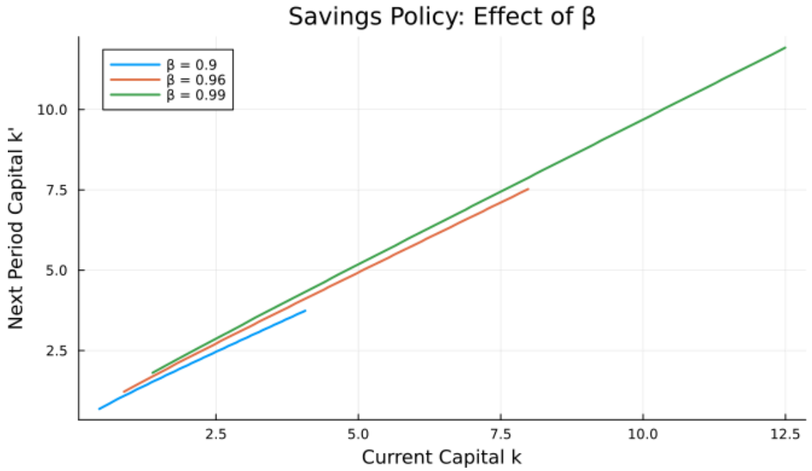
```
plot(legend=:topleft, size=(700, 400), title="Savings Policy: Effect of α")
for α_val in [0.25, 0.33, 0.40]
    m_trial = setup!(RBCModel(α=α_val, N_k=100))
    sol_trial = solveRBC(m_trial, tol=1e-5, verbose=false)
    s_mid = (m_trial.N_z + 1) ÷ 2
    plot!(m_trial.k_grid, sol_trial.kp[:, s_mid], linewidth=2, label="α = $α_val")
end
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
```



- Higher α → capital is more productive → agent saves more → higher steady-state capital

Effect of Discount Factor (β)

```julia
plot(legend=:topleft, size=(700, 400), title="Savings Policy: Effect of β")
for β_val in [0.90, 0.96, 0.99]
    m_trial = setup!(RBCModel(β=β_val, N_k=100))
    sol_trial = solveRBC(m_trial, tol=1e-5, verbose=false)
    s_mid = (m_trial.N_z + 1) ÷ 2
    plot!(m_trial.k_grid, sol_trial.kp[:, s_mid], linewidth=2, label="β = $β_val")
end
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
```



- More **patient** agents (β closer to 1) save more → higher steady-state capital
  - a lower interest rate $r_{ss} = 1/\beta - 1$ lowers the rental rate, so firms demand more capital
- Impatient agents consume more today, accumulate less capital

Linearized Solution

The Euler Equation

Notation

- Define **deviations from steady state**:

$$\hat{k}_t \equiv k_t - k_{ss}, \qquad \hat{c}_t \equiv c_t - c_{ss}, \qquad \hat{z}_t \equiv z_t - 1, \qquad \hat{r}_t \equiv r_t - r_{ss}$$

- We'll linearize by taking a **first-order Taylor expansion** around the steady state, dropping terms like $\hat{k}_t \cdot \hat{z}_t$ (second-order small)

> **Why Linearize?**
>
> VFI is accurate but **slow** ($N_k \times N_z$ optimization problems per iteration). Linearization gives a policy rule that's nearly exact near the steady state — and almost instant to compute.

Linearizing the Resource Constraint

The Resource Constraint

- Recall the aggregate resource constraint:

$$c_t + k_{t+1} = z_t \, k_t^{\alpha} + (1 - \delta) \, k_t$$

- Write each variable as steady state + deviation:

$$(c_{ss} + \hat{c}_t) + (k_{ss} + \hat{k}_{t+1}) = (1 + \hat{z}_t)(k_{ss} + \hat{k}_t)^{\alpha} + (1 - \delta)(k_{ss} + \hat{k}_t)$$

Differentiating the Resource Constraint

- First-order approximation of $(k_{ss} + \hat{k}_t)^\alpha$:

$$(k_{ss} + \hat{k}_t)^\alpha \approx k_{ss}^\alpha + \alpha\, k_{ss}^{\alpha-1}\, \hat{k}_t$$

- Expanding the right side (dropping $\hat{z}_t \cdot \hat{k}_t$):

$$\text{RHS} \approx k_{ss}^\alpha + \underbrace{\alpha k_{ss}^{\alpha-1}}_{r_{ss}+\delta}\hat{k}_t + \underbrace{k_{ss}^\alpha}_{y_{ss}}\hat{z}_t + (1-\delta)k_{ss} + (1-\delta)\hat{k}_t$$

- The steady-state MPK is the firm's rental rate $r_{ss} + \delta$

Linearized Resource Constraint

- Subtracting the steady-state equation ($c_{ss} + k_{ss} = y_{ss} + (1-\delta)k_{ss}$), the constant terms cancel:

$$\hat{c}_t + \hat{k}_{t+1} = \underbrace{(1 + r_{ss})}_{=1/\beta}\hat{k}_t + y_{ss}\,\hat{z}_t$$

$$\boxed{\hat{c}_t = \frac{1}{\beta}\,\hat{k}_t - \hat{k}_{t+1} + y_{ss}\,\hat{z}_t}$$

- This tells us $\hat{c}_t$ as a **linear function** of $\hat{k}_t$, $\hat{k}_{t+1}$, and $\hat{z}_t$
- Used $1 + r_{ss} = 1/\beta$ (the household's steady-state Euler equation)

Linearizing the Euler Equation

The Euler Equation

- Recall the household's Euler equation, with the interest rate from the firm's problem:

$$\frac{1}{c_t} = \beta \mathbb{E}_t \left[\frac{1 + r_{t+1}}{c_{t+1}} \right], \qquad r_{t+1} = \alpha z_{t+1} k_{t+1}^{\alpha - 1} - \delta$$

- **Left side**: marginal cost of saving one more unit (foregone consumption today)
- **Right side**: marginal benefit (discounted expected return, valued in consumption units)
- By the First Welfare Theorem, this is also the planner's first-order condition

The Euler Equation

At the optimum, the household is **indifferent** between consuming one more unit today vs. saving it and consuming the return tomorrow.

Differentiating the Euler Equation

- Linearize **left side**: $\frac{1}{c_{ss} + \hat{c}_t} \approx \frac{1}{c_{ss}} - \frac{\hat{c}_t}{c_{ss}^2}$

- Linearize **right side**: expand $(1 + r_{t+1})/c_{t+1}$ around steady state

$$\frac{1 + r_{ss} + \hat{r}_{t+1}}{c_{ss} + \hat{c}_{t+1}} \approx \frac{1 + r_{ss}}{c_{ss}} + \frac{\hat{r}_{t+1}}{c_{ss}} - \frac{(1 + r_{ss})\,\hat{c}_{t+1}}{c_{ss}^2}$$

Linearized Interest Rate

- Differentiating the firm's condition $r_{t+1} = \alpha z_{t+1} k_{t+1}^{\alpha-1} - \delta$:

$$\hat{r}_{t+1} = \underbrace{\frac{(\alpha - 1)(r_{ss} + \delta)}{k_{ss}}}_{\text{diminishing MPK}} \hat{k}_{t+1} + \underbrace{(r_{ss} + \delta)}_{\text{productivity effect}} \hat{z}_{t+1}$$

- More capital $\rightarrow$ **lower** interest rate (diminishing returns, since $\alpha - 1 < 0$)
- Higher productivity $\rightarrow$ **higher** interest rate
- Depreciation is a constant cost, so it drops out of the deviation

Linearized Euler Equation

- Setting LHS = $\beta\cdot$ RHS and using $\beta(1 + r_{ss}) = 1$ to cancel constant terms:

$$-\frac{\hat{c}_t}{c_{ss}^2} = \frac{\beta}{c_{ss}}\mathbb{E}_t[\hat{r}_{t+1}] - \frac{1}{c_{ss}^2}\mathbb{E}_t[\hat{c}_{t+1}]$$

- Multiply by $-c_{ss}^2$:

$$\boxed{\hat{c}_t = \mathbb{E}_t[\hat{c}_{t+1}] - \beta\, c_{ss}\, \mathbb{E}_t[\hat{r}_{t+1}]}$$

- This is an **intertemporal substitution** equation: when expected interest rates are high ($\hat{r}_{t+1} > 0$), the household **saves more** today ($\hat{c}_t < \mathbb{E}_t[\hat{c}_{t+1}]$)

Solving the Linearized System

Consumption Is Linear in $\hat{k}_t$ and $\hat{z}_t$

- **Guess** a linear policy rule: $\hat{k}_{t+1} = \phi_k \hat{k}_t + \phi_z \hat{z}_t$
- Substitute into the linearized resource constraint $\hat{c}_t = \frac{1}{\beta}\hat{k}_t - \hat{k}_{t+1} + y_{ss}\hat{z}_t$:

$$\hat{c}_t = \underbrace{\left(\frac{1}{\beta} - \phi_k\right)}_{\equiv \psi_k}\hat{k}_t + \underbrace{(y_{ss} - \phi_z)}_{\equiv \psi_z}\hat{z}_t$$

$$\boxed{\hat{c}_t = \psi_k \hat{k}_t + \psi_z \hat{z}_t}$$

- Once we know (ϕ_k, ϕ_z), consumption is **pinned down** by the resource constraint

Expected Future Consumption and Interest Rate

- Using $\mathbb{E}_t[\hat{z}_{t+1}] = \rho\,\hat{z}_t$ and $\hat{k}_{t+1} = \phi_k \hat{k}_t + \phi_z \hat{z}_t$:

$$\mathbb{E}_t[\hat{c}_{t+1}] = \psi_k \underbrace{(\phi_k \hat{k}_t + \phi_z \hat{z}_t)}_{\hat{k}_{t+1}} + \psi_z \underbrace{\rho\,\hat{z}_t}_{\mathbb{E}_t[\hat{z}_{t+1}]}$$

- Define $\gamma \equiv \frac{(\alpha-1)\,(r_{ss}+\delta)}{k_{ss}}$ (how fast the interest rate falls with capital). Then:

$$\mathbb{E}_t[\hat{r}_{t+1}] = \gamma \underbrace{(\phi_k \hat{k}_t + \phi_z \hat{z}_t)}_{\hat{k}_{t+1}} + (r_{ss} + \delta) \underbrace{\rho\,\hat{z}_t}_{\mathbb{E}_t[\hat{z}_{t+1}]}$$

- Both expectations are **linear** in $(\hat{k}_t, \hat{z}_t)$ — so matching coefficients will pin down ϕ

Finding ϕ_k: Match Coefficients on $\hat{k}_t$

- Plug into the Euler equation $\hat{c}_t = \mathbb{E}_t[\hat{c}_{t+1}] - \beta c_{ss}\, \mathbb{E}_t[\hat{r}_{t+1}]$ and collect terms on $\hat{k}_t$:

$$\psi_k = \phi_k\big(\psi_k - \beta\, c_{ss}\, \gamma\big)$$

- Since $\psi_k = \frac{1}{\beta} - \phi_k$, this is **one equation in one unknown** (ϕ_k):

$$\boxed{\frac{1}{\beta} - \phi_k = \phi_k\left(\frac{1}{\beta} - \phi_k - \beta\, c_{ss}\, \gamma\right)}$$

- This is a **quadratic** in ϕ_k — pick the stable root ($|\phi_k| < 1$)

Finding ϕ_z: Match Coefficients on $\hat{z}_t$

- Once ϕ_k is known ($\Rightarrow \psi_k$ is known), collect terms on $\hat{z}_t$:

$$\psi_z = \psi_k \phi_z + \psi_z \rho - \beta\, c_{ss} \left(\gamma \phi_z + (r_{ss} + \delta)\, \rho \right)$$

- Substitute $\psi_z = y_{ss} - \phi_z$ and collect ϕ_z terms:

$$\boxed{\phi_z = \frac{y_{ss}(1-\rho) + \beta\, c_{ss}\, (r_{ss} + \delta)\, \rho}{\psi_k - \beta\, c_{ss}\, \gamma + (1-\rho)}}$$

- This is a **linear** equation — just arithmetic once ϕ_k is known

Summary of the Approach

Summary of the Approach

1. Guess $\hat{k}_{t+1} = \phi_k \hat{k}_t + \phi_z \hat{z}_t \rightarrow$ consumption $\hat{c}_t = \psi_k \hat{k}_t + \psi_z \hat{z}_t$ follows from the RC
2. Compute $\mathbb{E}_t[\hat{c}_{t+1}]$ and $\mathbb{E}_t[\hat{r}_{t+1}]$ as linear functions of $(\hat{k}_t, \hat{z}_t)$
3. Match $\hat{k}_t$ coefficients in the Euler equation $\rightarrow$ **quadratic** for ϕ_k
4. Match $\hat{z}_t$ coefficients $\rightarrow$ **linear** equation for ϕ_z

Julia Implementation

Solving for ϕ_k

- Use `solve1D` from the course package `EC410` to solve the ϕ_k equation, then compute ϕ_z directly:

```
using EC410
```

```
function linearize_rbc(m)
    (; α, β, δ, ρ) = m
    ss_vals = steady_state(m)
    k_ss = ss_vals.k;  y_ss = ss_vals.y;  c_ss = ss_vals.c;  r_ss = ss_vals.r
    γ = (α - 1) * (r_ss + δ) / k_ss                  # slope of r in k

    # Step 1: solve for ϕ_k  (ψ_k = ϕ_k·(ψ_k − β·c_ss·γ) with ψ_k = 1/β − ϕ_k)
    ϕk_residual(ϕ_k) = (1/β - ϕ_k) - ϕ_k * (1/β - ϕ_k - β * c_ss * γ)
    ϕ_k = solve1D(ϕk_residual, 0.0, 1.0)            # stable root lies in (0, 1)

    # Step 2: compute ϕ_z directly
    ψ_k = 1/β - ϕ_k
    ϕ_z = (y_ss*(1 - ρ) + β*c_ss*(r_ss + δ)*ρ) / (ψ_k - β*c_ss*γ + (1 - ρ))

    return (ϕ_k=ϕ_k, ϕ_z=ϕ_z)
end
```

```
linearize_rbc (generic function with 1 method)
```

- `solve1D(f, a, b)` finds the root of `f` **between** `a` and `b`
  - the residual is positive at 0 ($= 1/\beta$) and negative at 1 ($= \beta c_{ss} \gamma < 0$), so the bracket $[0, 1]$ captures the **stable** root

The Linearized Policy Rule

```
lin = linearize_rbc(m)
println("ϕ_k = $(round(lin.ϕ_k, digits=4))")
println("ϕ_z = $(round(lin.ϕ_z, digits=4))")
```

```
ϕ_k = 0.8784
ϕ_z = 1.0427
```

- The linearized capital policy:

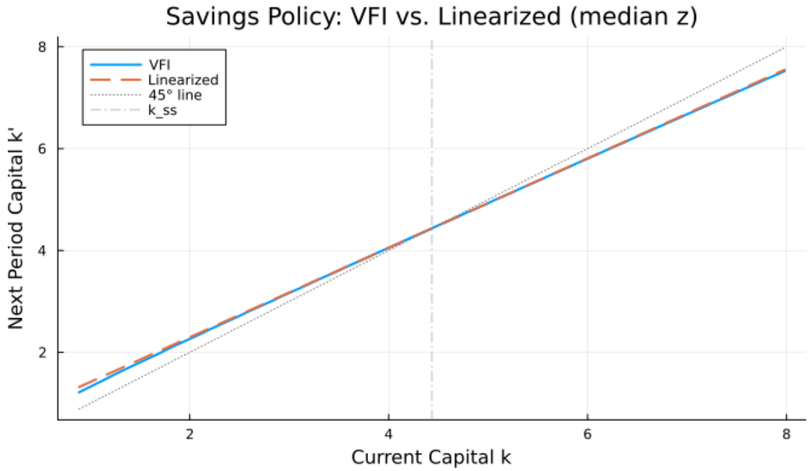
$$k_{t+1} \approx k_{ss} + \phi_k \left(k_t - k_{ss}\right) + \phi_z \left(z_t - 1\right)$$

- $|\phi_k| < 1$: capital **mean-reverts** toward the steady state
- $\phi_z > 0$: higher productivity $\rightarrow$ **save more**

Comparing to VFI

Policy Functions

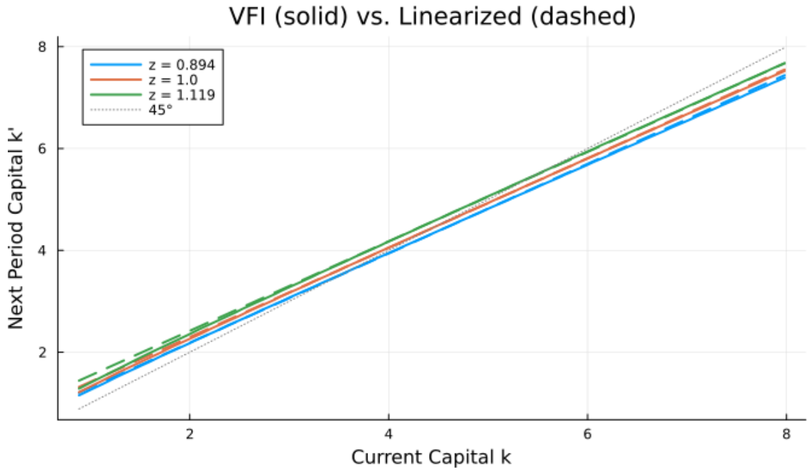
```
s_mid = (m.N_z + 1) ÷ 2
kp_linear = ss.k .+ lin.ϕ_k .* (m.k_grid .- ss.k) .+ lin.ϕ_z .* (m.z_grid[s_mid] - 1.0)

plot(m.k_grid, sol.kp[:, s_mid], linewidth=2, label="VFI",
     legend=:topleft, size=(700, 400))
plot!(m.k_grid, kp_linear, linewidth=2, linestyle=:dash, label="Linearized")
plot!(m.k_grid, m.k_grid, color=:gray, linestyle=:dot, label="45° line")
vline!([ss.k], color=:gray, linestyle=:dashdot, label="k_ss", alpha=0.5)
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
title!("Savings Policy: VFI vs. Linearized (median z)")
```



- Near k_{ss}: the two solutions **nearly coincide**
- Far from k_{ss}: the linearized solution diverges — it can't capture curvature

Policy Functions Across Productivity Levels

```julia
plot(legend=:topleft, size=(700, 400), title="VFI (solid) vs. Linearized (dashed)")
for s in [1, 4, 7]
    z = m.z_grid[s]
    kp_lin = ss.k .+ lin.ϕ_k .* (m.k_grid .- ss.k) .+ lin.ϕ_z .* (z - 1.0)
    plot!(m.k_grid, sol.kp[:, s], linewidth=2,
          label="z = $(round(z, digits=3))", color=palette(:default)[s ÷ 3 + 1])
    plot!(m.k_grid, kp_lin, linewidth=2, linestyle=:dash,
          label="", color=palette(:default)[s ÷ 3 + 1])
end
plot!(m.k_grid, m.k_grid, color=:gray, linestyle=:dot, label="45°")
xlabel!("Current Capital k")
ylabel!("Next Period Capital k'")
```



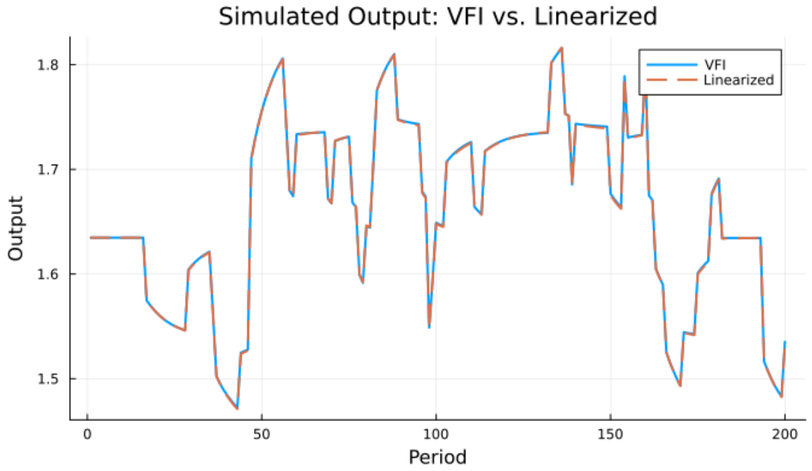
Simulating the Linearized Model

```julia
function simulateRBC_linear(m, lin, T; seed=42)
    Random.seed!(seed)
    (; α, δ, z_grid, P_z, N_z) = m
    k_ss = steady_state(m).k
    k_sim = zeros(T);  s_sim = zeros(Int, T)
    y_sim = zeros(T);  c_sim = zeros(T);  i_sim = zeros(T)
    k_sim[1] = k_ss
    s_sim[1] = (N_z + 1) ÷ 2
    for t in 1:T
        z = z_grid[s_sim[t]]
        kp = k_ss + lin.ϕ_k * (k_sim[t] - k_ss) + lin.ϕ_z * (z - 1.0)
        y_sim[t] = z * k_sim[t]^α
        c_sim[t] = y_sim[t] + (1-δ)*k_sim[t] - kp
        i_sim[t] = kp - (1-δ)*k_sim[t]
        if t < T
            k_sim[t+1] = kp
            s_sim[t+1] = rand(Categorical(P_z[s_sim[t], :]))
        end
```

```
simulateRBC_linear (generic function with 1 method)
```

Comparing Simulated Output

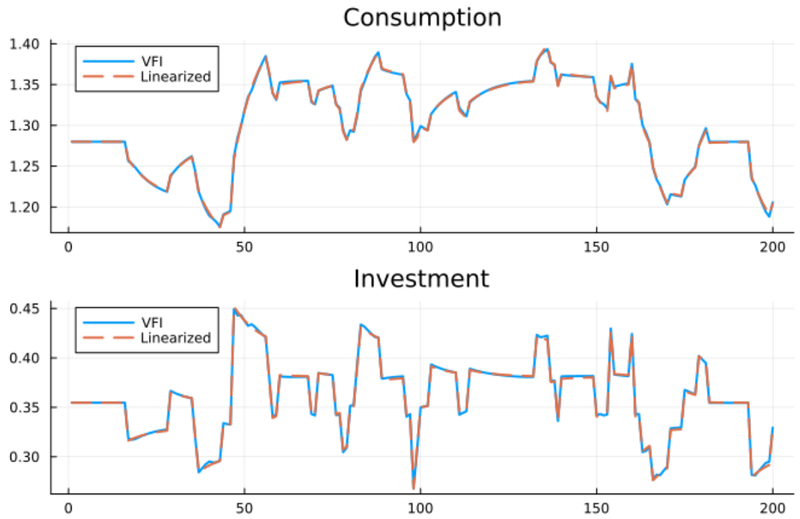
```
sim_vfi = simulateRBC(m, sol, 200)
sim_lin = simulateRBC_linear(m, lin, 200)

plot(sim_vfi.y, linewidth=2, label="VFI", legend=:topright, size=(700, 400))
plot!(sim_lin.y, linewidth=2, linestyle=:dash, label="Linearized")
xlabel!("Period")
ylabel!("Output")
title!("Simulated Output: VFI vs. Linearized")
```



- The two simulations track each other **very closely**
- Near the steady state, linearization is an excellent approximation

Comparing Simulated Consumption and Investment

```
p1 = plot(sim_vfi.c, linewidth=2, label="VFI", title="Consumption", size=(700, 200))
plot!(p1, sim_lin.c, linewidth=2, linestyle=:dash, label="Linearized")
p2 = plot(sim_vfi.i, linewidth=2, label="VFI", title="Investment", size=(700, 200))
plot!(p2, sim_lin.i, linewidth=2, linestyle=:dash, label="Linearized")
plot(p1, p2, layout=(2,1), size=(700, 450))
```



Linearization: Pros and Cons

Pros: Almost instant to compute, gives closed-form policy rules, easy to analyze theoretically.

Cons: Only accurate near the steady state, misses nonlinearities (e.g., precautionary savings, occasionally binding constraints).

Most of the macro literature uses linearization (or higher-order perturbation). VFI is the gold standard for accuracy but is computationally demanding.

Summary

Key Takeaways

Concepts

- Real Business Cycle theory
- Decentralization: households earn r_t, firms pay $r_t + \delta$
- Steady state and transition dynamics without shocks
- Calibration: matching long-run moments and the Solow residual
- Social planner's Bellman equation, deterministic and stochastic
- Linear interpolation for continuous states
- Rouwenhorst discretization of AR(1) processes
- Business cycle statistics: volatilities, correlations
- Linearization via method of undetermined coefficients and root solving

Julia Tools

- `Interpolations.jl`: `linear_interpolation` with call syntax
- `Optim.jl`: `Brent()` for 1D optimization
- `EC410`: `solve1D(f, a, b)` for bracketed root finding
- `DataFrames.jl`: tabular simulation output
- Mutable structs with `setup!` and `(; ...) = m` destructuring
- Multiple dispatch: one function name, a method per model type
- `Categorical` for Markov simulation
- Broadcasting for element-wise operations