Computational Economics Lecture 5: Intro to Iteration Methods I: Infinite Periods

Min Fang

University of Florida

Spring 2025

Outline

1. **Motivation**
2. **Value Function Iteration**
3. **Policy Function Iteration**
4. **Endogenous Grid Method**

Motivation

- We are often facing a dynamic programming (DP) problem in economics
- Often discrete DPs in IO/labor type of applications, while continuous DPs in Macro
- Assuming contraction mapping conditions are satisfied, we can solve by iteration
- We will introduce three methods today and some techniques in practice
- Please read the handout by Prof. Jesus Fernandez-Villaverde: Lecture on DP
- VFI: Chapter 34. Optimal Growth I: The Stochastic Optimal Growth Model
- PFI: Chapter 35. Optimal Growth II: Time Iteration
- EGM: Chapter 36. Optimal Growth III: The Endogenous Grid Method
- We will also talk a bit about finite period iteration in the next lecture
  - The idea of MIT shock and its applications
  - Backward iteration for forward-looking problems

The Stochastic Optimal Growth Model

- We will always use the stochastic optimal growth model as the benchmark
- Simple setup to maximize welfare by using the following Bellman Equation

$$w(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+)$$

  where

$$y_{t+1} = f(y_t - c_t)\xi_{t+1} \quad \text{and} \quad 0 \leq c_t \leq y_t \quad \text{for all } t$$

$$\{\xi_t\} \text{ is assumed to be IID}$$

- The policy value function v_σ associated with a given policy σ is defined by

$$v_\sigma(y) = \mathbb{E}\left[\sum_{t=0}^{\infty} \beta^t u(\sigma(y_t))\right]$$

- The value function is then defined as

$$v^*(y) := \sup_{\sigma \in \Sigma} v_\sigma(y)$$

Value Function Iteration: The Bellman Operator

- How, then, should we compute the value function? Use the so-called Bellman operator
- The Bellman operator is denoted by T and defined by

$$Tw(y) := \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int w(f(y-c)z)\phi(dz) \right\} \qquad (y \in \mathbb{R}_+)$$

- The solutions to the Bellman equation exactly coincide with the fixed points of T
- For example, if $Tw = w$, then, for any $y \geq 0$,

$$w(y) = Tw(y) = \max_{0 \leq c \leq y} \left\{ u(c) + \beta \int v^*(f(y-c)z)\phi(dz) \right\}$$

- It says precisely that w is a solution to the Bellman equation
- It follows that v^* is a fixed point of T

Value Function Iteration: Computation

- Implementing the Bellman operator using linear interpolation

```julia
function T(w; p, tol = 1e-10)
    (; beta, u, f, Xi, y) = p # unpack parameters
    w_func = LinearInterpolation(y, w)

    Tw = similar(w)
    sigma = similar(w)
    for (i, y_val) in enumerate(y)
        # solve maximization for each point in y, using y itself as initial condition.
        results = maximize(c -> u(c; p) +
                                beta * mean(w_func.(f(y_val - c; p) .* Xi)),
                           tol, y_val)
        Tw[i] = maximum(results)
        sigma[i] = maximizer(results)
    end
    return (; w = Tw, sigma) # returns named tuple of results
end
```

Value Function Iteration: An Example with Analytical Solution

- Setup: $f(k) = k^\alpha$, $u(c) = \ln c$, ϕ is the distribution of $\exp(\mu + \sigma\zeta)$ when $\zeta \sim N(0,1)$
- For this particular problem, an exact analytical solution is available with

$$v^*(y) = \frac{\ln(1-\alpha\beta)}{1-\beta} + \frac{(\mu + \alpha\ln(\alpha\beta))}{1-\alpha}\left[\frac{1}{1-\beta} - \frac{1}{1-\alpha\beta}\right] + \frac{1}{1-\alpha\beta}\ln y$$

- The optimal consumption policy is

$$\sigma^*(y) = (1-\alpha\beta)y$$

Value Function Iteration: Code the Setup

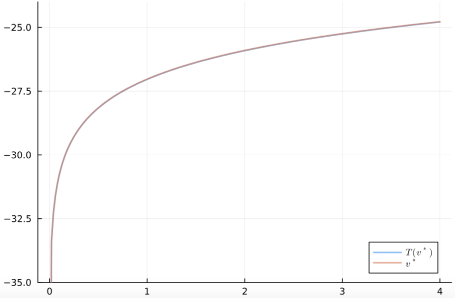
In addition to the model parameters, we need a grid and some shock draws for Monte Carlo integration.

```julia
Random.seed!(42) # for reproducible results
u(c; p) = log(c) # utility
f(k; p) = k^p.alpha # deterministic part of production function
function OptimalGrowthModel(; alpha = 0.4, beta = 0.96, mu = 0.0, s = 0.1,
                            u = u, f = f, # defaults defined above
                            y = range(1e-5, 4.0, length = 200), # grid on y
                            Xi = exp.(mu .+ s * randn(250)))
    return (; alpha, beta, mu, s, u, f, y, Xi)
end # named tuples defaults

# True value and policy function
function v_star(y; p)
    (; alpha, mu, beta) = p
    c1 = log(1 - alpha * beta) / (1 - beta)
    c2 = (mu + alpha * log(alpha * beta)) / (1 - alpha)
    c3 = 1 / (1 - beta)
    c4 = 1 / (1 - alpha * beta)
    return c1 + c2 * (c3 - c4) + c4 * log(y)
end
c_star(y; p) = (1 - p.alpha * p.beta) * y
```

Value Function Iteration: Test 1

Test 1: See what happens when we apply our Bellman operator to the exact solution v^*

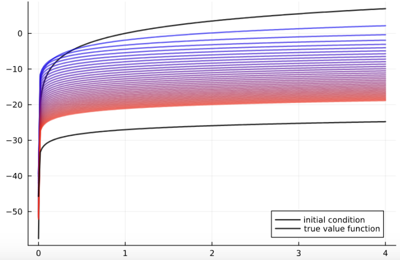
```
p = OptimalGrowthModel() # use all default parameters from named tuple
w_star = v_star.(p.y; p)  # evaluate closed form value along grid

w = T(w_star; p).w # evaluate operator, access Tw results

plt = plot(ylim = (-35, -24))
plot!(plt, p.y, w, linewidth = 2, alpha = 0.6, label = L"T(v^*)")
plot!(plt, p.y, w_star, linewidth = 2, alpha = 0.6, label = L"v^*")
plot!(plt, legend = :bottomright)
```



Value Function Iteration: Test 2

Test 2: See what happens starting off from an arbitrary initial condition

```julia
w = 5 * log.(p.y)  # An initial condition -- fairly arbitrary
n = 35

plot(xlim = (extrema(p.y)), ylim = (-50, 10))
lb = "initial condition"
plt = plot(p.y, w, color = :black, linewidth = 2, alpha = 0.8, label = lb)
for i in 1:n
    w = T(w; p).w
    plot!(p.y, w, color = RGBA(i / n, 0, 1 - i / n, 0.8), linewidth = 2,
          alpha = 0.6,
          label = "")
end

lb = "true value function"
plot!(plt, p.y, v_star.(p.y; p), color = :black, linewidth = 2, alpha = 0.8,
      label = lb)
plot!(plt, legend = :bottomright)
```



Policy Function Iteration: Setup

- In some situations, we could solve the problem with policy function iteration
  - u and f are continuously differentiable and strictly concave with $f(0) = 0$
  - $\lim_{c \to 0} u'(c) = \infty$ and $\lim_{c \to \infty} u'(c) = 0$
  - $\lim_{k \to 0} f'(k) = \infty$ and $\lim_{k \to \infty} f'(k) = 0$
- As a result, the value function is strictly concave and continuously differentiable

$$(v^*)'(y) = u'(c^*(y)) := (u' \circ c^*)(y)$$

- Differentiability of the value function and interiority of the optimal policy:

$$v^*(y) = \max_{0 \le k \le y} \left\{ u(y - k) + \beta \int v^*(f(k)z)\phi(dz) \right\}$$

$$u'(c^*(y)) = \beta \int (v^*)'(f(y - c^*(y))z) f'(y - c^*(y)) z \phi(dz)$$

- We could then derive the Euler equation (and then as a functional equation)

$$(u' \circ c^*)(y) = \beta \int (u' \circ c^*)(f(y - c^*(y))z) f'(y - c^*(y)) z \phi(dz)$$

$$(u' \circ \sigma)(y) = \beta \int (u' \circ \sigma)(f(y - \sigma(y))z) f'(y - \sigma(y)) z \phi(dz)$$

Policy Function Iteration: The Coleman Operator

- This operator K will act on the set of all $\sigma \in \Sigma$ that is continuous, strictly increasing, and interior (i.e., $0 < \sigma(y) < y$ for all strictly positive y)
- Henceforth we denote this set of policies by $\mathscr{P}$
    - The operator K takes as its argument a $\sigma \in \mathscr{P}$
    - Returns a new function $K\sigma$, where $K\sigma(y)$ is the $c \in (0, y)$ that solves

$$u'(c) = \beta \int (u' \circ \sigma)(f(y-c)z)f'(y-c)z\phi(dz)$$

- The optimal policy c^* is a fixed point that solves

$$u'(c) = \beta \int (u' \circ c^*)(f(y-c)z)f'(y-c)z\phi(dz)$$

- In this specific case, the Coleman operator is well-defined
- In this specific case, it is an identical object to the Bellman operator

Policy Function Iteration: Computation

- Implementing the Coleman operator using linear interpolation

```julia
using LinearAlgebra, Statistics
using BenchmarkTools, Interpolations, LaTeXStrings,  Plots, Roots
using Optim, Random
```

```julia
using BenchmarkTools, Interpolations, Plots, Roots
```

```julia
function K!(Kg, g, grid, beta, dudc, f, f_prime, shocks)
    # This function requires the container of the output value as argument Kg

    # Construct linear interpolation object
    g_func = LinearInterpolation(grid, g, extrapolation_bc = Line())

    # solve for updated consumption value
    for (i, y) in enumerate(grid)
        function h(c)
            vals = dudc.(g_func.(f(y - c) * shocks)) .* f_prime(y - c) .* shocks
            return dudc(c) - beta * mean(vals)
        end
        Kg[i] = find_zero(h, (1e-10, y - 1e-10))
    end
    return Kg
end

# The following function does NOT require the container of the output value as argument
function K(g, grid, beta, dudc, f, f_prime, shocks)
    K!(similar(g), g, grid, beta, dudc, f, f_prime, shocks)
end
```

Policy Function Iteration: An Example

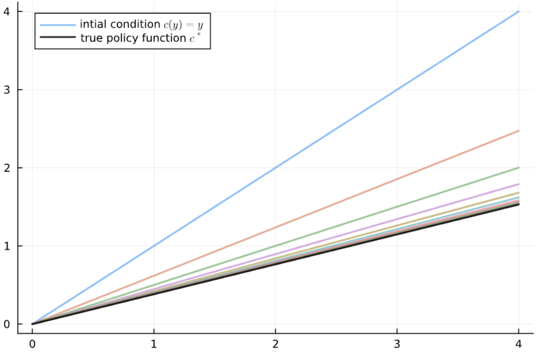
- Implementing the Coleman operator on the same example

```
isoelastic(c, gamma) = isone(gamma) ? log(c) : (c^(1 - gamma) - 1) / (1 - gamma)
function Model(; alpha = 0.65,                              # Productivity parameter
               beta = 0.95,                                 # Discount factor
               gamma = 1.0,                                  # Risk aversion
               mu = 0.0,                                   # First parameter in lognorm(mu, s
               s = 0.1,                                   # Second parameter in lognorm(mu, s
               grid = range(1e-6, 4, length = 200), # Grid
               grid_min = 1e-6,                           # Smallest grid point
               grid_max = 4.0,                            # Largest grid point
               grid_size = 200,                           # Number of grid points
               u = (c, gamma = gamma) -> isoelastic(c, gamma),  # utility function
               dudc = c -> c^(-gamma),                       # u_prime
               f = k -> k^alpha,                             # production function
               f_prime = k -> alpha * k^(alpha - 1))
    return (; alpha, beta, gamma, mu, s, grid, grid_min, grid_max, grid_size, u,
            dudc, f, f_prime)
end
```

Policy Function Iteration: Test 1

- Test 1: Try iterating from an arbitrary initial condition and see if we converge

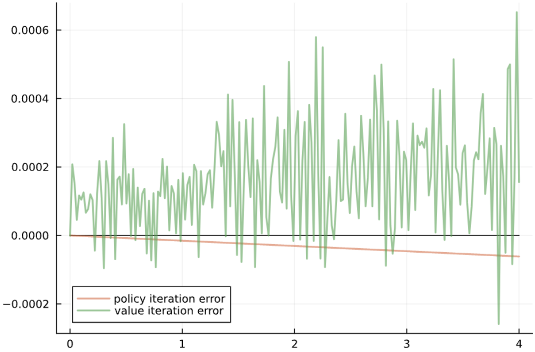
```
check_convergence(m, shocks, c_star, m.grid, n_iter = 15)
```



Policy Function Iteration: Test 2

- Test 2: Compare the accuracy of iteration using the Coleman and Bellman operators

```
compare_error(m, shocks, m.grid, m.u.(m.grid), sim_length = 20)
```



Endogenous Grid Method: Motivation

- Can we further improve on PFI on speed? Yes, by using the endogenous grid method
- First, let's talk about our current exogenous grid method of numerical approximation
- Represent a policy function by a set of values on a finite grid with interpolation
- To obtain a finite representation of an updated consumption policy, we
    - fixed a grid of income points $\{y_i\}$
    - calculated the consumption value c_i corresponding to each y_i with a root-finding routine
- Each c_i is then interpreted as the value of the function Kg at y_i
- Thus, with the points $\{y_i, c_i\}$ in hand, we can reconstruct Kg via approximation
- Iteration then continues…
- Cons: The root-finding routine to find the c_i corresponding to a given income value y_i is costly because it involves a significant number of function evaluations!

Endogenous Grid Method: Kill the Root-finding

- Simple, we can avoid this if y_i is chosen endogenously!
- The only assumption required is that u' is invertible on $(0, \infty)$.
- First we fix an exogenous grid $\{k_i\}$ for capital ($k = y - c$).
- Then we obtain c_i via

$$c_i = (u')^{-1} \left\{ \beta \int (u' \circ g)(f(k_i) z) \, f'(k_i) \, z \, \phi(dz) \right\}$$

  where $(u')^{-1}$ is the inverse function of u'

- Finally, for each c_i we set $y_i = c_i + k_i$
- It is clear that each (y_i, c_i) pair constructed in this manner satisfies the above equation
- With the points $\{y_i, c_i\}$ in hand, we can reconstruct Kg via approximation as before
- The name EGM comes from the fact that the grid $\{y_i\}$ is determined endogenously

Endogenous Grid Method: Computation

- Implementing the Coleman operator using EGM

```julia
using LinearAlgebra, Statistics
using BenchmarkTools, Interpolations, LaTeXStrings, Plots, Random, Roots
```

```julia
function coleman_egm(g, k_grid, beta, u_prime, u_prime_inv, f, f_prime, shocks)

    # Allocate memory for value of consumption on endogenous grid points
    c = similar(k_grid)

    # Solve for updated consumption value
    for (i, k) in enumerate(k_grid)
        vals = u_prime.(g.(f(k) * shocks)) .* f_prime(k) .* shocks
        c[i] = u_prime_inv(beta * mean(vals))
    end

    # Determine endogenous grid
    y = k_grid + c  # y_i = k_i + c_i

    # Update policy function and return
    Kg = LinearInterpolation(y, c, extrapolation_bc = Line())
    Kg_f(x) = Kg(x)
    return Kg_f
end
```

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

Endogenous Grid Method: An Example

- Implementing the Coleman operator on the same example

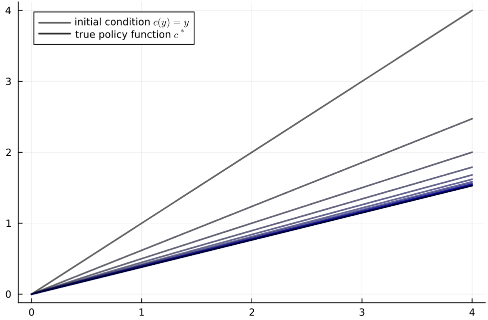
```julia
# model

function Model(; alpha = 0.65, # productivity parameter
                beta = 0.95, # discount factor
                gamma = 1.0,  # risk aversion
                mu = 0.0,  # lognorm(mu, sigma)
                s = 0.1,  # lognorm(mu, sigma)
                grid_min = 1e-6, # smallest grid point
                grid_max = 4.0,  # largest grid point
                grid_size = 200, # grid size
                u = gamma == 1 ? log : c -> (c^(1 - gamma) - 1) / (1 - gamma), # utility
                u_prime = c -> c^(-gamma), # u'
                f = k -> k^alpha, # production function
                f_prime = k -> alpha * k^(alpha - 1), # f'
                grid = range(grid_min, grid_max, length = grid_size)) # grid
    return (; alpha, beta, gamma, mu, s, grid_min, grid_max, grid_size, u,
            u_prime,
            f, f_prime, grid)
end
```

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

Endogenous Grid Method: Test 1

- Test 1: Convergence

```
check_convergence(mlog, shocks, c_star, identity, n)
```



We see that the policy has converged nicely, in only a few steps.

Endogenous Grid Method: Test 2

- Test 2: Speed

```julia
@benchmark coleman($mcrra)
```

```
BenchmarkTools.Trial: 3 samples with 1 evaluation.
 Range (min … max):  2.199 s …   2.250 s  ┊ GC (min … max): 1.60% … 1.88%
 Time  (median):     2.209 s              ┊ GC (median):    1.76%
 Time  (mean ± σ):   2.219 s ± 27.048 ms  ┊ GC (mean ± σ):  1.75% ± 0.14%

  2.2 s          Histogram: frequency by time        2.25 s <

 Memory estimate: 922.71 MiB, allocs estimate: 937680.
```

```julia
@benchmark egm($mcrra)
```

```
BenchmarkTools.Trial: 74 samples with 1 evaluation.
 Range (min … max):  65.351 ms … 86.277 ms  ┊ GC (min … max): 0.00% … 17.90%
 Time  (median):     66.640 ms              ┊ GC (median):    1.78%
 Time  (mean ± σ):   67.859 ms ±  3.341 ms  ┊ GC (mean ± σ):  2.02% ±  2.65%

  65.4 ms         Histogram: frequency by time         75.6 ms <

 Memory estimate: 20.22 MiB, allocs estimate: 120245.
```

We see that the EGM version is about 30 times faster.