Lecture 8: Value Function Iteration

Jacob Adenbaum

University of Edinburgh

Spring 2024

Dynamic Programs are Everywhere

- Dynamic problems show up everywhere in Economics
- You find one anytime an agent is choosing how to trade off a reward today against waiting for tomorrow. E.g.
  - Saving for tomorrow vs. consuming today
  - How much costly effort to put into a job search
  - Should a bus mechanic repair the engine today, or wait until next month?
- The trouble is that they are extremely difficult to solve
- In general, pen-and-paper solutions don't exist, and we have to solve them on a computer

Why do we want to solve them?

Roadmap for the future

- The main goal we have is to be able to **simulate fake data** from our models
  - For that, we need to solve for the optimal policy rule that our agents have under any situation that could arise
  - Then we just randomly simulate the shocks, and step our simulated agents forward using their policy rules

    Don't worry if this doens't make much sense right now. I'll be much more precise about this next week
- Once we can simulate data from our model, we can study the model's predictions under various parameters
- **Estimation:** We can choose parameters that make our model's simulated data match the real data
- **Policy Experiments:** We can change government policy, and see how agents' behavior changes.

  We can figure out what is the optimal policy

Section 1

Finite Horizon Dynamic Programs

The neoclassical growth model in 3 periods

- Suppose we take the standard neoclassical growth model with only three periods.
  - Households choose between consuming and investing in the capital stock
  - Capital depreciation at rate δ, and initial capital k_1
  - Flow utility $u(c)$ and production function $F_t(k) = A_t k^\alpha$
- We can write this problem with a period by period budget constraint:

$$
\begin{aligned}
\mathbf{v_1(k_1)} := \max_{c_1, c_2, c_3, k_2, k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\
\text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha + (1-\delta)k_1 \quad \text{Period 1 BC} \\
& c_2 + k_3 \leq A_2 k_2^\alpha + (1-\delta)k_2 \quad \text{Period 2 BC} \\
& c_3 \leq A_3 k_3^\alpha + (1-\delta)k_3 \quad \text{Period 3 BC}
\end{aligned} \tag{1}
$$

- If given $u(c), \delta, \beta$ and $\{A_t\}$, you know how to put this on a computer and solve it:
  - Sequentially substitute out budget constraints and maximize over the variables k_1 and k_2
  - You did something like this in an earlier problem set
- Call the maximized value $v_1(k_1)$

The neoclassical growth model in 3 periods

- Suppose we take the standard neoclassical growth model with only three periods.
  - Households choose between consuming and investing in the capital stock
  - Capital depreciation at rate δ, and initial capital k_1
  - Flow utility $u(c)$ and production function $F_t(k) = A_t k^\alpha$
- We can write this problem with a period by period budget constraint:

$$\begin{aligned} \mathbf{v_1(k_1)} := \max_{c_1,c_2,c_3,k_2,k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha + (1-\delta)k_1 \quad \text{Period 1 BC} \\ & c_2 + k_3 \leq A_2 k_2^\alpha + (1-\delta)k_2 \quad \text{Period 2 BC} \\ & c_3 \leq A_3 k_3^\alpha + (1-\delta)k_3 \quad \text{Period 3 BC} \end{aligned} \tag{1}$$

- If given $u(c), \delta, \beta$ and $\{A_t\}$, you know how to put this on a computer and solve it:
  - Sequentially substitute out budget constraints and maximize over the variables k_1 and k_2
  - You did something like this in an earlier problem set
- Call the maximized value $v_1(k_1)$

The neoclassical growth model in 3 periods

- Suppose we take the standard neoclassical growth model with only three periods.
  - Households choose between consuming and investing in the capital stock
  - Capital depreciation at rate δ, and initial capital k_1
  - Flow utility $u(c)$ and production function $F_t(k) = A_t k^\alpha$
- We can write this problem with a period by period budget constraint:

$$\begin{aligned} \mathbf{v_1(k_1)} := \max_{c_1,c_2,c_3,k_2,k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\ \text{s.t.} \quad & c_1 + k_2 \le A_1 k_1^\alpha + (1-\delta)k_1 \quad \text{Period 1 BC} \\ & c_2 + k_3 \le A_2 k_2^\alpha + (1-\delta)k_2 \quad \text{Period 2 BC} \\ & c_3 \le A_3 k_3^\alpha + (1-\delta)k_3 \quad \text{Period 3 BC} \end{aligned} \tag{1}$$

- If given $u(c), \delta, \beta$ and $\{A_t\}$, you know how to put this on a computer and solve it:
  - Sequentially substitute out budget constraints and maximize over the variables k_1 and k_2
  - You did something like this in an earlier problem set
- Call the maximized value $v_1(k_1)$

The neoclassical growth model in 3 periods

- Suppose we take the standard neoclassical growth model with only three periods.
  - Households choose between consuming and investing in the capital stock
  - Capital depreciation at rate δ, and initial capital k_1
  - Flow utility $u(c)$ and production function $F_t(k) = A_t k^\alpha$
- We can write this problem with a period by period budget constraint:

$$
\begin{aligned}
\mathbf{v_1(k_1)} := \max_{c_1,c_2,c_3,k_2,k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\
\text{s.t.} \quad & c_1 + k_2 \le A_1 k_1^\alpha + (1-\delta)k_1 \quad \text{Period 1 BC} \\
& c_2 + k_3 \le A_2 k_2^\alpha + (1-\delta)k_2 \quad \text{Period 2 BC} \\
& c_3 \le A_3 k_3^\alpha + (1-\delta)k_3 \quad \text{Period 3 BC}
\end{aligned}
\tag{1}
$$

- If given $u(c), \delta, \beta$ and $\{A_t\}$, you know how to put this on a computer and solve it:
  - Sequentially substitute out budget constraints and maximize over the variables k_1 and k_2
  - You did something like this in an earlier problem set
- Call the maximized value $v_1(k_1)$

Multi-stage budgeting

- This will work for T periods when T is small, but it doesn't generalize well...
- Instead, let's split the problem up into multiple stages, each of which looks easier to solve
- Define $v_3(k_3)$ as the value you get from starting period 3 with a capital stock k_3:

$$\begin{aligned} v_3(k_3) = \max_{c_3} \quad & u(c_3) \\ \text{s.t.} \quad & c_3 \leq A_3 k_3^\alpha + (1-\delta)k_3 \end{aligned} \tag{2}$$

- And define $v_2(k_2)$ as the value you get from starting period 2 with a capital stock k_2:

$$\begin{aligned} v_2(k_2) = \max_{c_2, k_3} \quad & u(c_2) + \beta v_3(k_3) \\ \text{s.t.} \quad & c_2 + k_3 \leq A_2 k_2^\alpha + (1-\delta)k_2 \end{aligned} \tag{3}$$

  - Notice that we now have a **continuation value**

- Finally, we can rewrite the first period problem as

$$\begin{aligned} v_1(k_1) = \max_{c_1, k_1} \quad & u(c_1) + \beta v_2(k_2) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha + (1-\delta)k_1 \end{aligned} \tag{4}$$

Multi-stage budgeting

- This will work for T periods when T is small, but it doesn't generalize well...
- Instead, let's split the problem up into multiple stages, each of which looks easier to solve
- Define $v_3(k_3)$ as the value you get from starting period 3 with a capital stock k_3:

$$\begin{aligned} v_3(k_3) = \max_{c_3} \quad & u(c_3) \\ \text{s.t.} \quad & c_3 \leq A_3 k_3^\alpha + (1-\delta)k_3 \end{aligned} \tag{2}$$

- And define $v_2(k_2)$ as the value you get from starting period 2 with a capital stock k_2:

$$\begin{aligned} v_2(k_2) = \max_{c_2, k_3} \quad & u(c_2) + \beta v_3(k_3) \\ \text{s.t.} \quad & c_2 + k_3 \leq A_2 k_2^\alpha + (1-\delta)k_2 \end{aligned} \tag{3}$$

  - Notice that we now have a **continuation value**

- Finally, we can rewrite the first period problem as

$$\begin{aligned} v_1(k_1) = \max_{c_1, k_1} \quad & u(c_1) + \beta v_2(k_2) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha + (1-\delta)k_1 \end{aligned} \tag{4}$$

Multi-stage budgeting

- This will work for T periods when T is small, but it doesn't generalize well...
- Instead, let's split the problem up into multiple stages, each of which looks easier to solve
- Define $v_3(k_3)$ as the value you get from starting period 3 with a capital stock k_3:

$$\begin{aligned} v_3(k_3) = \max_{c_3} \quad & u(c_3) \\ \text{s.t.} \quad & c_3 \leq A_3 k_3^{\alpha} + (1-\delta)k_3 \end{aligned} \tag{2}$$

- And define $v_2(k_2)$ as the value you get from starting period 2 with a capital stock k_2:

$$\begin{aligned} v_2(k_2) = \max_{c_2, k_3} \quad & u(c_2) + \beta v_3(k_3) \\ \text{s.t.} \quad & c_2 + k_3 \leq A_2 k_2^{\alpha} + (1-\delta)k_2 \end{aligned} \tag{3}$$

  - Notice that we now have a **continuation value**
- Finally, we can rewrite the first period problem as

$$\begin{aligned} v_1(k_1) = \max_{c_1, k_1} \quad & u(c_1) + \beta v_2(k_2) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^{\alpha} + (1-\delta)k_1 \end{aligned} \tag{4}$$

Multi-stage budgeting

- This will work for T periods when T is small, but it doesn't generalize well...
- Instead, let's split the problem up into multiple stages, each of which looks easier to solve
- Define $v_3(k_3)$ as the value you get from starting period 3 with a capital stock k_3:

$$\begin{aligned} v_3(k_3) = \max_{c_3} \quad & u(c_3) \\ \text{s.t.} \quad & c_3 \leq A_3 k_3^\alpha + (1-\delta)k_3 \end{aligned} \tag{2}$$

- And define $v_2(k_2)$ as the value you get from starting period 2 with a capital stock k_2:

$$\begin{aligned} v_2(k_2) = \max_{c_2, k_3} \quad & u(c_2) + \beta v_3(k_3) \\ \text{s.t.} \quad & c_2 + k_3 \leq A_2 k_2^\alpha + (1-\delta)k_2 \end{aligned} \tag{3}$$

  - Notice that we now have a **continuation value**

- Finally, we can rewrite the first period problem as

$$\begin{aligned} v_1(k_1) = \max_{c_1, k_1} \quad & u(c_1) + \beta v_2(k_2) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha + (1-\delta)k_1 \end{aligned} \tag{4}$$

Multi-stage budgeting

- This will work for T periods when T is small, but it doesn't generalize well...
- Instead, let's split the problem up into multiple stages, each of which looks easier to solve
- Define $v_3(k_3)$ as the value you get from starting period 3 with a capital stock k_3:

$$\begin{aligned} v_3(k_3) = \max_{c_3} \quad & u(c_3) \\ \text{s.t.} \quad & c_3 \leq A_3 k_3^{\alpha} + (1-\delta)k_3 \end{aligned} \tag{2}$$

- And define $v_2(k_2)$ as the value you get from starting period 2 with a capital stock k_2:

$$\begin{aligned} v_2(k_2) = \max_{c_2, k_3} \quad & u(c_2) + \beta v_3(k_3) \\ \text{s.t.} \quad & c_2 + k_3 \leq A_2 k_2^{\alpha} + (1-\delta)k_2 \end{aligned} \tag{3}$$

  - Notice that we now have a **continuation value**
- Finally, we can rewrite the first period problem as

$$\begin{aligned} v_1(k_1) = \max_{c_1, k_1} \quad & u(c_1) + \beta v_2(k_2) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^{\alpha} + (1-\delta)k_1 \end{aligned} \tag{4}$$

Why does this work?

- For simplicity, set $\delta = 1$ (full depreciation) and look at our problem again:

$$\begin{aligned} v_1(k_1) := \max_{c_1,c_2,c_3,k_2,k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha && \text{Period 1 BC} \\ & c_2 + k_3 \leq A_2 k_2^\alpha && \text{Period 2 BC} \\ & c_3 \leq A_3 k_3^\alpha && \text{Period 3 BC} \end{aligned} \tag{5}$$

- If we substitute in our budget constraints, we see that the payoff at period t only depends on the capital stock you take into the period, and choices you make later

$$\begin{aligned} v_1(k_1) &= \max_{k_2,k_3} \; u(A_1 k_1^\alpha - k_2) + \beta u(A_2 k_2^\alpha - k_3) + \beta^2 u(A_3 k_3^\alpha) \\ &= \max_{k_2} \; u(A_1 k_1^\alpha - k_2) + \beta \underbrace{\left[\max_{k_3} \; u(A_2 k_2^\alpha - k_3) + \beta u(A_3 k_3^\alpha) \right]}_{v_2(k_2)} \end{aligned} \tag{6}$$

- Our problem has a **recursive structure**

Why does this work?

- For simplicity, set $\delta = 1$ (full depreciation) and look at our problem again:

$$\begin{aligned} v_1(k_1) := \max_{c_1,c_2,c_3,k_2,k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\ \text{s.t.} \quad & c_1 + k_2 \leq A_1 k_1^\alpha && \text{Period 1 BC} \\ & c_2 + k_3 \leq A_2 k_2^\alpha && \text{Period 2 BC} \\ & c_3 \leq A_3 k_3^\alpha && \text{Period 3 BC} \end{aligned} \tag{5}$$

- If we substitute in our budget constraints, we see that the payoff at period t only depends on the capital stock you take into the period, and choices you make later

$$\begin{aligned} v_1(k_1) &= \max_{k_2,k_3} \; u(A_1 k_1^\alpha - k_2) + \beta u(A_2 k_2^\alpha - k_3) + \beta^2 u(A_3 k_3^\alpha) \\ &= \max_{k_2} \; u(A_1 k_1^\alpha - k_2) + \beta \underbrace{\left[\max_{k_3} \; u(A_2 k_2^\alpha - k_3) + \beta u(A_3 k_3^\alpha) \right]}_{v_2(k_2)} \end{aligned} \tag{6}$$

- Our problem has a **recursive structure**

Why does this work?

- For simplicity, set $\delta = 1$ (full depreciation) and look at our problem again:

$$
\begin{aligned}
v_1(k_1) := \max_{c_1, c_2, c_3, k_2, k_3} \quad & u(c_1) + \beta u(c_2) + \beta^2 u(c_3) \\
\text{s.t.} \quad & c_1 + k_2 \le A_1 k_1^{\alpha} && \text{Period 1 BC} \\
& c_2 + k_3 \le A_2 k_2^{\alpha} && \text{Period 2 BC} \\
& c_3 \le A_3 k_3^{\alpha} && \text{Period 3 BC}
\end{aligned}
\tag{5}
$$

- If we substitute in our budget constraints, we see that the payoff at period t only depends on the capital stock you take into the period, and choices you make later

$$
\begin{aligned}
v_1(k_1) &= \max_{k_2, k_3} \; u(A_1 k_1^{\alpha} - k_2) + \beta u(A_2 k_2^{\alpha} - k_3) + \beta^2 u(A_3 k_3^{\alpha}) \\
&= \max_{k_2} \; u(A_1 k_1^{\alpha} - k_2) + \beta \underbrace{\left[\max_{k_3} \; u(A_2 k_2^{\alpha} - k_3) + \beta u(A_3 k_3^{\alpha}) \right]}_{v_2(k_2)}
\end{aligned}
\tag{6}
$$

- Our problem has a **recursive structure**

Finite Horizon: General Case

- You can follow this logic through to the general case where we have T periods
- You've seen the finite horizon problem in its **sequential formulation:**

$$\begin{aligned} v_0(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^\alpha + (1-\delta) k_t \qquad \text{for all } t \geq 0 \end{aligned} \tag{7}$$

- This can be re-written in a **recursive formulation** as:

$$\begin{aligned} v_t(k) := \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^\alpha + (1-\delta) k \\ v_{T+1}(k) := 0 & \end{aligned} \tag{8}$$

- This is called a **Bellman equation**
- We've turned a T dimensional optimization problem into a sequence of T separate 1 dimensional optimization problems
- Note however: we have to solve eq. (8) for many different values of k

In general, it is usually still worth it to reformulate the problem this way, even in the finite horizon case

Finite Horizon: General Case

- You can follow this logic through to the general case where we have T periods
- You've seen the finite horizon problem in its **sequential formulation:**

$$\begin{aligned} v_0(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^\alpha + (1-\delta) k_t \qquad \text{for all } t \geq 0 \end{aligned} \tag{7}$$

- This can be re-written in a **recursive formulation** as:

$$\begin{aligned} v_t(k) := \max_{c, k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^\alpha + (1-\delta) k \\ v_{T+1}(k) := 0 & \end{aligned} \tag{8}$$

- This is called a **Bellman equation**
- We've turned a T dimensional optimization problem into a sequence of T separate 1 dimensional optimization problems
- Note however: we have to solve eq. (8) for many different values of k

In general, it is usually still worth it to reformulate the problem this way, even in the finite horizon case

Finite Horizon: General Case

- You can follow this logic through to the general case where we have T periods
- You've seen the finite horizon problem in its **sequential formulation:**

$$\begin{aligned} v_0(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{T} \beta^t u(c_t) \\ \text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^{\alpha} + (1-\delta) k_t \qquad \text{for all } t \geq 0 \end{aligned} \tag{7}$$

- This can be re-written in a **recursive formulation** as:

$$\begin{aligned} v_t(k) := \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^{\alpha} + (1-\delta) k \\ v_{T+1}(k) := 0 & \end{aligned} \tag{8}$$

- This is called a **Bellman equation**
- We've turned a T dimensional optimization problem into a sequence of T separate 1 dimensional optimization problems
- Note however: we have to solve eq. (8) for many different values of k

In general, it is usually still worth it to reformulate the problem this way, even in the finite horizon case

Finite Horizon: General Case

- You can follow this logic through to the general case where we have T periods
- You've seen the finite horizon problem in its **sequential formulation:**

$$
\begin{aligned}
v_0(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{T} \beta^t u(c_t) \\
\text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^\alpha + (1-\delta) k_t \qquad \text{for all } t \geq 0
\end{aligned} \tag{7}
$$

- This can be re-written in a **recursive formulation** as:

$$
\begin{aligned}
v_t(k) := \max_{c, k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\
\text{s.t.} \quad & c + k' \leq A_t k^\alpha + (1-\delta) k \\
v_{T+1}(k) := 0 &
\end{aligned} \tag{8}
$$

- This is called a **Bellman equation**
- We've turned a T dimensional optimization problem into a sequence of T separate 1 dimensional optimization problems
- Note however: we have to solve eq. (8) for many different values of k

In general, it is usually still worth it to reformulate the problem this way, even in the finite horizon case

Finite Horizon: General Case

- You can follow this logic through to the general case where we have T periods
- You've seen the finite horizon problem in its **sequential formulation:**

$$
\begin{aligned}
v_0(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{T} \beta^t u(c_t) \\
\text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^\alpha + (1-\delta) k_t \qquad \text{for all } t \geq 0
\end{aligned}
\tag{7}
$$

- This can be re-written in a **recursive formulation** as:

$$
\begin{aligned}
v_t(k) := \max_{c, k'} \quad & u(c) + \beta v_{t+1}(k') \qquad\qquad \text{for all } 0 \leq t \leq T \\
\text{s.t.} \quad & c + k' \leq A_t k^\alpha + (1-\delta) k \\
v_{T+1}(k) := 0 &
\end{aligned}
\tag{8}
$$

- This is called a **Bellman equation**
- We've turned a T dimensional optimization problem into a sequence of T separate 1 dimensional optimization problems
- Note however: we have to solve eq. (8) for many different values of k

In general, it is usually still worth it to reformulate the problem this way, even in the finite horizon case

Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

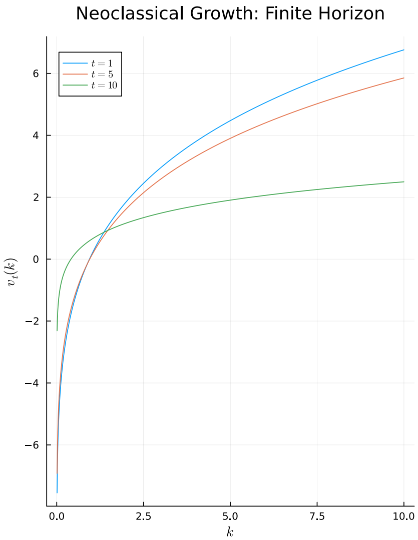
Backwards Induction

- This suggests that for finite horizon problems, at least, we can solve the problem backwards
  1. For $t = T$, solve eq. (8) taking $v_{T+1}(k)$ as given. Save the results
  2. Next, for $t = T - 1$, solve eq. (8) taking $v_{t+1}(k)$ as given (you just solved for it in the previous step)
  3. Do the same for $t = T - 2$, then $t = T - 3$, and so on, until we reach $t = 0$.
- This algorithm is called **Backwards Induction**. If T is finite, it is always well-defined, and will always finish.
- How you solve for v_t depends on your preferences/the properties of the problem
  - You can discretize the problem (a grid of k_i values, and a grid of $v_{t,i}$ values)
  - You can use a function approximation technique from last week, and use $\widehat{v}_{t+1,i}$ when you solve at time t
  - If you interpolate, you can use faster optimization methods on the inside maximization problem

Backwards Induction: $T = 10$

```julia
using Parameters
p    = (β = 0.9, δ = 0.1, α = 0.5, A = 1.0)
u(c) = c > 0 ? log(c) : -Inf

function update_bellman!(p, V, policy, kgrid, V0
    @unpack A, β, δ, α = p
    for i in eachindex(V, kgrid)
        k  = kgrid[i]
        z  = A * k^α + (1-δ) * k
        v′, i′ = findmax(eachindex(kgrid)) do ki
            c  =  z - kgrid[ki]
            return u(c) + β * V0[ki]
        end
        V[i]        = v′
        policy[i]   = i′
    end
end
n            = 1000
kgrid        = LinRange(1e-4, 10, n)
V, policy    = zeros(n,11), zeros(Int, n, 11)
for i in 10:-1:1
    update_bellman!(
        p,V[:,i],policy[:,i],kgrid,V[:, i+1])
end
```



Generalizing to Other Models

- I've shown you this for just the neoclassical growth model, but this all generalizes to a very wide class of models
- It will work for anything that you can write as:

$$
\begin{aligned}
v_t(s) = \max_x \quad & f(s,x) + \beta v_{t+1}(s') \\
\text{s.t.} \quad & s' = g(s,x) \\
& x \in \mathcal{D}(x)
\end{aligned} \tag{9}
$$

  - s denotes the **state variables** (carried from one period to the next)
  - x denotes the **control variables** (picked by the decision maker)
  - $f(s,x)$ denotes the **flow value** (profits, utility, etc...)
  - $g(s,x)$ denotes the **law of motion** for the state variables
  - $\mathcal{D}(x)$ denotes the **decision set** (or constraint set) of our decision maker
- The key trick to writing a problem recursively is to think carefully about which variables are control variables, and which ones are state variables

Section 2

Infinite Horizon Dynamic Programs

Infinite horizon case

- Before, you've seen the neoclassical growth model written in its infinite horizon formulation:

$$\begin{aligned} v(k_0) = \max_{c_t, k_{t+1}} \quad & \sum_{t=0}^{\infty} \beta^t u(c_t) \\ \text{s.t.} \quad & c_t + k_{t+1} \leq A_t k_t^{\alpha} + (1-\delta) k_t \qquad \text{for all } t \geq 0 \end{aligned} \tag{10}$$

- It turns out the two-stage budgeting logic works here as well:

$$\begin{aligned} v(k) = \max_{c, k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq A k^{\alpha} + (1-\delta) k \end{aligned} \tag{11}$$

- Note: for simplicity, I've assumed that A is constant here.

  Otherwise, we would need A to be a state variable, or do something to transform the problem along the balanced growth path

Infinite horizon case: Why it works

To see (intuitively) why this works, let $BC(k)$ encode the budget constraint set with starting capital stock k

$$v(k_0) = \max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_t) \quad \text{Rewrite eq. (10)}$$

$$= \max_{(c_t, k_{t+1}) \in BC(k_t)} u(c_0) + \sum_{t=1}^{\infty} \beta^t u(c_t) \quad \text{Split off } t = 0$$

$$= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=1}^{\infty} \beta^{t-1} u(c_t) \right) \quad \text{Factor } \beta \text{ and split the max}$$

$$= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_{t+1}) \right) \quad \text{Reindex the sum}$$

$$= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta v(k_1) \quad \text{Substitute def of } v$$

Note: if you want to be very precise, there is a fair bit of work left to do to show that the recursively defined v agrees with the sequentially defined v, but that is beyond the scope of this course. You have to trust me that in general this is true. (Or go to grad school!)

Infinite horizon case: Why it works

To see (intuitively) why this works, let $BC(k)$ encode the budget constraint set with starting capital stock k

$$
\begin{aligned}
v(k_0) &= \max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_t) && \text{Rewrite eq. (10)} \\
&= \max_{(c_t, k_{t+1}) \in BC(k_t)} u(c_0) + \sum_{t=1}^{\infty} \beta^t u(c_t) && \text{Split off } t = 0 \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=1}^{\infty} \beta^{t-1} u(c_t) \right) && \text{Factor } \beta \text{ and split the max} \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^{t} u(c_{t+1}) \right) && \text{Reindex the sum} \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta v(k_1) && \text{Substitute def of } v
\end{aligned}
$$

Note: if you want to be very precise, there is a fair bit of work left to do to show that the recursively defined v agrees with the sequentially defined v, but that is beyond the scope of this course. You have to trust me that in general this is true. (Or go to grad school!)

Infinite horizon case: Why it works

To see (intuitively) why this works, let $BC(k)$ encode the budget constraint set with starting capital stock k

$$
\begin{aligned}
v(k_0) &= \max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_t) && \text{Rewrite eq. (10)} \\
&= \max_{(c_t,k_{t+1})\in BC(k_t)} u(c_0) + \sum_{t=1}^{\infty} \beta^t u(c_t) && \text{Split off } t=0 \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=1}^{\infty} \beta^{t-1} u(c_t) \right) && \text{Factor } \beta \text{ and split the max} \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=0}^{\infty} \beta^{t} u(c_{t+1}) \right) && \text{Reindex the sum} \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta v(k_1) && \text{Substitute def of } v
\end{aligned}
$$

Note: if you want to be very precise, there is a fair bit of work left to do to show that the recursively defined v agrees with the sequentially defined v, but that is beyond the scope of this course. You have to trust me that in general this is true. (Or go to grad school!)

Infinite horizon case: Why it works

To see (intuitively) why this works, let $BC(k)$ encode the budget constraint set with starting capital stock k

$$
\begin{aligned}
v(k_0) &= \max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_t) && \text{Rewrite eq. (10)} \\
&= \max_{(c_t, k_{t+1}) \in BC(k_t)} u(c_0) + \sum_{t=1}^{\infty} \beta^t u(c_t) && \text{Split off } t = 0 \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=1}^{\infty} \beta^{t-1} u(c_t) \right) && \text{Factor } \beta \text{ and split the max} \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t, k_{t+1}) \in BC(k_t)} \sum_{t=0}^{\infty} \beta^{t} u(c_{t+1}) \right) && \text{Reindex the sum} \\
&= \max_{(c_0, k_1) \in BC(k_0)} u(c_0) + \beta v(k_1) && \text{Substitute def of } v
\end{aligned}
$$

Note: if you want to be very precise, there is a fair bit of work left to do to show that the recursively defined v agrees with the sequentially defined v, but that is beyond the scope of this course. You have to trust me that in general this is true. (Or go to grad school!)

Infinite horizon case: Why it works

To see (intuitively) why this works, let $BC(k)$ encode the budget constraint set with starting capital stock k

$$
\begin{aligned}
v(k_0) &= \max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=0}^{\infty} \beta^t u(c_t) && \text{Rewrite eq. (10)} \\
&= \max_{(c_t,k_{t+1})\in BC(k_t)} u(c_0) + \sum_{t=1}^{\infty} \beta^t u(c_t) && \text{Split off } t=0 \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=1}^{\infty} \beta^{t-1} u(c_t) \right) && \text{Factor } \beta \text{ and split the max} \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta \left(\max_{(c_t,k_{t+1})\in BC(k_t)} \sum_{t=0}^{\infty} \beta^{t} u(c_{t+1}) \right) && \text{Reindex the sum} \\
&= \max_{(c_0,k_1)\in BC(k_0)} u(c_0) + \beta v(k_1) && \text{Substitute def of } v
\end{aligned}
$$

Note: if you want to be very precise, there is a fair bit of work left to do to show that the recursively defined v agrees with the sequentially defined v, but that is beyond the scope of this course. You have to trust me that in general this is true. (Or go to grad school!)

How to solve for v: finite horizon logic

- We now have this **recursive formulation** of our problem:

$$\begin{aligned} v(k) = \max_{c,k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq Ak^\alpha + (1-\delta)k \end{aligned} \tag{12}$$

  but how do we actually solve it?

- Let's go back to the finite horizon problem, and imagine that T is really large.

$$\begin{aligned} v_t(k) = \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^\alpha + (1-\delta)k \\ v_{T+1}(k) = h(k) & \end{aligned} \tag{13}$$

- How important is the terminal (boundary) condition V_{T+1} to the solution at $t = 0$?
- Notice that it gets discounted by β every period. If $\beta < 1$,

$$\lim_{T\to\infty} \beta^{T+1} V_{T+1}(k) = 0 \tag{14}$$

- As T gets large, the finite horizon problem (at $t = 0$) looks more and more like the infinite horizon problem

How to solve for v: finite horizon logic

- We now have this **recursive formulation** of our problem:

$$\begin{aligned} v(k) = \max_{c,k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \end{aligned} \tag{12}$$

  but how do we actually solve it?

- Let's go back to the finite horizon problem, and imagine that T is really large.

$$\begin{aligned} v_t(k) = \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^{\alpha} + (1-\delta)k \\ v_{T+1}(k) = h(k) & \end{aligned} \tag{13}$$

- How important is the terminal (boundary) condition V_{T+1} to the solution at $t = 0$?
- Notice that it gets discounted by β every period. If $\beta < 1$,

$$\lim_{T\to\infty} \beta^{T+1} V_{T+1}(k) = 0 \tag{14}$$

- As T gets large, the finite horizon problem (at $t = 0$) looks more and more like the infinite horizon problem

How to solve for v: finite horizon logic

- We now have this **recursive formulation** of our problem:

$$\begin{aligned} v(k) = \max_{c,k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \end{aligned} \tag{12}$$

  but how do we actually solve it?

- Let's go back to the finite horizon problem, and imagine that T is really large.

$$\begin{aligned} v_t(k) = \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^{\alpha} + (1-\delta)k \\ v_{T+1}(k) = h(k) & \end{aligned} \tag{13}$$

- How important is the terminal (boundary) condition V_{T+1} to the solution at $t = 0$?
- Notice that it gets discounted by β every period. If $\beta < 1$,

$$\lim_{T \to \infty} \beta^{T+1} V_{T+1}(k) = 0 \tag{14}$$

- As T gets large, the finite horizon problem (at $t = 0$) looks more and more like the infinite horizon problem

How to solve for v: finite horizon logic

- We now have this **recursive formulation** of our problem:

$$\begin{aligned} v(k) = \max_{c,k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \end{aligned} \tag{12}$$

  but how do we actually solve it?

- Let's go back to the finite horizon problem, and imagine that T is really large.

$$\begin{aligned} v_t(k) = \max_{c,k'} \quad & u(c) + \beta v_{t+1}(k') \qquad \text{for all } 0 \leq t \leq T \\ \text{s.t.} \quad & c + k' \leq A_t k^{\alpha} + (1-\delta)k \\ v_{T+1}(k) = h(k) & \end{aligned} \tag{13}$$

- How important is the terminal (boundary) condition V_{T+1} to the solution at $t = 0$?
- Notice that it gets discounted by β every period. If $\beta < 1$,

$$\lim_{T\to\infty} \beta^{T+1} V_{T+1}(k) = 0 \tag{14}$$

- As T gets large, the finite horizon problem (at $t = 0$) looks more and more like the infinite horizon problem

Value Function Iteration: Algorithm

- This suggests a simple approach: define

$$
\begin{aligned}
v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\
\text{s.t.} \quad & c + k' \leq Ak^\alpha + (1-\delta)k
\end{aligned}
\tag{15}
$$

1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step
3. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

Note: we're indexing our iterations forward instead of backwards here, so our boundary condition is at $s = 0$ not $t = T$

- This algorithm is called **Value Function Iteration**

Value Function Iteration

Convergence and Uniqueness

$$\begin{aligned} v(k) = \max_{c,k'} \quad & u(c) + \beta v(k') \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \end{aligned} \tag{12}$$

- You can show that most functions defined this way have a unique solution

  The details are complicated, but this will basically be true anytime you have a max operator on the LHS, a well-behaved constraint set, and a discount rate $\beta < 1$.

  I will not ever ask you about problems where the recursive formulation doesn't yield a unique solution, or where value function iteration fails to converge.

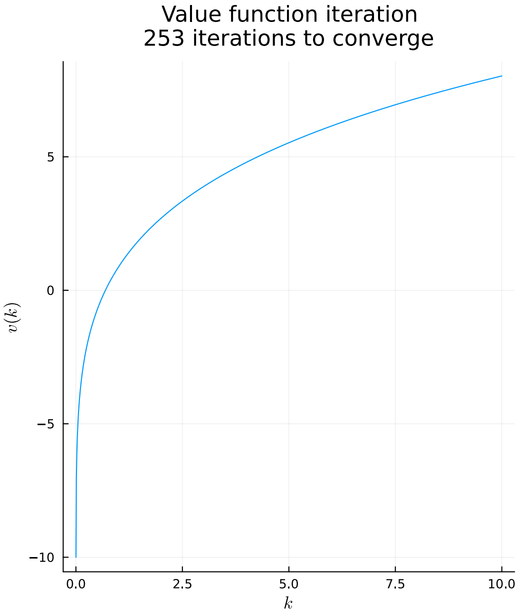
- Moreover, value function iteration converges geometrically to the true solution at a rate proportional to β
  - When β is close to 1, the problem converges more slowly
- This means that for appropriately defined problems, you can *always* use value function iteration, and it will *always* converge to the **unique** solution

VFI in Practice

```julia
function solve!(p, kgrid, V0; tol = 1e-12)
    V       = similar(V0)
    policy  = zeros(Int, size(V0))
    iter    = 0
    while true
        iter += 1
        update_bellman!(
            p, V, policy, kgrid, V0
        )
        converged(V, V0, tol) && break
        V0 .= V
    end
    return (; V, policy, iter)
end

function converged(X, X0, tol)
    return maximum(abs.(X-X0)) < tol
end

solve!(p, LinRange(1e-4, 10, 100), zeros(100)
```



Value Function Iteration is Slow

- Usually requires several hundred (or more) iterations to converge
- Inside each iteration, we have to repeatedly solve a costly maximization problem
- Like all the methods we'll see here, suffers badly from the curse of dimensionality:
  - Suppose your state space is multi-dimensional You need to put a grid of values on each dimension.
  - If you have n dimensions, and your grid G is

    $$G = G_1 \times G_2 \times \cdots \times G_n$$

    then the total number of grid points is $|G| = \prod_{i=1}^{n} |G_i|$
  - If n is 6, and $|G_i| = 10$ (a coarse grid) then we have to solve 1 million maximization problems at every iteration. This gets **very costly** very fast

Speeding it up: Policy Function Iteration

$$
\begin{aligned}
v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\
\text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k
\end{aligned}
\tag{15}
$$

- We started with eq. (15), however, it is very costly to compute the maximization step
- When we are close to the true solution, the optimal policy will not be changing very much
- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^{\star}, k'^{\star}$ from the previous iteration?

Speeding it up: Policy Function Iteration

$$
\begin{aligned}
v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\
\text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k
\end{aligned}
\tag{15}
$$

- We started with eq. (15), however, it is very costly to compute the maximization step
- When we are close to the true solution, the optimal policy will not be changing very much
- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^{\star}, k'^{\star}$ from the previous iteration?

Speeding it up: Policy Function Iteration

$$\begin{aligned} v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\ \text{s.t.} \quad & c + k' \le Ak^\alpha + (1-\delta)k \end{aligned} \tag{15}$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^\star, k'^\star$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c_s^\star(k), k_s'^\star(k))$ and the solution as $v_s^0(k)$
  3. For $i = 1, \ldots, n$, set $v_s^i(k) := u(c_s^\star(k)) + \beta v_s^{i-1}(k_s'^\star(k))$
  4. Set $v_s(k) := v_s^n(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Speeding it up: Policy Function Iteration

$$v_s(k) = \max_{c,k'} \quad u(c) + \beta v_{s-1}(k') \qquad \text{s.t.} \quad c + k' \leq Ak^{\alpha} + (1-\delta)k \tag{15}$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^\star, k'^\star$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c^\star_s(k), k'^\star_s(k))$ and the solution as $v^0_s(k)$
  3. For $i = 1, \ldots, n$, set $v^i_s(k) := u(c^\star_s(k)) + \beta v^{i-1}_s(k'^\star_s(k))$
  4. Set $v_s(k) := v^n_s(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Speeding it up: Policy Function Iteration

$$\begin{aligned} v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\ \text{s.t.} \quad & c + k' \leq Ak^\alpha + (1-\delta)k \end{aligned} \tag{15}$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^\star, k'^\star$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c_s^\star(k), k_s'^\star(k))$ and the solution as $v_s^0(k)$
  3. For $i = 1, \ldots, n$, set $v_s^i(k) := u(c_s^\star(k)) + \beta v_s^{i-1}(k_s'^\star(k))$
  4. Set $v_s(k) := v_s^n(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Speeding it up: Policy Function Iteration

$$v_s(k) = \max_{c,k'} \quad u(c) + \beta v_{s-1}(k')$$
$$\text{s.t.} \quad c + k' \leq Ak^{\alpha} + (1-\delta)k \tag{15}$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^\star, k'^\star$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c_s^\star(k), k_s'^\star(k))$ and the solution as $v_s^0(k)$
  3. For $i = 1, \ldots, n$, set $v_s^i(k) := u(c_s^\star(k)) + \beta v_s^{i-1}(k_s'^\star(k))$
  4. Set $v_s(k) := v_s^n(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Speeding it up: Policy Function Iteration

$$\begin{aligned} v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \end{aligned} \tag{15}$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^{\star}, k'^{\star}$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c_s^{\star}(k), k_s'^{\star}(k))$ and the solution as $v_s^0(k)$
  3. For $i = 1, \ldots, n$, set $v_s^i(k) := u(c_s^{\star}(k)) + \beta v_s^{i-1}(k_s'^{\star}(k))$
  4. Set $v_s(k) := v_s^n(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Speeding it up: Policy Function Iteration

$$
\begin{aligned}
v_s(k) = \max_{c,k'} \quad & u(c) + \beta v_{s-1}(k') \\
\text{s.t.} \quad & c + k' \leq Ak^\alpha + (1-\delta)k
\end{aligned}
\tag{15}
$$

- **Key Idea:** What if we skipped the maximization step, and just used the optimal $c^\star, k'^\star$ from the previous iteration?
- **Algorithm:**
  1. Start from any "terminal" condition $v_0(k) = h(k)$ you like
  2. Solve the model "backwards," just like when we did backwards induction on the finite horizon problem. I.e, for each iteration s, solve eq. (15) with v_{s-1} from the previous step, **but save the optimal policy** $(c_s^\star(k), k_s'^\star(k))$ and the solution as $v_s^0(k)$
  3. For $i = 1, \ldots, n$, set $v_s^i(k) := u(c_s^\star(k)) + \beta v_s^{i-1}(k_s'^\star(k))$
  4. Set $v_s(k) := v_s^n(k)$
  5. Stop when $||v_s - v_{s-1}|| < \epsilon$ for some preset tolerance level

In general, this tends to be much faster than value function iteration, although you have to be careful. Convergence is *not guaranteed*

Section 3

Extensions

Stochastic Productivity

- Consider the neoclassical growth model, but where A is a persistent, log-normal shock

$$\begin{aligned} v(k, A) = \max_{c,k'} \quad & u(c) + \beta \mathbb{E}\left[v(k', A') | A\right] \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \\ & \log(A') = \rho \log(A) + \epsilon \\ & \epsilon \sim N(0, \sigma) \end{aligned} \tag{16}$$

- **Key Difference:** now we have another state variable, and we have to take expectations over A' tomorrow.
- How do we handle the expectations operator?
  - **Naive Approach:** simply replace expectations with an integral and calculate it numerically in every function call:

$$\mathbb{E}\left[v(k', A') | A\right] = \int_{-\infty}^{\infty} v(k', \exp(\rho \log(A) + \epsilon)) f(\epsilon) d\epsilon \tag{17}$$

    where f is the pdf of ϵ
  - This will work, but that integral is costly to compute and you will have to calculate it *many, many, many* times

Stochastic Productivity

- Consider the neoclassical growth model, but where A is a persistent, log-normal shock

$$\begin{aligned} v(k, A) = \max_{c,k'} \quad & u(c) + \beta \mathbb{E}\left[v(k', A') | A\right] \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \\ & \log(A') = \rho \log(A) + \epsilon \\ & \epsilon \sim N(0, \sigma) \end{aligned} \tag{16}$$

- **Key Difference:** now we have another state variable, and we have to take expectations over A' tomorrow.
- How do we handle the expectations operator?
  - **Naive Approach:** simply replace expectations with an integral and calculate it numerically in every function call:

$$\mathbb{E}\left[v(k', A') \middle| A\right] = \int_{-\infty}^{\infty} v(k', \exp(\rho \log(A) + \epsilon)) f(\epsilon) d\epsilon \tag{17}$$

    where f is the pdf of ϵ
  - This will work, but that integral is costly to compute and you will have to calculate it *many, many, many* times

Stochastic Productivity

- Consider the neoclassical growth model, but where A is a persistent, log-normal shock

$$\begin{aligned} v(k,A) = \max_{c,k'} \quad & u(c) + \beta \mathbb{E}\left[v(k',A')|A\right] \\ \text{s.t.} \quad & c + k' \leq Ak^{\alpha} + (1-\delta)k \\ & \log(A') = \rho \log(A) + \epsilon \\ & \epsilon \sim N(0,\sigma) \end{aligned} \tag{16}$$

- **Key Difference:** now we have another state variable, and we have to take expectations over A' tomorrow.
- How do we handle the expectations operator?
  - **Naive Approach:** simply replace expectations with an integral and calculate it numerically in every function call:

$$\mathbb{E}\left[v(k',A')\big|A\right] = \int_{-\infty}^{\infty} v(k', \exp(\rho \log(A) + \epsilon)) f(\epsilon) d\epsilon \tag{17}$$

    where f is the pdf of ϵ
  - This will work, but that integral is costly to compute and you will have to calculate it *many, many, many* times

Expectations Operator: Better Approach

Discretize the AR(1)

- Remember from Week 4 that we can discretize an AR(1) process. I.e., we find a grid of ${A_i}_{i=1}^{N_A}$ and a Markov transition P matrix such that

$$\Pr(A' = A_i | A_j) = P_{ij}$$

  is a good discrete approximation of our process. You should use Rouwenhorst's Method to find this. An implementation is available in QuantEcon (both for Python and Julia)

- Note that I've defined this such that the columns of P sum to 1 (make sure you check this, otherwise you need to use the transpose of P)

- Now we can write our expectations operator as

$$\mathbb{E}\left[v(k', A') \mid A = A_j\right] = \sum_{i=1}^{N_A} \Pr(A' = A_i \mid A_j) v(k', A_i) = \sum_{i=1}^{N_A} v(k', A_i) P_{ij} \qquad (18)$$

- Whenever you see a sum like this, you should be thinking about matrix multiplication

Expectations Operator: Better Approach

Discretize the AR(1)

- Remember from Week 4 that we can discretize an AR(1) process. I.e., we find a grid of ${A_i}_{i=1}^{N_A}$ and a Markov transition P matrix such that

$$\Pr(A' = A_i | A_j) = P_{ij}$$

  is a good discrete approximation of our process. You should use Rouwenhorst's Method to find this. An implementation is available in QuantEcon (both for Python and Julia)

- Note that I've defined this such that the columns of P sum to 1 (make sure you check this, otherwise you need to use the transpose of P)
- Now we can write our expectations operator as

$$\mathbb{E}\left[v(k', A') \mid A = A_j\right] = \sum_{i=1}^{N_A} \Pr(A' = A_i \mid A_j) v(k', A_i) = \sum_{i=1}^{N_A} v(k', A_i) P_{ij} \tag{18}$$

- Whenever you see a sum like this, you should be thinking about matrix multiplication

Expectations as a Matrix Product

- Suppose we want to calculate this expectation for a vector of $\{k_s\}_{s=1}^{N_k}$.
- If we stack them up in a matrix: $V_{sj} = v(k_s, A_j)$ then we can compute

$$EV := \underbrace{\begin{bmatrix} v(k_1, A_1) & v(k_1, A_2) & \dots & v(k_1, A_{N_A}) \\ v(k_2, A_1) & v(k_2, A_2) & \dots & v(k_2, A_{N_A}) \\ \vdots & \vdots & \ddots & \vdots \\ v(k_{N_k}, A_1) & v(k_{N_k}, A_2) & \dots & v(k_{N_k}, A_{N_A}) \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,N_A} \\ P_{2,1} & P_{2,2} & \dots & P_{2,N_A} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N_A,1} & P_{N_A,2} & \dots & P_{N_A,N_A} \end{bmatrix}}_{P}$$

- You can check that

$$EV_{sj} = \sum_{i=1}^{N_A} v(k_s, A_i) P_{ij} = \sum_{i=1}^{N_A} v(k_s, A_i) \Pr(A' = A_i | A_j) = \mathbb{E}\left[v(k_s, A) | A_j \right]$$

- In other words, we can calculate our expectations for all the relevant values of k just *once* per value function iteration loop
- To evaluate k off-grid, we can use interpolation *once* on EV instead of V
- In general, this delivers huge speed gains

Expectations as a Matrix Product

- Suppose we want to calculate this expectation for a vector of $\{k_s\}_{s=1}^{N_k}$.
- If we stack them up in a matrix: $V_{sj} = v(k_s, A_j)$ then we can compute

$$EV := \underbrace{\begin{bmatrix} v(k_1,A_1) & v(k_1,A_2) & \dots & v(k_1,A_{N_A}) \\ v(k_2,A_1) & v(k_2,A_2) & \dots & v(k_2,A_{N_A}) \\ \vdots & \vdots & \ddots & \vdots \\ v(k_{N_k},A_1) & v(k_{N_k},A_2) & \dots & v(k_{N_k},A_{N_A}) \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,N_A} \\ P_{2,1} & P_{2,2} & \dots & P_{2,N_A} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N_A,1} & P_{N_A,2} & \dots & P_{N_A,N_A} \end{bmatrix}}_{P}$$

- You can check that

$$EV_{sj} = \sum_{i=1}^{N_A} v(k_s, A_i) P_{ij} = \sum_{i=1}^{N_A} v(k_s, A_i) \Pr(A' = A_i | A_j) = \mathbb{E}\left[v(k_s, A) | A_j\right]$$

- In other words, we can calculate our expectations for all the relevant values of k just *once* per value function iteration loop
- To evaluate k off-grid, we can use interpolation *once* on EV instead of V
- In general, this delivers huge speed gains

Expectations as a Matrix Product

- Suppose we want to calculate this expectation for a vector of $\{k_s\}_{s=1}^{N_k}$.
- If we stack them up in a matrix: $V_{sj} = v(k_s, A_j)$ then we can compute

$$EV := \underbrace{\begin{bmatrix} v(k_1, A_1) & v(k_1, A_2) & \dots & v(k_1, A_{N_A}) \\ v(k_2, A_1) & v(k_2, A_2) & \dots & v(k_2, A_{N_A}) \\ \vdots & \vdots & \ddots & \vdots \\ v(k_{N_k}, A_1) & v(k_{N_k}, A_2) & \dots & v(k_{N_k}, A_{N_A}) \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} P_{1,1} & P_{1,2} & \dots & P_{1,N_A} \\ P_{2,1} & P_{2,2} & \dots & P_{2,N_A} \\ \vdots & \vdots & \ddots & \vdots \\ P_{N_A,1} & P_{N_A,2} & \dots & P_{N_A,N_A} \end{bmatrix}}_{P}$$

- You can check that

$$EV_{sj} = \sum_{i=1}^{N_A} v(k_s, A_i) P_{ij} = \sum_{i=1}^{N_A} v(k_s, A_i) \Pr(A' = A_i | A_j) = \mathbb{E}\left[v(k_s, A) | A_j\right]$$

- In other words, we can calculate our expectations for all the relevant values of k just *once* per value function iteration loop
- To evaluate k off-grid, we can use interpolation *once* on EV instead of V
- In general, this delivers huge speed gains