Lecture 11: Dynamic Programming II

Robert Garrard

Previously

Recall that some problem admit a recursive structure that we can describe with a **Bellman equation**

$$V(k_t) = \max_{c_t} \; \{U(c_t, k_t) + \beta V(g(c_t, k_t))\}$$

We saw that the Bellman operator was a contraction mapping with modulus β.

We could solve the optimization problem numerically by performing **value function iteration**.

Programming a Deterministic VFI

Let's go through the process of programming a VFI in the context of a Ramsey growth model without labor.

Recall that the problem is

$$
\begin{aligned}
V(k_t) = \max_{c_t} \quad & \{U(c_t) + \beta V(k_{t+1})\} \\
\text{s.t.} \quad & c_t = Ak_t^{\alpha} + (1-\delta)k_t - k_{t+1}
\end{aligned}
$$

Let's parameterize the model to have: $U(c_t) = \log c_t$, $\beta = 0.95$, $\alpha = 1/3$, $\delta = 0.05$, $A = 1$.

Programming a Deterministic VFI

[1]:
```
import numpy as np

#####################
# Plotting
import matplotlib
import matplotlib.pyplot as plt
from IPython import display
import seaborn as sns

# Set text.usetex to False if you do not have LaTeX installed
sns.set(context='paper',
        style='whitegrid',
        font='serif',
        font_scale=2,
        rc={'text.usetex': True})
sns.set_palette('deep')

%matplotlib inline
```

[2]:
```
# Model Parameters
BETA = 0.95
ALPHA = 1/3
DELTA = 0.05
A = 1
```

Programming a Deterministic VFI

Suppose we have have one state variable, k_t. Construct a grid of n points for this variable: $[k_0, \ldots, k_n]$.

Recall that the steady state of this system is :

$$\bar{k} = \left[\frac{\alpha\beta A}{1-\beta(1-\delta)}\right]^{\frac{1}{1-\alpha}} \qquad \bar{c} = A\bar{k}^{\alpha} - \delta\bar{k}$$

Let's center the grid around the steady state. Say, within 50%, $k \in [(1-0.5)\bar{k}, (1+0.5)\bar{k}]$.

[3]:
```
KBAR = ((ALPHA*BETA*A)/(1-BETA*(1-DELTA)))**(1/(1-ALPHA))
CBAR = A*KBAR**ALPHA - DELTA*KBAR

n = 1000
KGRID = np.linspace((1-0.5)*KBAR, (1+0.5)*KBAR, n)
TOL = 1e-5
```

Programming a Deterministic VFI

Let U be an $n \times n$ matrix for the period utility, where U_{ij} is the payoff from having i lots of the state variable today (k_t) and taking j lots into tomorrow (k_{t+1}).

We can find what the choice variable (c_t) must be today using the resource constraint.

We also need to ensure that the constraints are satisfied. In particular, consumption must be non-negative.

If $c_t > 0$, set $U_{ij} = \log(c_t)$. If $c_t \leq 0$, set $U_{ij} = -\infty$.

Programming a Deterministic VFI

```
[4]: # Start U with -inf
U = np.ones((n, n)) * -np.inf

for i in range(n):
    for j in range(n):
        # Our capital values are the ith and jth elements of the grid.
        kt = KGRID[i]
        ktp1 = KGRID[j]

        # From the constraint:
        ct = A*kt**ALPHA + (1-DELTA)*kt - ktp1

        # If the constraint is satisfied with this (k_{t}, k_{t+1})
        # combination, evaluate utility.
        if ct > 0:
            U[i,j] = np.log(ct)
```

Programming a Deterministic VFI

Let our current guess for the value function be the $n \times 1$ vector $V = [V_1, \ldots, V_n]'$. Let $\mathbf{1}$ be an $n \times 1$ vector of ones. Then

$$\begin{bmatrix} TV_1 \\ \vdots \\ TV_n \end{bmatrix} = \max_{c_t} \left\{ \begin{pmatrix} U_{11} & \ldots & U_{1n} \\ \vdots & \ldots & \vdots \\ U_{n1} & \ldots & U_{nn} \end{pmatrix} + \beta \begin{pmatrix} V_1 & V_2 & \ldots & V_n \\ \vdots & \ldots & \ldots & \vdots \\ V_1 & V_2 & \ldots & V_n \end{pmatrix} \right\}$$

Where the max operator is understood row-wise. We can write this more compactly as:

$$TV = \max_{c_t} \left\{ U + \mathbf{1}V' \right\}$$

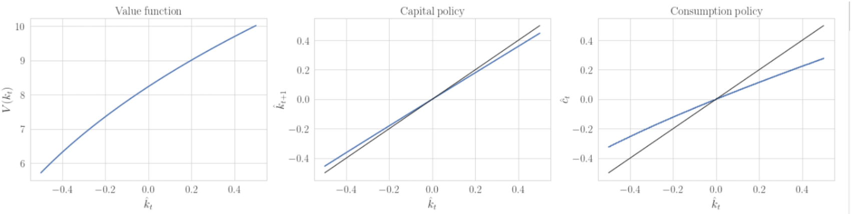
Begin with an initial guess, say $V^0 = [0, \ldots, 0]'$, and iterate until convergence.

Programming a Deterministic VFI

```
[5]: V = np.zeros((n, 1))
     ones = np.ones((n, 1))

     for i in range(300):
         # Store previous iteration
         V_old = V

         # Apply Bellman operator
         RHS = U + BETA*ones.dot(V.T) # Note that this is an outer product
         TV = np.max(RHS, axis=1)     # Maximize along rows

         # Update guess
         V = TV.reshape(n, 1)

         # Stopping criterion
         if np.max(abs(V - V_old)) < TOL:
             print(f'Converged in {i} iterations.')
             break

     sigma_k = KGRID[np.argmax(RHS, axis=1)]
     sigma_c = A*KGRID**ALPHA + (1-DELTA)*KGRID - sigma_k
```

```
Converged in 203 iterations.
```

Programming a Deterministic VFI



Programming a Stochastic VFI

Recall the consumption/savings problem we explored with random income.

$$V(k_t, y_t) = \max \quad \log(c_t) + \beta \mathbb{E}_t V(k_{t+1}, y_{t+1})$$
$$\text{s.t.} \quad c_t = (1+r)k_t + y_t - k_{t+1}$$
$$\mathbb{P}(y_t = 10) = 0.5 \quad \mathbb{P}(y_t = 0) = 0.5$$

We now have an extra state variable with two states, we'll call these states $\{1, 2\}$. In state 1 we get $y_t = 10$, in state 2 we get $y_t = 0$.

Let's decompose this into having a different value function in each state, V_1 and V_2, and a different utility matrix in each state, U_1, U_2.

Note that this model doesn't have a steady state, so we'll continue in levels rather than deviation from steady state.

Let's use $r = 5\%$, and have our grid be $k_t \in [0, 300]$.

Programming a Stochastic VFI

[12]:
```
# Parameters
R = 0.05     # Interest rate
K0 = 100     # Initial savings
n = 1000     # Grid size
KGRID = np.linspace(0, 300, n) # Grid

# Precompute the utility matrices
U1 = np.ones((n, n)) * -np.inf
U2 = np.ones((n, n)) * -np.inf
for i in range(n):
    for j in range(n):
        # Current and next period capital
        kt = KGRID[i]
        ktp1 = KGRID[j]

        # Consumption in each case
        ct1 = (1+R)*kt + 10 - ktp1 # Get income in this case
        ct2 = (1+R)*kt - ktp1      # Not in this case

        # Check constraint isn't violated,
        # evaluate utility.
        if ct1 > 0:
            U1[i,j] = np.log(ct1)
        if ct2 > 0:
            U2[i,j] = np.log(ct2)
```

Programming a Stochastic VFI

Define the transition matrix to be the matrix of probabilities of transitioning from one state to another.

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} \\ \mathcal{P}_{21} & \mathcal{P}_{22} \end{pmatrix}$$

In our case, $\mathcal{P}_{11} = \mathcal{P}_{12} = \mathcal{P}_{21} = \mathcal{P}_{22} = \frac{1}{2}$.

We split up the problem by conditioning on what state we're currently in.

$$TV_1 = \max_{c_t} \left\{ U_1 + \beta \mathcal{P}_{11} \mathbf{1} V_1' + \beta \mathcal{P}_{12} \mathbf{1} V_2' \right\}$$

$$TV_2 = \max_{c_t} \left\{ U_2 + \beta \mathcal{P}_{21} \mathbf{1} V_1' + \beta \mathcal{P}_{22} \mathbf{1} V_2' \right\}$$

Programming a Stochastic VFI

Stacking these together gives

$$\begin{bmatrix} TV_1 \\ TV_2 \end{bmatrix} = \max_{c_t} \left\{ \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} + \beta(\mathcal{P} \otimes \mathbf{1}) \begin{bmatrix} V_1' \\ V_2' \end{bmatrix} \right\}$$

Again, max is row-wise. On the LHS, we have a $2n \times 1$ vector. We just need to split it in half to retrieve our value function.

Iterate, and stop when the max across all entries in the LHS is within tolerance.

Programming a Stochastic VFI

[8]:
```python
P = np.array([[.5, .5], [.5, .5]]) # Transition matrix
ones = np.ones((n, 1))

# Stack the utility matrices
U = np.concatenate([U1, U2], axis=0)
```

[9]:
```python
# Initialize with zeros
V1, V2 = np.zeros((n, 1)), np.zeros((n, 1)) # Value functions for states 1 and 2
V = np.concatenate([V1, V2], axis=0) # Stack them

for i in range(300):
    V_old = V

    # RHS
    RHS = U + BETA*np.kron(P, ones).dot(V.reshape(2, n))

    # Bellman operator
    V = np.max(RHS, axis=1).reshape(2*n, 1)

    # Stopping condition
    if np.max(abs(V - V_old)) < TOL:
        print(f'Converged in {i} iteration.')
        break
```

```
Converged in 236 iteration.
```

[10]:
```python
sigma_k = KGRID[np.argmax(RHS, axis=1)].reshape(2, n).T
V = V.reshape(2, n).T

V1, V2 = V[:, 0], V[:, 1]
sigma_k1, sigma_k2 = sigma_k[:, 0], sigma_k[:, 1]
sigma_c1, sigma_c2 = (1+R)*KGRID + 10 - sigma_k1, (1+R)*KGRID - sigma_k2
```

Programming a Stochastic VFI

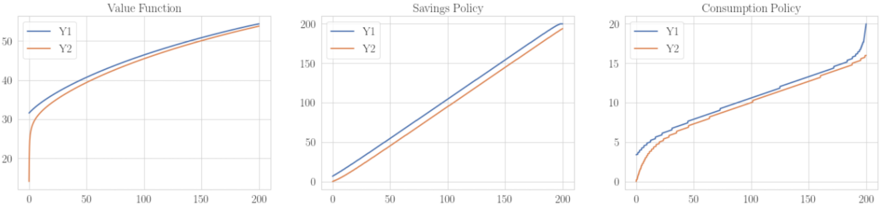


Figure: Value and policy functions

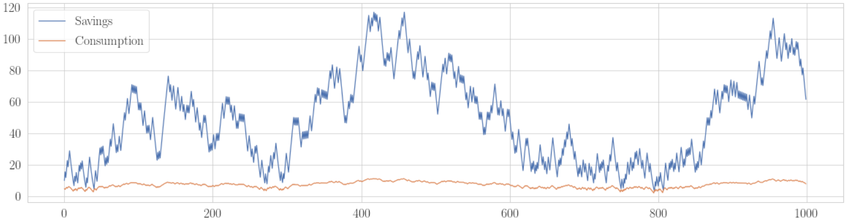


Figure: Simulation for 1000 periods. $K_0 = 10$.

Static Variables

What if we want to include extra variables that are chosen within each period? E.g.

$$\max_{\{c_t, \ell_t, k_{t+1}\}} \quad \sum_{t=0}^{\infty} \beta^t U(c_t, \ell_t)$$
$$\text{s.t.} \quad c_t + k_{t+1} = F(k_t, \ell_t) + (1-\delta)k_t$$

Static Variables

Recall that that the first order condition for labor is:

$$\frac{U_\ell(c_t, k_t)}{U_c(c_t, \ell_t)} = F_\ell(k_t, \ell_t)$$

Solve this first order condition for labor in terms of the control and state: $\ell_t = \ell(c_t, k_t)$, and include it as a constraint in the problem

$$V(k_t) = \max_{c_t} \; \{U(c_t, \ell(c_t, k_t)) + \beta V\left(g(c_t, k_t)\right)\}$$

If you can't solve for $\ell(c_t, k_t)$ analytically, use a numerical solver at each point of your grid.

Static Variables

Let's solve the deterministic growth model with labor supply:

$$V(k_t) = \max_{c_t} \ \{\log(c_t) + \eta \log(1-\ell_t) + \beta V(k_{t+1})\}$$
$$\text{s.t.} \quad c_t = Ak_t^{\alpha}\ell_t^{1-\alpha} + (1-\delta)k_t - k_{t+1}$$

with $\eta = 2$, $\alpha = 1/3$, $\beta = 0.95$, $\delta = 0.05$.

Note the steady state is:

$$\bar{\ell} = \frac{1}{1 + \frac{\eta}{1-\alpha}(1 - \frac{\alpha\beta\delta}{1-\beta(1-\delta}))}$$
$$\bar{k} = \bar{\ell}\left[\frac{\alpha\beta A}{1-\beta(1-\delta)}\right]^{\frac{1}{1-\alpha}}$$
$$\bar{c} = A\bar{k}^{\alpha}\bar{\ell}^{1-\alpha} - \delta\bar{k}$$

Static Variables

```
[14]: from scipy.optimize import fsolve
      import time
      from humanfriendly import format_timespan
```

```
[28]: # Deep Parameters
      BETA = 0.95  # Discount factor
      DELTA = 0.05 # Depreciation rate
      ETA = 2      # Elasticity of leisure
      TOL = 1e-5   # Stopping tolerance
      ALPHA = 0.33 # Capital share of output
      A = 1        # TFP
      n = 1000     # Gridsize
```

```
[29]: # Steady state
      LBAR = 1 / (1 + (ETA / (1-ALPHA))* (1 - BETA*DELTA*ALPHA/(1-BETA*(1-DELTA))))
      KBAR = LBAR*((ALPHA*A)/(1/BETA - (1-DELTA)))**(1/(1-ALPHA))
      CBAR = A*KBAR**ALPHA * LBAR**(1-ALPHA) - DELTA*KBAR

      KGRID = np.linspace((1-0.5)*KBAR, (1+0.5)*KBAR, n).reshape(n, 1)
```

Static Variables

[30]:
```python
# Fill in utility and labor supply matrix
start = time.time()

U = np.ones((n, n)) * -np.inf

for i in range(n):
    for j in range(n):
        # Current and next period capital
        kt, ktp1 = KGRID[i], KGRID[j]

        # First order condition for labor supply
        f = lambda l: ETA*((kt**ALPHA) * (l**(1-ALPHA)) + (1-DELTA)*kt - ktp1) - (1-ALPHA)*(kt**ALPHA) * (l**(-ALPHA))*(1-l)
        lt = fsolve(f, 0.01)

        # Now that we know labor we can compute consumption
        ct = kt**ALPHA * lt**(1-ALPHA) + (1-DELTA)*kt - ktp1

        # Impose all constraints
        if (ct > 0) and (0 < lt < 1):
            U[i,j] = np.log(ct) + ETA*np.log(1-lt)

end = time.time() - start
format_timespan(end)
```

[30]: '2 minutes and 59.42 seconds'

Static Variables

```
[34]: V = np.zeros(shape=(n, 1))

      start = time.time()
      for i in range(300):
          # Stopping criterion
          old_V = V

          # Bellman operator
          RHS = U + BETA*np.kron(np.ones((n,1)), V.T)
          TV = np.max(RHS, axis=1).reshape(n,1)

          V = TV

          # Stopping condition
          if max(abs(V - old_V)) < TOL:
              print(f"Converged in {i} iterations.")
              break

      end = time.time() - start
      print(f"Runtime: {format_timespan(end)}")

      sigma_k = KGRID[np.argmax(RHS, axis=1)]
```

```
Converged in 234 iterations.
Runtime: 4.18 seconds
```

Static Variables

```
[35]: # Another bit that's extra compared to no static variables.
      # Knowing the optimal policy for capital, solve for
      # optimal policies for labor/conumption.
      sigma_l = np.zeros((n,1))
      sigma_c = np.zeros((n,1))

      for i in range(n):
          kt = KGRID[i]
          ktp1 = sigma_k[i]

          # Labor
          f = lambda l: ETA*((kt**ALPHA) * (l**(1-ALPHA)) + (1-DELTA)*kt - ktp1) - (1-ALPHA)*(kt**ALPHA) * (l**(-ALPHA))*(1-l)
          lt = fsolve(f, 0.01)
          sigma_l[i] =lt

          # Consumption
          sigma_c[i] = (kt**ALPHA)*(lt**(1-ALPHA)) + (1-DELTA)*kt - ktp1
```

Static Variables

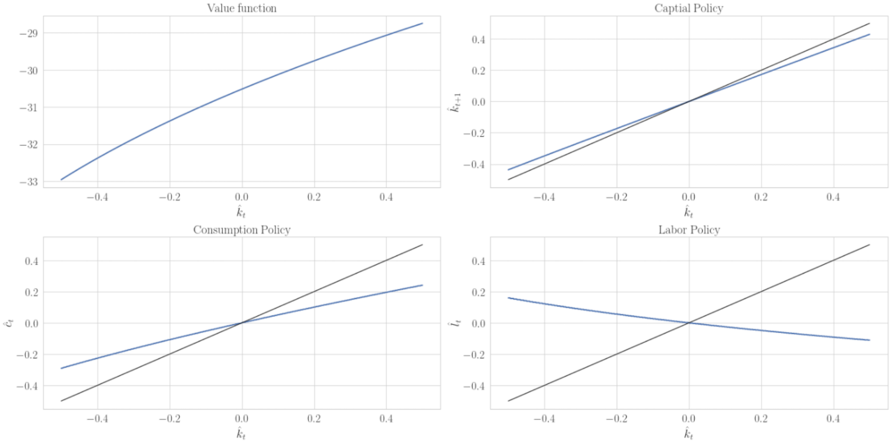


Figure: Value and policy functions for growth model with labor

Discretizing an AR(1)

When we model the process for TFP, it's convenient to specify an AR(1). For example, in a stochastic growth model:

$$
\begin{aligned}
\max_{\{c_t, k_{t+1}\}} \quad & \sum_{t=0}^{\infty} \beta^t \log(c_t) \\
\text{s.t.} \quad & c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta) k_t \\
& z_{t+1} = \rho z_t + \varepsilon \\
& \varepsilon \sim N(0, \sigma_\varepsilon^2)
\end{aligned}
$$

To use our existing methods, we need to know the transition probabilities

$$
\mathbb{P}(z_{t+1} = x \mid z_t = y)
$$

Discretizing an AR(1)

We can use Tauchen's (1986, Econ. Lett.) method.

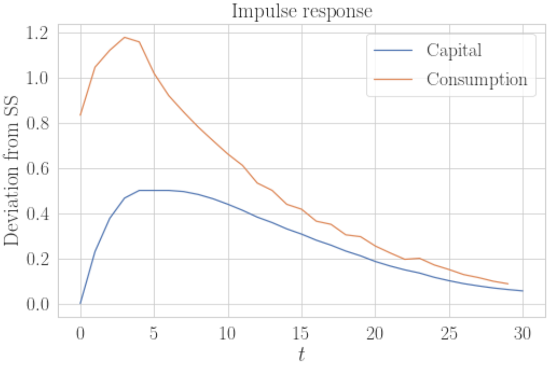
Let our grid be $\lambda_1 < \cdots < \lambda_m$, evenly spaced around the mean of z (0 in this case). Using the fact that ε is normal gives us an approximation:

$$\mathcal{P}_{ij} \approx \begin{cases} \Phi(\frac{\lambda_1 + w/2 - \rho\lambda_i}{\sigma_\varepsilon}) & \text{if } j = 1 \\ \Phi(\frac{\lambda_1 + w/2 - \rho\lambda_i}{\sigma_\varepsilon}) - \Phi(\frac{\lambda_1 - w/2 - \rho\lambda_i}{\sigma_\varepsilon}) & \text{if } 1 < j < m \\ 1 - \Phi(\frac{\lambda_1 - w/2 - \rho\lambda_i}{\sigma_\varepsilon}) & \text{if } j = m \end{cases}$$

Where $w = \lambda_i - \lambda_{i-1}$ is the grid's mesh.

Discretizing an AR(1)

Now that we have productivity shocks, we can talk about what happens when we start in the steady state and are hit by a shock of a particular size (say 1 standard deviation).



Learning Outcomes

You should be able to:

- Numerically solve an optimization problem with value function iteration.
- Construct an appropriate transition matrix for random quantities.