Macroeconomics III
Lecture 5: Practical Dynamic Programming

Tiago Cavalcanti

FGV/EESP
São Paulo

Road Map

1. Practical Dynamic Programming - Value Function Iterations.

Algorithm from Contraction Mapping Theorem

Algorithm to find V:

- Guess V_0; then use operator

$$V_1 = T[V_0] = \max_{0 \le k' \le f(k)} \{u(f(k) - k') + \beta V_0(k')\}$$

  to find V_1.
- Check if $V_0 \approx V_1$;
- if not, find $V_2 = T[V_1]$ from

$$V_2 = T[V_1] = \max_{0 \le k' \le f(k)} \{u(f(k) - k') + \beta V_1(k')\};$$

- continue that until $V_n \approx V_{n-1}$.

Example

Brock and Mirman model:

$$\begin{aligned} \max_{\{c_t, k_{t+1}\}} \quad & \sum_{t=0}^{\infty} \beta^t \ln(c_t) \\ \text{s.t. } k_{t+1} + c_t &= A k_t^{\alpha}, \\ A &> 0; \alpha, \beta \in (0,1) \\ c_t &\geq 0. \end{aligned}$$

Value function:

$$V(k) = \max_{0 \leq k' \leq Ak^{\alpha}} \{ln(Ak^{\alpha} - k') + \beta V(k')\}.$$

Practical Dynamic Programming

Consider our growth model with $u(c) = \log c$ and $f(k) = Ak^{\alpha}$ and define the operator:

$$V_j(k) = \max_{k'}\{log(Ak^{\alpha} - k') + \beta V_{j-1}(k')]\}.$$

How to write a computer code:

- Discretize the state space: $k_1 < k_2 < ... < k_n$; $V_j(k) = \max_{k'}$

$$\begin{bmatrix} log(Ak_1^{\alpha} - k_1') + \beta V_{j-1}(k_1') & log(Ak_2^{\alpha} - k_1') + \beta V_{j-1}(k_1') & \dots & log(Ak_n^{\alpha} - k_1') + \beta V_{j-1}(k_1') \\ log(Ak_1^{\alpha} - k_2') + \beta V_{j-1}(k_2') & log(Ak_2^{\alpha} - k_2') + \beta V_{j-1}(k_2') & \dots & log(Ak_n^{\alpha} - k_2') + \beta V_{j-1}(k_2') \\ \wr & \wr & \dots & \wr \\ log(Ak_1^{\alpha} - k_n') + \beta V_{j-1}(k_n') & log(Ak_2^{\alpha} - k_n') + \beta V_{j-1}(k_n') & \dots & log(Ak_n^{\alpha} - k_n') + \beta V_{j-1}(k_n') \end{bmatrix}$$

- Start with $V_0(k) = 0$
- Find $V_1(k)$. Check if $|V_1(k) - V_0(k)| < \epsilon$;
- If not, find $V_2(k)$. Keep iterating until $|V_n(k) - V_{n-1}(k)| < \epsilon$.

Solving with iterations, using Matlab

Goal: 'Translate' the DP algorithm into a computer code.

1. Set parameters: α, β, A; Compute steady-state: $\bar{k} = [\alpha\beta A]^{\frac{1}{1-\alpha}}$.
2. Grid *(vector)* for k: $k \in [k_1 < k_2 < ... < k_m]$ with $k_1 > 0$
   Tradeoff: *The larger the m is the better the approximation is, but as m increases the computing time increases dramatically.*
3. Given the grid for k, compute the grid for c: $c = Ak^{\alpha} - k'$.
   Matrix, with dimensions $m \times m$:

$$
\begin{array}{c} \\ \uparrow \\ k' \\ \downarrow \end{array}
\begin{array}{c} \longleftarrow k \longrightarrow \\ \\ \\ \\ \\ \end{array}
\begin{bmatrix}
c_{11} & \cdots & c_{1j} & \cdots & c_{1m} \\
\vdots & & & & \vdots \\
c_{i1} & & \boxed{c_{ij}} & & c_{im} \\
\vdots & & & & \vdots \\
c_{m1} & \cdots & c_{mj} & \cdots & c_{mm}
\end{bmatrix}
$$

Rows represent k' and columns k, such that

$$c_{ij} = Ak_j^{\alpha} - k_i'$$

Solving with iterations, using Matlab

Two ways:

- Loop:

```
c = zeros(m,m);
k1 = k;
for i = 1:m
    for j = 1:m
       % columns are for k and rows for k'
       c(i,j) = A*((k(j))^alpha) - k1(i);
    end
end
```

- Vectorization:

```
k1 = k;
c = ones(m,1)*(A*k.^alpha)' - k1*ones(1,m);
```

Redefine $c = \epsilon$ (use a very small number) whenever $c \leq 0$.

4. Given the consumption matrix c, create the utility matrix, U, of dimensions $m \times m$.

Value Function Iterations I

1. Initialize the value function, V, by defining a vector of zeros, with dimensions $m \times 1$.
2. Compute the first period value function and call it TV as follows:

$$\begin{bmatrix} TV(k_1) \\ TV(k_2) \\ \vdots \\ TV(k_m) \end{bmatrix} = \left(\max_{k'} \left\{ \begin{bmatrix} U_{11} & U_{12} & & U_{1m} \\ U_{21} & U_{22} & & \\ & & \ddots & \\ U_{m1} & U_{m2} & & U_{mm} \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \right\} \right)'$$

Given a k_j (i.e. k) we want to find k_i (i.e. k') that maximizes the expression in the curly brackets. I.e. we choose the row number i, that maximizes the expression for column j. If we have a matrix, the *max* function in Matlab does exactly that: it gives a row vector containing the maximum element from each column. We need to transpose the result, in order to finally have a column vector TV. In the code this will look like:

$V = zeros(m, 1)$; % Initializes value function

$o = ones(1, m)$; % Auxiliary vector

$TV = (max(U + beta * V * o))'$; % Finds the first step iteration for TV

Value Function Iterations II

3. Create a loop to iterate on the value function:

   - Set V to be the TV computed from the previous step.
   - Compute a matrix that contains all possible values for $\ln c + \beta V$, then of all the elements of the matrix choose the max and call it TV (like before).
   - Then compare your old value function V with the new value function TV for the stopping rule.

   Stop the loop using a stopping criterion and call the final approximate value function V^*.

(Some comments: while the loop is running you might want to record the number of iterations that where required for convergence)

Policy Function

To find $k' = \tilde{h}(k)$, we need to find k' that solves

$$TV = \max(U + \beta V^*(k'))$$

for all k'. To do this

1. Find the vector of indexes where $U + \beta V^*(k')$ takes its *max* value. To do that, use the function *max* exactly like before, but request the index where the maximization occurs.
2. To find the policy function, use the index and the grid of capital to find $k' = \tilde{h}(k)$

Simulating the Model

With the optimal policy function for the state variable k, we can simulate the time path of k and c over time:

1. Choose an initial value for capital k_0 (an element of the capital grid vector).
2. Create a loop that
   - 2.1 finds which element of the grid k_0 is, and get the index that corresponds to it (use function `find`).
   - 2.2 use this index to find the optimal next period's capital, using the optimal policy vector.

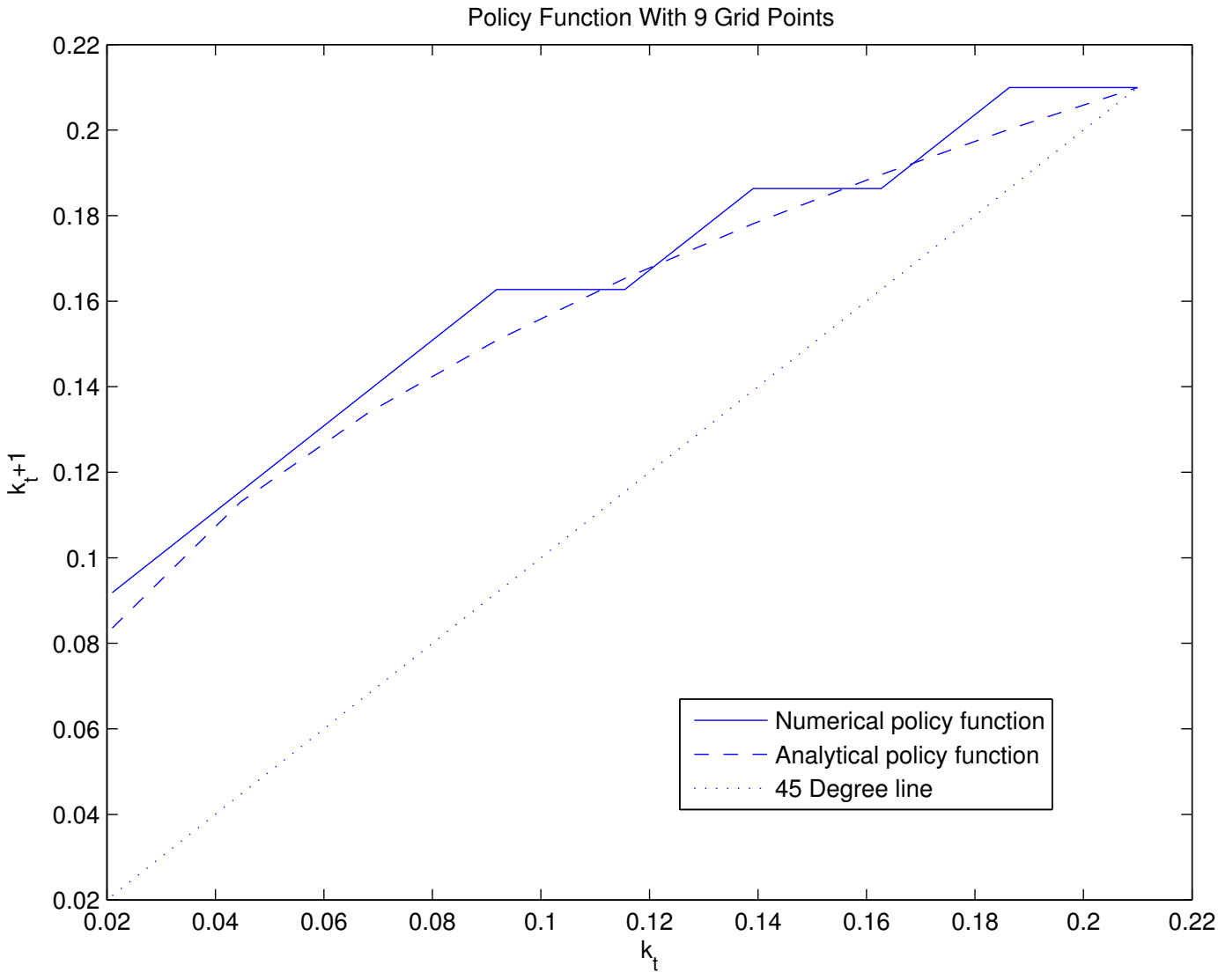
The loop will look like

```
for t = 1:T
    i = find(k == k_path(t));
    k_path(t+1) = policy_k(i);
end
```

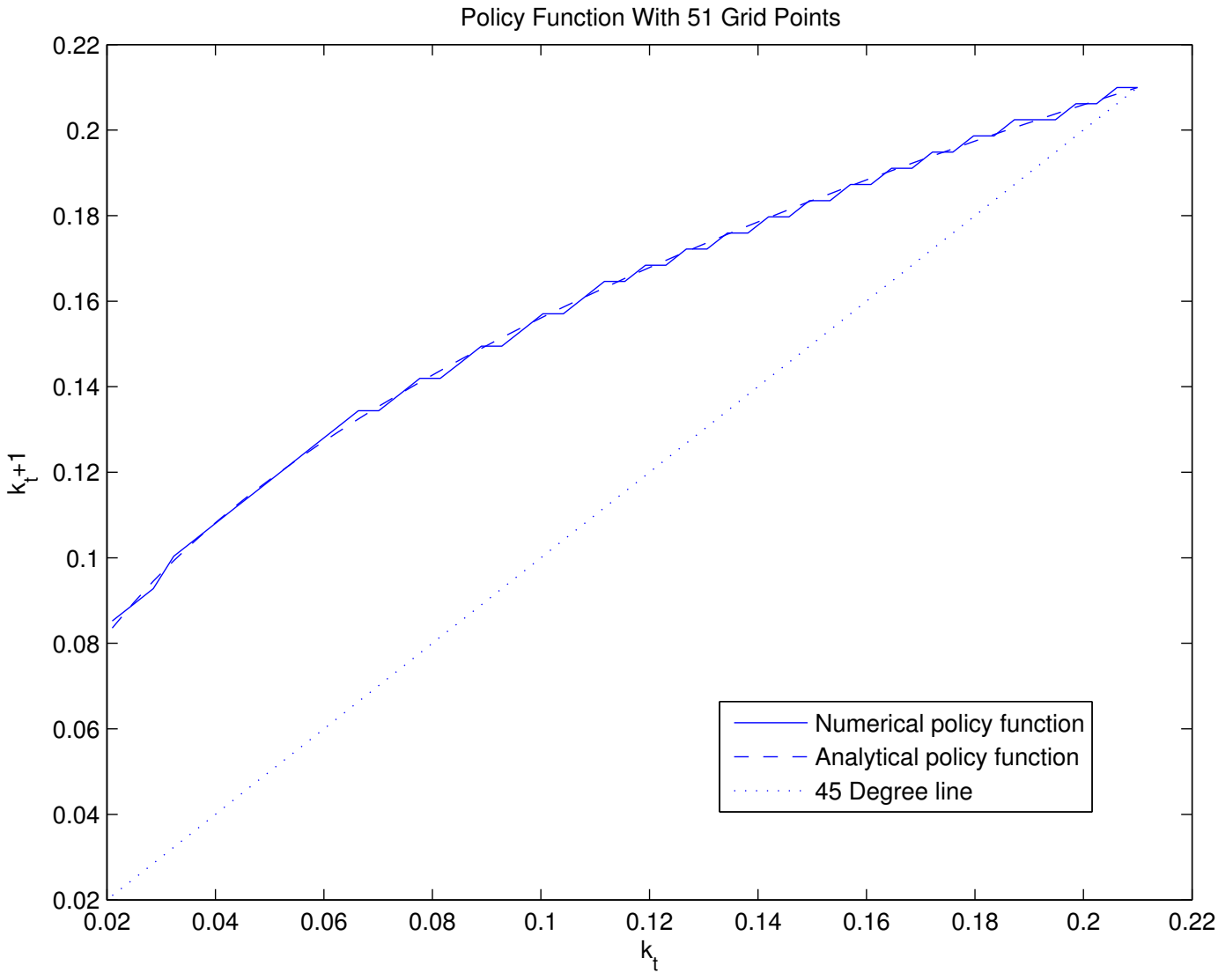
Brock and Mirman (1972) without Uncertainty

MatLab Code

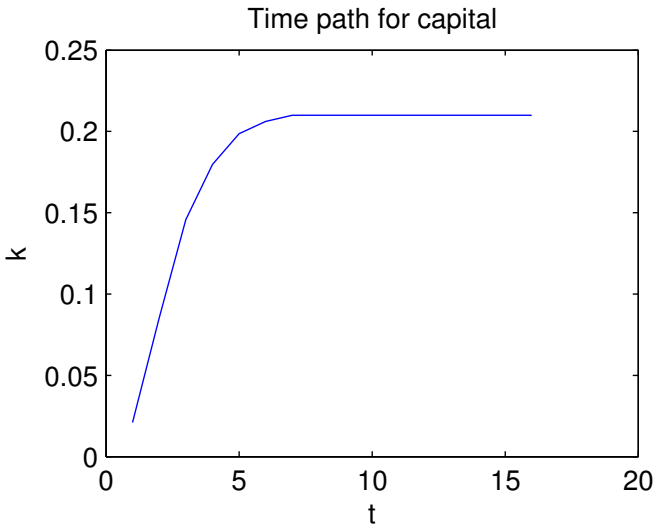
Solution with 9 Grid Points

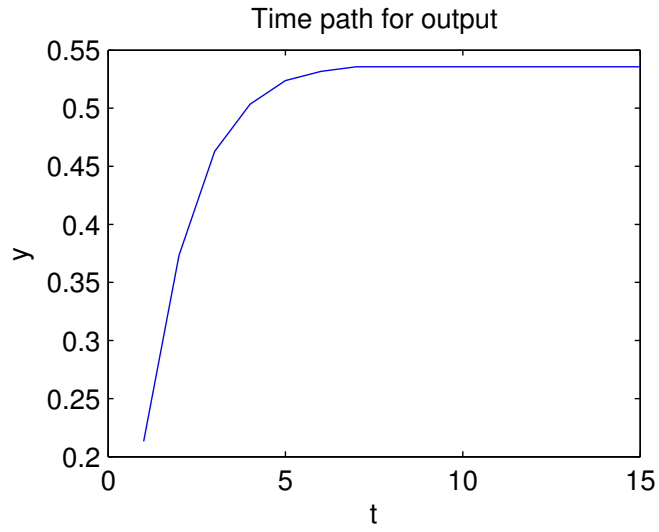


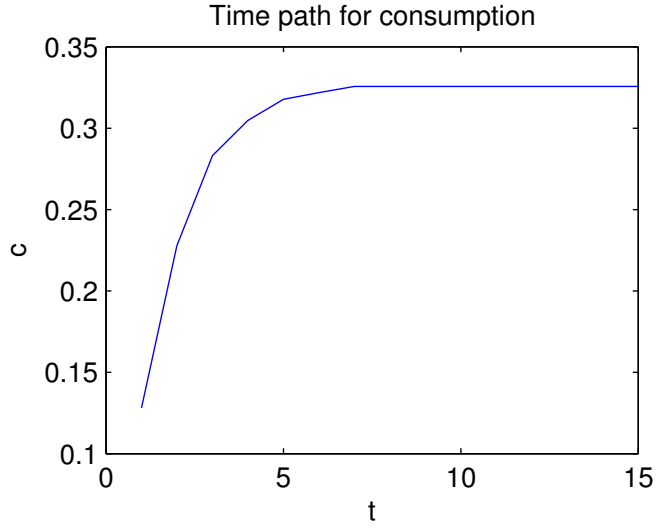
Solution with 51 Grid Points

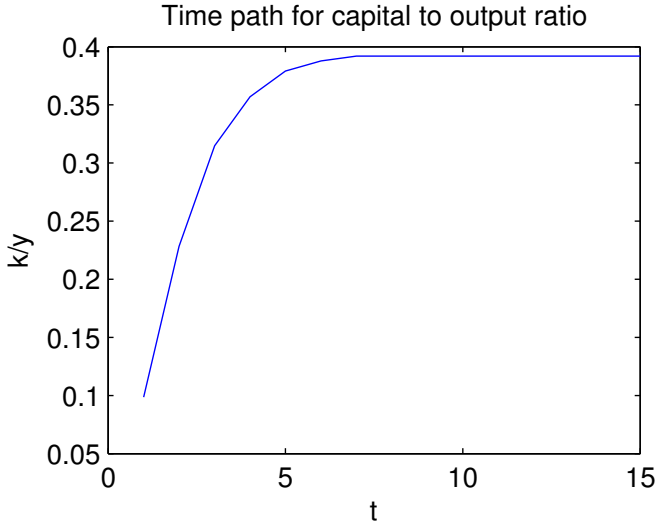


Simulation


Time path for capital
k
t
0.25
0.2
0.15
0.1
0.05
0
0
5
10
15
20


Time path for output
y
t
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0
5
10
15


Time path for consumption
c
t
0.35
0.3
0.25
0.2
0.15
0.1
0
5
10
15


Time path for capital to output ratio
k/y
t
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
5
10
15

Optimal Growth Problem with Uncertainty I

- Suppose consumer maximizes

$$V(k_0, z_0) = \max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} = E_0 \left[\sum_{t=0}^{\infty} \beta^t u(c_t) \right], \ 0 < \beta < 1 \ (1)$$

$$\text{s.t. } k_{t+1} + c_t \leq z_t f(k_t)$$

$$z_{t+1} = \rho z_t + \epsilon_{t+1} \quad , \ \rho \in (0,1), \ \epsilon \sim F(0, \sigma_\epsilon^2)$$

$$k_0 > 0 \text{ given, and } c_t \geq 0.$$

- Note: $u(.)$ and $f(.)$ satisfy standard properties.
- At period t agents know the realization of the shock z_t.

Optimal Growth Problem with Uncertainty II

- We can show that:

$$V(k_0, z_0) = \max_{0 \leq k_1 \leq z_0 f(k_0)} \{u(z_0 f(k_0) - k_1) + \beta E_0[V(k_1, z_1)]\}$$

- since the problem repeats itself in every period, we have that:

$$V(k, z) = \max_{0 \leq k' \leq z f(k)} \{u(z f(k) - k') + \beta E[V(k', z')]\}, \qquad \text{(BE)}$$

  where $(')$ stands for future variables.

- Policy function: $c = g(k, z)$ [or $k' = h(k, z)$]

Model with Uncertainty

- Brock and Mirman (1972):

$$\begin{aligned} V(k_0) &= \max E_0 \sum_{t=0}^{\infty} \beta^t \ln(c_t) \\ k_{t+1} &= A z_t k_t^{\alpha} - c_t \\ k_0 &: \text{given} \\ z_t &: \text{stochastic shock} \end{aligned}$$

- What is z_t?
  - as an iid shock
  - as a Markov chain
  - as an autoregressive process (need to approximate)
- **Two** state variables, k_t and z_t
- The equivalent of a steady state is a limiting stationary distribution

With an iid Shock I

- Assume

$$z_t = \begin{cases} z_1 & \text{w.p. } \pi_1 \\ z_2 & \text{w.p. } \pi_2 \end{cases}, \text{ for all } t$$

- The Bellman equation is then

$$\begin{aligned} V(k,z) &= \max\left\{U + \beta EV(k',z')\right\} \\ &= \max\left\{U + \beta \sum_{i=1}^{2} \pi_i V_i(k',z_i')\right\} \end{aligned}$$

- Steps are like for the deterministic case but repeat twice:
  1. Define parameters, get steady state and make grid (around steady state).
  2. Define two values for z.

With an iid Shock II

- Steps:
  3. Compute 2 matrices for consumption, conditional on the shock, i.e. $C_i = Az_i k^{\alpha} - k'$ and two utility matrices
  4. Compute the first value functions like before, by using

$$\begin{aligned} TV_1 &= (\max\{U_1 + \beta * zeros(m,1) * ones(1,m)\})' \\ TV_2 &= (\max\{U_2 + \beta * zeros(m,1) * ones(1,m)\})' \end{aligned}$$

  5. Two stopping criteria:
     while $check1 > 0.0001$ *OR* $check2 > 0.0001$ (we want the loop to stop when both conditions are satisfied)

$$\begin{aligned} TV_1 &= (\max\{U_1 + \beta * (\pi_1 V_1 + \pi_2 V_2) * ones(1,m)\})' \\ TV_2 &= (\max\{U_2 + \beta * (\pi_1 V_1 + \pi_2 V_2) * ones(1,m)\})' \end{aligned}$$

  6. Compute two policy functions (one for each state z)
  7. Simulate economy many times to get statistics.
- To simulate, you will need to create a function that generates randomly the two states.
- This can be easily generalized to more than two states, but can be very cumbersome if you have a lot of them!

Brock and Mirman (1972) with Uncertainty

MatLab Code

With a Markov Chain

- Assume

$$z_t = \begin{cases} z_1 \\ z_2 \end{cases}$$

with transition matrix

$$\Pi = \begin{bmatrix} \pi_{11} & \pi_{12} \\ \pi_{21} & \pi_{22} \end{bmatrix}$$

where $\pi_{i1} + \pi_{i2} = 1$ (rows sum up to one)

- The steps are the same as before – step 5 is now

5. Two stopping criteria:
   while $check1 > 0.0001$ *OR* $check2 > 0.0001$

$$\begin{aligned} TV_1 &= (\max\{U_1 + \beta * (\pi_{11}V_1 + \pi_{12}V_2) * ones(1,m)\})' \\ TV_2 &= (\max\{U_2 + \beta * (\pi_{21}V_1 + \pi_{22}V_2) * ones(1,m)\})' \end{aligned}$$

- To simulate, you will need to create a function that generates a Markov chain.

Additional Steps

- Check that the policy function is not constrained by the discrete state space. If k' is equal to the highest or lowest value of capital in the grid for some i, relax the bounds of k and redo the value function iteration.
- Check that the error tolerance is small enough. If a small reduction in *tol* results in large changes in the value or policy functions, the tolerance is too high.
- Check whether or not the grid is large enough. If an increase in the grid results in a substantially different solution, the grid might be too sparse.
- A good initial guess of the value function can reduce computation time substantially.

Value Function Iterations: Some Comments

1. Advantages.
   - **Always works**. It can handle discontinuities, non-convex budget constraints, non-concave objective functions.
   - Some problems are naturally discrete choice.

2. Disadvantages.
   - Very slow relatively.
   - Suffers greatly from the **"curse of dimensionality".**

Speed Improvements

- Value function iteration is stable, in that it converges to the true solution, however it is also slow.
- The simplest way to speed up the algorithm is to ensure that you are not doing redoing costly computations over and over again in the iteration loop.
- The operator max is the most time-consuming step in the value function iteration.

Recall the Value Function Algorithm

Algorithm to find V:

- Guess V_0; then use operator

$$V_1 = T[V_0] = \max_{0 \le k' \le f(k)} \{u(f(k) - k') + \beta V_0(k')\}$$

  to find V_1.
- Check if $V_0 \approx V_1$;
- if not, find $V_2 = T[V_1]$ from

$$V_2 = T[V_1] = \max_{0 \le k' \le f(k)} \{u(f(k) - k') + \beta V_1(k')\};$$

- continue that until $V_n \approx V_{n-1}$.

Howard's Improvement

- Howard's improvement reduces the number of times we update the policy function relative to the number of times we update the value function.
  1. Guess V_0; then use operator

$$V_1 = T[V_0] = \max_{0 \leq k' \leq f(k)} \{u(f(k) - k') + \beta V_0(k')\}$$

     to find V_1 and $k' = h(k)$

     1a. Then for some finite $n_h \in \{1, 2, \ldots N_h\}$ iterate:

$$V_1^{n_h} = u(f(k) - h(k)) + \beta V_1^{n_h - 1}(h(k))$$

  2. Check if $V_0 \approx V_1^{N_h}$;
  3. if not, find repeat (1) until $V_n \approx V_{n-1}^{N_h}$.
- Note that there is no optimization in the new step.

Policy Function Iterations

- Advantages
  - Very cost-efficient when maximization is costly;
  - Widely known.
- Disadvantages
  - Can be a little tricky to program because you need to compute the update TV in exactly the same way as in VFI for that theoretical result to hold.
  - Only applies to infinite-horizon problems.

Changing the Problem: Cash-at-hand Formulations

- Many times the largest speed improvements come not from different algorithms but from a reformulated problem, the most common of which is a cash-at-hand. Consider the following setup:
  - Infinitely-lived households;
  - TFP shock z follows Markov-chain, output is zk^{α};
  - Save in a discount bond b' with price q;
  - Save also in capital k';
  - Period utility function u, discount β.

Recursive Representation

- To solve this directly, one has 2 continuous state variables and 1 discrete state variable.

$$V(b,k,z) = \max_{b',k'} \{u(c) + \beta \sum_{z'} V(b',k',z')F(z'/z)\},$$

subject to

$$c + qb' + k' = b + zk^{\alpha} + (1-\delta)k,$$

$$c \geq 0, k' \geq 0, b \geq -\underline{B}.$$

Cash-at-hand Formulation

- A cash-at-hand reformulation is a much easier problem:

$$\tilde{V}(x,z) = \max_{b',k'} \{u(c) + \beta \sum_{z'} \tilde{V}(x'(b',k',z'),z')F(z'/z)\},$$

subject to

$$c + qb' + k' = x,$$

$$x'(b',k',z') = b' + z'k'^{\alpha} + (1-\delta)k',$$

$$c \geq 0, k' \geq 0, b \geq -\underline{B}.$$

- Now there is only 1 continuous state variable and 1 discrete state. This is a massive improvement:

Cash-at-hand Formulation

- Very easy to go back to the original problem once one has the value $\tilde{V}$ and policies $\tilde{b}'$, $\tilde{k}'$, and $\tilde{c}$. Define:

$$x(b,k,z) = b + zk^{\alpha} + (1-\delta)k.$$

- Then:
  - $V(b,k,z) = \tilde{V}(x(b,k,z),z)$;
  - $c(b,k,z) = \tilde{c}(x(b,k,z),z)$;
  - $k'(b,k,z) = \tilde{k}'(x(b,k,z),z)$;
  - $b'(b,k,z) = \tilde{b}(x(b,k,z),z)$.

Note: this step usually requires interpolation.

Cash-at-hand Formulation

- Suppose

$$x = \{0, 0.5, 1, 1.5, 2, 2.5, 3\},$$
$$k = \{0, 0.5, 1, 1.5, 2\},$$
$$b' = \{0, 0.5, 1, 1.5, 2\}.$$

- Let's assume tat $x = k^{\alpha} + b$ with $\alpha = 0.5$

- It can be that given $x = 2$, then the choice is $k' = 1.5$ and $b' = 1$ $\Rightarrow$ $x' = 2.22$. But 2.22 is not in the x grid.

Cash-at-hand Formulation

- Advantages
  - Can provide massive speedups
  - Very often applicable.
- Disadvantages
  - Have to do interpolation (extra coding step);
  - Have to think about an appropriate grid for x.

Euler Equation Errors

A valid criticism of computational economics is that it is often a black box:

- Code produces some results which cannot be verified theoretically;
- Often the model mechanics are not clear (often because the paper is poorly written);
- Often not clear that a good solution has been found;
- Euler equation errors address the last point by assessing the accuracy of an approximation.

Since almost all macro models have some sort of Euler equation, these are very useful.

Euler Equation Errors

Standard Euler equation:

$$u'(c(k,z)) - \beta E_z[R'u'(c'(k,z),z')] = 0.$$

- For the true optimal policies, this would hold exactly.
- However, our computed optimal policies will result in this not holding exactly. We assess the error. Define the Euler equation Error (EEE) in $\log_{10}$ as:

$$EEE(k,z) = \log_{10}\left|1 - \frac{u'^{-1}(\beta E_z[R'u'(c'(k,z),z')])}{c(k,z)}\right|.$$

Euler Equation Errors

- Easy to calculate if shocks have discrete support
- If shocks have continuous support then it requires numerical integration
- Problem: magnitude of errors is hard to interpret (but can be used to check accuracy of different methods)