

# Discrete Dynamic Programming - Stochastic Growth

`x = 0; Remove["Global`*"]`

---

## Outline of notebook

This notebook consists of two basic parts. First, I specify a particular problem, a simple stochastic optimal growth problem. Second, I wrote some general dynamic programming code.

The DP code is set up for an arbitrary general DP, and the example problem shows how to set up a specific application. The general DP code refers only to points on the grids. This means that the setup includes defining functions of those grid points. So,  $\text{payoff}[ix, i\theta, x_{\text{next}}]$  is the payoff in the current period if the current state is  $(ix, i\theta)$  and  $x_{\text{next}}$  is the next grid point where  $ix, i\theta,$  and  $x_{\text{next}}$  are indices of the state space, and  $\text{payoff}$  is a function defined in the setup for a particular problem.

---

## Initializations

The first thing to do in a DP problem is to specify the grid of states, the payoff function, and the transition rules. In this notebook we will have the endogenous states evolve deterministically but have exogenous states evolve stochastically.

### ■ Inputs to Bellman

The initialization phase creates the primitives for the dynamic programming problem and is the basic data for the Bellman code.

The basic inputs are

- (1) the geometry of the grids for the states,
- (2) the Markov transition matrix,
- (3) the payoff function, and
- (4) an initial guess for the value function. We spell these out in more detail here.

**States:**

$x$ : The endogenous state; that is, the state that that can be affected by decisions

$\theta$ : The exogenous state

The states,  $x$  and  $\theta$ , are often continuous in the real world. However, we will use finite sets of points to represent the states. We will assume that these points are simple grids.

**Grids:**

$x_{grid}$ : The vector of grid points for the endogenous state.

$n_x$ : Length of  $x_{grid}$

$\theta_{grid}$ : The vector of grid points for the exogenous state

$n_\theta$ : Length of  $\theta_{grid}$

Terminology shorthand: Sometimes we will refer to a state by its index. For example, when we say that "the state is  $ix$ ", we mean that the state is  $x_{grid}[ix]$ . Similarly when we say the state is  $(ix, i\theta)$ .

**Markov transition matrix:**

Prb[ $i\theta, j\theta$ ]: The probability of the exogenous state being  $j\theta$  in the next period if the current exogenous state is  $i\theta$ .

**Payoff function:**

payoff[ $ik, i\theta, ikp$ ]: the payoff in a period if the current state is  $(ik, i\theta)$  and  $ikp$  is the next period's endogenous state.

**Program flow for setup:**

Here is a table describing the dependencies of the program elements created in the setup phase.

Create valinit, an $n_k \times n_\theta$ matrix	→	→	→	Bellman
Choose $n_k$	→	→	→	Bellman
↓				
Define kgrid				
↓				
Define kgridf				
↓				
Define payoff	→	→	→	Bellman
↑				
Define $\theta$ gridf				
↑				
Define $\theta$ grid	→	Define Prb	→	Bellman
↑				
Choose $n_\theta$	→	→	→	Bellman

## ■ Grid of states

We use a simple growth model as an example.

## ■ Endogenous states

The amount of capital in the economy is controlled by the planner, and output is a function of the endogenous capital and an exogenous productivity factor.

$k$ : capital stock - the endogenous state variable

$k_{\min}$  ( $k_{\max}$ ): the minimum (max) permissible capital stock. [Note: these bounds are made for mathematical convenience. They should be chosen so that they do not bind in the solution.]

$k_{\text{grid}}$ : the vector of permissible capital stocks, a subset of  $[k_{\min}, k_{\max}]$

$n_k$ : length of  $k_{\text{grid}}$

Construct  $k_{\text{grid}}$ : We will construct a uniform grid.  $\kappa$  is the uniform difference between successive states. The possible choices of net investment will be integer multiples of  $\kappa$ .

```

       $\kappa := \frac{k_{\max} - k_{\min}}{nk - 1};$ 
      kgrid := Table[kmin + (i - 1)  $\kappa$ , {i, 1, nk}];

```

Nonuniform grids can also be used. The key requirements is that `kgrid` contain the list of endogenous states and `kgridf` is a function mapping indices to values for the state.

`kgrid` is passed to `Bellman`. `kgridf` is used to define the payoff function below; it is contained in the code passed to `Bellman` computing the payoff function.

```

xgrid := kgrid; nx := nk;

```

We define a function that takes the index of a grid point, `ik`, and computes the capital stock it represents

```

kgridf[ik_] := kmin + (ik - 1)  $\kappa$ 

```



- **Exogenous states**

The productivity states move exogenously.

$\theta$ : productivity state - the exogenous state variable  
 $\theta_{\min}$  ( $\theta_{\max}$ ): the minimum (maximum) value for  $\theta$ .  
 $\theta_{\text{grid}}$ : the vector of possible  $\theta$  values, a subset of  $[\theta_{\min}, \theta_{\max}]$   
 $n\theta$ : length of  $\theta_{\text{grid}}$

Construct  $\theta_{\text{grid}}$ : We will construct a uniform grid.  $\delta\theta$  is the uniform difference between successive states.

$$\delta\theta := \frac{\theta_{\max} - \theta_{\min}}{n\theta - 1};$$

$$\theta_{\text{grid}} := \text{Table}[\theta_{\min} + (i - 1) \delta\theta, \{i, 1, n\theta\}];$$

Here again the index  $i\theta$  is the index in the  $\theta$  grid that contains the value for  $\theta_{\text{grid}}[i\theta]$ .

We define a function that takes the index of a theta grid point,  $i\theta$ , and computes the  $\theta$  value it represents

```
 $\theta$ gridf[i $\theta$ _] :=  $\theta$ min +  $\delta\theta$  (i $\theta$  - 1)
```

Nonuniform grids can also be used. The key requirements is that  $\theta$ grid contain the list of exogenous states and  $\theta$ gridf is a function mapping indices to values for the exogenous state.

$\theta$ grid is passed to Bellman.  $\theta$ gridf is used to define the payoff function below; it is contained in the code passed to Bellman for computing the payoff function.

- Construct grid inputs for Bellman

Choose parameters for kgrid.

```
kmin = 0.1; kmax = 1.1; nk = 51;
```

Output kgrid and kgridf.

```
kgrid
```

```
{0.1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.22, 0.24, 0.26, 0.28, 0.3,
 0.32, 0.34, 0.36, 0.38, 0.4, 0.42, 0.44, 0.46, 0.48, 0.5,
 0.52, 0.54, 0.56, 0.58, 0.6, 0.62, 0.64, 0.66, 0.68, 0.7,
 0.72, 0.74, 0.76, 0.78, 0.8, 0.82, 0.84, 0.86, 0.88, 0.9,
 0.92, 0.94, 0.96, 0.98, 1., 1.02, 1.04, 1.06, 1.08, 1.1}
```

```
kgridf[ik]
```

```
0.1 + 0.02 (-1 + ik)
```

Choose parameters for  $\theta$  grid.

```
 $\theta_{\min} = 0.95$ ;  $\theta_{\max} = 1.05$ ;  $n_{\theta} = 2$ ;
```

Output  $\theta_{\text{grid}}$  and  $\theta_{\text{gridf}}$ .

```
 $\theta_{\text{grid}}$ 
```

```
{0.95, 1.05}
```

```
 $\theta_{\text{gridf}}[\mathbf{i\theta}]$ 
```

```
0.95 + 0.1 (-1 +  $i\theta$ )
```

### ■ Transition rule for $\theta$ values

We next choose the probabilities of the transition matrix for  $\theta$ .

$\text{Prb}[i,j]$  is the probability that  $\theta$  moves from  $\theta_{\text{grid}}[i]$  value to  $\theta_{\text{grid}}[j]$ .

$$\mathbf{Prb}[1, 1] = \mathbf{Prb}[2, 2] = 2 / 3;$$

$$\mathbf{Prb}[1, 2] = \mathbf{Prb}[2, 1] = 1 / 3;$$

- **Construct payoff function**

We next build up the payoff function from the primitives of the production function and the utility function.

- **Production function**

The net-of-depreciation production function is

$$f[k_, \theta_] := \theta \frac{(1 - \beta) k^\alpha}{\beta \alpha}$$

We have defined the production function,  $f[k, \theta]$ , in terms of capital and productivity, the natural state variables. However, the Bellman code understands only indices. Hence, in order to use the Bellman code, we need to define the production function in terms of the state indices. More specifically, if  $k = kgrid[ik]$  and  $\theta = \thetagrid[i\theta]$ , then  $fgrid[ik, i\theta] = f[k, \theta]$ .

$$fgrid[ik_, i\theta_] := f[kgridf[ik], \thetagridf[i\theta]]$$

In this case, the  $kgrid$  and  $\thetagrid$  grids are not sent to Bellman since we have simple formulas for the grid points in terms of the indices.

■ **The dynamic equation for capital stock**

The capital stock evolves according to

$$k_{t+1} = k_t + f[k_t, \theta_t] - c_t$$

At time  $t$ ,  $k_t$  and  $\theta_t$  are the state variables.  $k_{t+1}$  is chosen. Therefore, consumption is

$$c_t = f[k_t, \theta_t] + k_t - k_{t+1}$$



- The utility function

$$u[c\_]=\frac{c^{1-\gamma}}{1-\gamma}; \quad (* \gamma > 0 \text{ in order for } u[c] \text{ to be concave. } *)$$

```
util[cc_] = If[cc > .001, u[cc], -1010];
```

```
(* this is needed in order to avoid evaluating u[c] for negative c. *)
```

We need to define the payoff function in terms of the current state, with index  $ik$ , and the choice of tomorrow's capital stock, with index  $ikp$ . We use the law of motion.

The initial level of capital has index  $ik$ , and equals  $kgridf[ik]$ .

Net output is  $fgrid[ik, i\theta]$ .

Tomorrow's capital stock has index  $ikp$ , and equals  $kgridf[ikp]$ .

Recall that consumption is  $c_t = f[k_t, \theta_t] + k_t - k_{t+1}$

Investment net of depreciation equals  $k_{t+1} - k_t = \kappa (ikp - ik)$ .

Therefore, the payoff at state  $(ik, i\theta)$  and choice  $ikp$  is utility of the resulting consumption, which is

```
payoff[ik_, iθ_, ikp_] := util[fgrid[ik, iθ] - κ (ikp - ik)]
```

- Parameter Values

$$\alpha = .25; \gamma = 3.0; \beta = .9;$$

payoff[ik, iθ, ikp]

$$\text{If} \left[ -0.02 (-ik + ikp) + 0.444444 (0.1 + 0.02 (-1 + ik))^{0.25} (0.95 + 0.1 (-1 + i\theta)) > 0.001, \right. \\ \left. u \left[ -0.02 (-ik + ikp) + 0.444444 (0.1 + 0.02 (-1 + ik))^{0.25} (0.95 + 0.1 (-1 + i\theta)) \right], -10^{10} \right]$$

### ■ Initial value function

We create `valinit`, the initial value function. The default is a matrix of zeros.

```
valinit = Table[0, {i, 1, nx}, {j, 1, nθ}];
```

In many cases, one has a much better initial guess for the value function. This is the point where `valinit` should be set to that alternative. Otherwise, `valinit` will be the matrix of zeros.

---

## Value function iteration

The value function is really a two-dimensional array, `val`, where `val[i,j]` is the value function if the endogenous state is `xgrid[i]` and the current exogenous state  $\theta$  is `theta[j]`

We create two matrices, `valold` and `valnew`, to represent the old and new value functions in value function iteration. We initially make them equal to zero matrices.

```
valold = Table[0, {i,1,nx}, {j,1,ntheta}];  
valnew = Table[0, {i,1,nx}, {j,1,ntheta}];
```

`errmax` is an expression that computes the maximum difference between old and new value functions.

```
errmax := Max[Abs[valnew-valold]]
```

■ **Code that looks like basic C or Fortran**

The following defines the routine for one iteration of value function iteration. It takes the old value function, `valold`, and produces the new value function, `valnew`.

```
Bellman :=
(Do[valGuess = -1010; maxNext = nx; minNext = 1;
  (* The following steps through the choices for xnext,
  evaluating the value of each choice, and picking the max. *)
  Do[
    valtry = payoff[ix, iθ, xnext] +
      β Sum[ Prb[iθ, iθp] valold[[xnext, iθp]], {iθp, 1, nθ]];
    If[valtry > valGuess, valGuess = valtry],
    {xnext, minNext, maxNext]];
  (* Record the maximum value in the new value function. *)
  valnew[[ix, iθ] = valGuess,
  (* Cycle through all the endogenous and exogenous states. *)
  {ix, 1, nx}, {iθ, 1, nθ}]];
  (* Output is new value function plus
  difference between the old and new value functions. *)
  {valnew, Max[Abs[valnew - valold] ]}]])
```

*Mathematica* note: the symbol `:=` means that `Bellman` is defined to be the sequence of operations on the right-hand side. Each time you invoke `Bellman`, *Mathematica* will take the current choices for `valit`, `util`, etc., and execute the commands on the right hand side. If you want to change the payoff function, all you need to do is redefine `util` above.

The first iteration takes the initial guess as the old value function

```
valold = valinit; Bellman;  
errmax  
8.86927
```

Later iterations sets the old value function to be the last calculated valnew.

```
valold = valnew; Bellman;  
errmax
```

```
7.49966
```

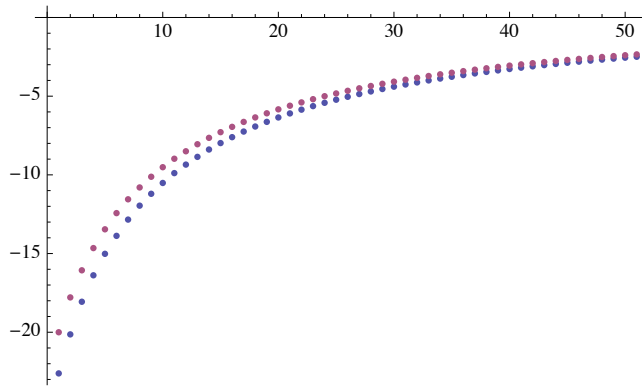


```
valold = valnew; Bellman;  
errmax
```

6.24719

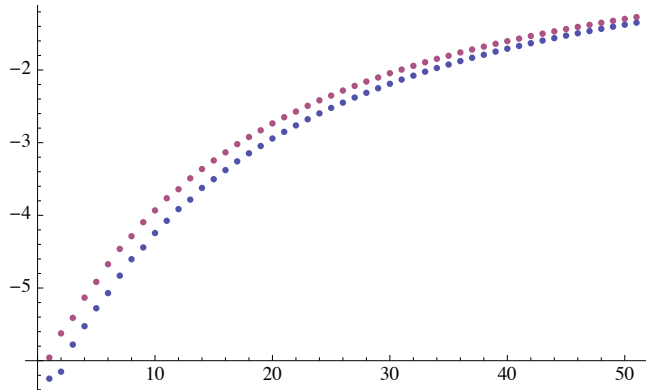
Let's graph the new value function, one curve for each  $\theta$ .

```
ListPlot[Transpose[valnew]]
```



Let's graph the differences between valold and valnew.

ListPlot[Transpose[valnew - valold]]



- **Code for Matlab - vectorized code**

We now vectorize our code. The vectorized versions should be faster since they take advantage of the Max operator on vectors.

```
Bellman2 :=
(Do[
  maxNext = nx; minNext = 1;
  (* In this version, we create the vectors
  corresponding to the possible current payoffs and
  the current possible discounted future values. *)
  tab1 =  $\beta$  Table[valold[[xnext, i $\theta$ ]],
    {xnext, minNext, maxNext}];
  tab2 = Table[payoff[ix, i $\theta$ , xnext],
    {xnext, minNext, maxNext}];
  (* We use the max command for vectors
  to get the new value. *)
  valGuess = tab1 + tab2;
  valnew[[ix, i $\theta$ ] = Max[valGuess],
    {ix, 1, nx}, {i $\theta$ , 1, n $\theta$ }];
{valnew, Max[Abs[valnew - valold]]})
```

Let's compare the time difference. Vectorization cuts time down by a third. The decomposition helps only a little bit but likely larger in other languages.

```
valold = valnew;  
Bellman; // Timing  
Bellman2; // Timing  
  
{0.414532, Null}  
  
{0.298405, Null}
```

- **Restricted search**

In most problems we have some rough bounds on what  $x_{next}$  is. We now incorporate that into the value function iteration

The parameter  $span$  tells Bellman to consider values of  $x_{next}$  between  $ix - span$  and  $ix + span$ .  $span$  is specified by user.

```

span = 5;
Bellman3 :=
  (Do[
    maxNext = Min[nx, ix + span]; minNext = Max[1, ix - span];
    (* In this version, we create the vectors
      corresponding to the possible current payoffs and
      the current possible discounted future values. *)
    tab1 =  $\beta$  Table[valold[[xnext, i $\theta$ ]],
      {xnext, minNext, maxNext}];
    tab2 = Table[payoff[ix, i $\theta$ , xnext],
      {xnext, minNext, maxNext}];
    (* We use the max command for vectors
      to get the new value. *)
    valGuess = tab1 + tab2;
    valnew[[ix, i $\theta$ ] = Max[valGuess],
      {ix, 1, nx}, {i $\theta$ , 1, n $\theta$ }];
    {valnew, Max[Abs[valnew - valold]]})
  )

```

Let's compare the time difference. Vectorization cuts time down by a third. Limiting the span of the search results in another one-third reduction.

```
valold = valnew;  
Bellman; // Timing  
Bellman2; // Timing  
Bellman3; // Timing  
  
{0.627356, Null}  
  
{0.453327, Null}  
  
{0.102611, Null}
```

- Value function iteration

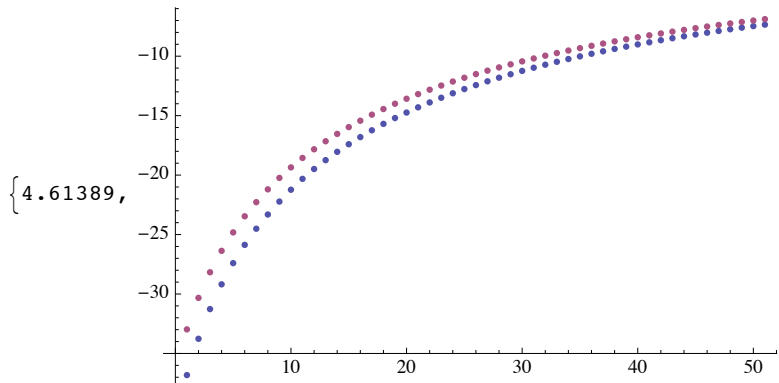
The following defines a complete cycle of the basic iteration

```
ValIter :=
  (valold = valnew; {valnew, errmax} = Bellman3;)
```

The following defines a complete value function iteration where we reset valold, evaluate Bellman, and compute the maximum difference, which is proportional to the error.

```
ValueIterPrint := (valold = valnew; {valnew, errmax} = Bellman3;
  Print[{errmax, ListPlot[Transpose[valnew]]}])
```

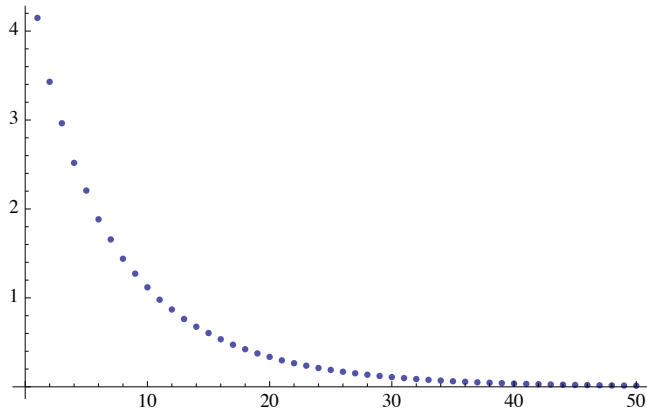
```
ValueIterPrint
```





Let's do a lot of iterations and watch the error decline in a table

```
ValueIterErr:= (valold = valnew; {valnew,errMax}=Bellman3; errMax)  
valold=valinit; tab=Table[ValueIterErr,{50}];  
ListPlot[tab]
```



A log plot will reveal the loglinear nature of the decline

```
ListPlot[Log[10, tab]]
```

