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, payoff[ix, iθ, xnext] is the payoff in the current period if the current state is (ix, iθ) and xnext is the next grid point where ix, iθ, and xnext are indices of the state space, and 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

θ: The exogenous state

The states, x and θ, 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:

xgrid: The vector of grid points for the endogenous state.

nx: Length of xgrid

θgrid: The vector of grid points for the exogenous state

nθ: Length of θgrid

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 xgrid[ix]. Similarly when we say the state is (ix, iθ).

Markov transition matrix:

Prb[iθ, jθ]: The probability of the exogenous state being jθ in the next period if the current exogenous state is iθ.

Payoff function:

payoff[ik, iθ, ikp]: the payoff in a period if the current state is (ik, iθ) 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 nk × nθ matrix	→	→	→	Bellman
Choose nk	→	→	→	Bellman
↓				
Define kgrid				
↓				
Define kgridf				
↓				
Define payoff	→	→	→	Bellman
↑				
Define θgridf				
↑				
Define θgrid	→	Define Prb	→	Bellman
↑				
Choose nθ	→	→	→	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
kmin (kmax): 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.]
kgrid: the vector of permissible capital stocks, a subset of [kmin, kmax]
nk: length of kgrid

Construct kgrid: We will construct a uniform grid. κ is the uniform difference between successive states. The possible choices of net investment will be integer multiples of κ.

```
                kmax - kmin
κ := ──────────────────────;
                nk - 1
kgrid := Table[kmin + (i - 1) κ, {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) κ
```

■ Exogenous states

The productivity states move exogenously.

θ:	productivity state - the exogenous state variable
θmin (θmax):	the minimum (maximum) value for θ.
θgrid:	the vector of possible θ values, a subset of [θmin, θmax]
nθ:	length of θgrid

Construct θgrid: We will construct a uniform grid. $\delta\theta$ is the uniform difference between successive states.

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

```
θgrid := Table[θmin + (i - 1) δθ, {i, 1, nθ}];
```

Here again the index iθ is the index in the θ grid that contains the value for θgrid[iθ].

We define a function that takes the index of a theta grid point, iθ, and computes the θ value it represents

```
θgridf[iθ_] := θmin + δθ (iθ - 1)
```

Nonuniform grids can also be used. The key requirements is that θgrid contain the list of exogenous states and θgridf is a function mapping indices to values for the exogenous state.
θgrid is passed to Bellman. θgridf 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 = 1.2; nk = 221;
```

Output kgrid and kgridf.

```
kgrid
```

```
{0.1, 0.105, 0.11, 0.115, 0.12, 0.125, 0.13, 0.135, 0.14, 0.145, 0.15, 0.155, 0.16, 0.165, 0.17, 0.175,
 0.18, 0.185, 0.19, 0.195, 0.2, 0.205, 0.21, 0.215, 0.22, 0.225, 0.23, 0.235, 0.24, 0.245, 0.25, 0.255,
 0.26, 0.265, 0.27, 0.275, 0.28, 0.285, 0.29, 0.295, 0.3, 0.305, 0.31, 0.315, 0.32, 0.325, 0.33, 0.335,
 0.34, 0.345, 0.35, 0.355, 0.36, 0.365, 0.37, 0.375, 0.38, 0.385, 0.39, 0.395, 0.4, 0.405, 0.41, 0.415,
 0.42, 0.425, 0.43, 0.435, 0.44, 0.445, 0.45, 0.455, 0.46, 0.465, 0.47, 0.475, 0.48, 0.485, 0.49,
 0.495, 0.5, 0.505, 0.51, 0.515, 0.52, 0.525, 0.53, 0.535, 0.54, 0.545, 0.55, 0.555, 0.56, 0.565, 0.57,
 0.575, 0.58, 0.585, 0.59, 0.595, 0.6, 0.605, 0.61, 0.615, 0.62, 0.625, 0.63, 0.635, 0.64, 0.645, 0.65,
 0.655, 0.66, 0.665, 0.67, 0.675, 0.68, 0.685, 0.69, 0.695, 0.7, 0.705, 0.71, 0.715, 0.72, 0.725, 0.73,
 0.735, 0.74, 0.745, 0.75, 0.755, 0.76, 0.765, 0.77, 0.775, 0.78, 0.785, 0.79, 0.795, 0.8, 0.805,
 0.81, 0.815, 0.82, 0.825, 0.83, 0.835, 0.84, 0.845, 0.85, 0.855, 0.86, 0.865, 0.87, 0.875, 0.88,
 0.885, 0.89, 0.895, 0.9, 0.905, 0.91, 0.915, 0.92, 0.925, 0.93, 0.935, 0.94, 0.945, 0.95, 0.955, 0.96,
 0.965, 0.97, 0.975, 0.98, 0.985, 0.99, 0.995, 1., 1.005, 1.01, 1.015, 1.02, 1.025, 1.03, 1.035, 1.04,
 1.045, 1.05, 1.055, 1.06, 1.065, 1.07, 1.075, 1.08, 1.085, 1.09, 1.095, 1.1, 1.105, 1.11, 1.115, 1.12,
 1.125, 1.13, 1.135, 1.14, 1.145, 1.15, 1.155, 1.16, 1.165, 1.17, 1.175, 1.18, 1.185, 1.19, 1.195, 1.2}
```

```
kgridf[ik]
```

```
0.1 + 0.005 (-1 + ik)
```

Choose parameters for θ grid.

```
θmin = 0.95; θmax = 1.05; nθ = 2;
```

Output θgrid and θgridf.

```
θgrid
```

```
{0.95, 1.05}
```

```
θgridf[iθ]
```

```
0.95 + 0.1 (-1 + iθ)
```

■ Transition rule for θ values

We next choose the probabilities of the transition matrix for θ.
Prb[i,j] is the probability that θ moves from θgrid[i] value to θgrid[j].

```
Prb[1, 1] = Prb[2, 2] = 2 / 3;
Prb[1, 2] = 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_, θ_] := θ (1 - β) k^α / (β α)
```

We have defined the production function, f[k,θ], 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 θ=θgrid[iθ], then fgrid[ik,iθ] = f[k,θ].

```
fgrid[ik_, iθ_] := f[kgridf[ik], θgridf[iθ]]
```

In this case, the kgrid and θgrid grids are not sent to Bellman since we 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 θ_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_] = c^(1-γ) / (1 - γ);  (* γ>0 in order for u[c] to be concave. *)
util[cc_] = If[cc > .001, u[cc], -10^10] ;
(* 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, iθ].
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, iθ) and choice ikp is utility of the resulting consumption, which is

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

Parameter Values

```
α = .25; γ = 3.0; β = .9;
```

```
payoff[ik, iθ, ikp]
```

```
If[-0.005 (-ik + ikp) + 0.444444 (0.1 + 0.005 (-1 + ik))^0.25 (0.95 + 0.1 (-1 + iθ)) > 0.001,
 u[-0.005 (-ik + ikp) + 0.444444 (0.1 + 0.005 (-1 + ik))^0.25 (0.95 + 0.1 (-1 + iθ))], -10^10]
```

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 θ is θgrid[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,nθ}];
valnew = Table[0,{i,1,nx},{j,1,nθ}];
```

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 = -10^10; 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
```

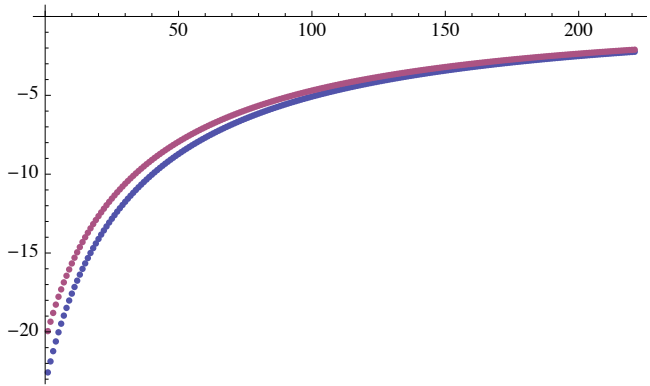
```
7.40848
```

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

```
6.29323
```

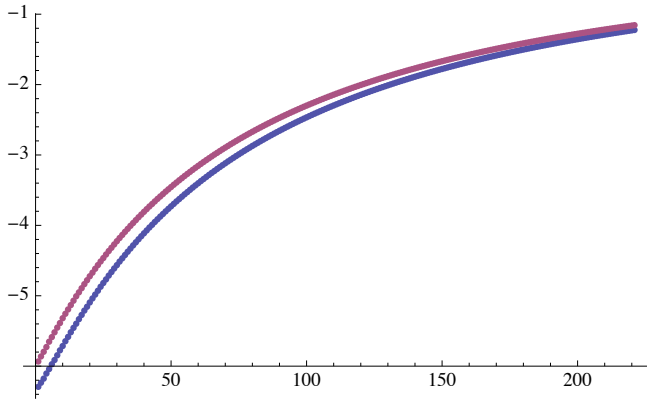
Let's graph the new value function, one curve for each θ.

```
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 = β Table[valold[[xnext, iθ]],
      {xnext, minNext, maxNext}];
   tab2 = Table[payoff[ix, iθ, xnext],
     {xnext, minNext, maxNext}];
   (* We use the max command for vectors
    to get the new value. *)
   valGuess = tab1 + tab2;
   valnew[[ix, iθ]] = Max[valGuess],
   {ix, 1, nx}, {iθ, 1, nθ}];
  {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
```

```
{7.93863, Null}
```

```
{5.44901, Null}
```

■ Restricted search

In most problems we have some rough bounds on what xnext is. We now incorporate that into the value function iteration

The parameter span tells Bellman to consider values of xnext 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 = β Table[valold[[xnext, iθ]],
      {xnext, minNext, maxNext}];
   tab2 = Table[payoff[ix, iθ, xnext],
     {xnext, minNext, maxNext}];
   (* We use the max command for vectors
    to get the new value. *)
   valGuess = tab1 + tab2;
   valnew[[ix, iθ]] = Max[valGuess],
   {ix, 1, nx}, {iθ, 1, nθ}];
  {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
```

```
{7.65037, Null}
```

```
{5.45601, Null}
```

```
{0.399849, Null}
```

■ Value function iteration

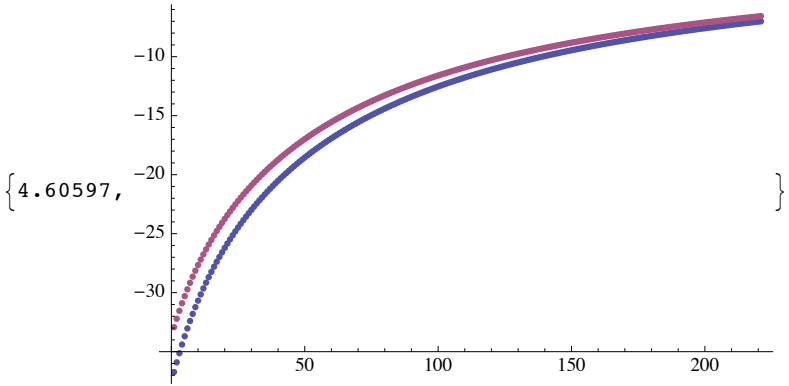
The following defines a complete cycle of the basic iteration

```
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.

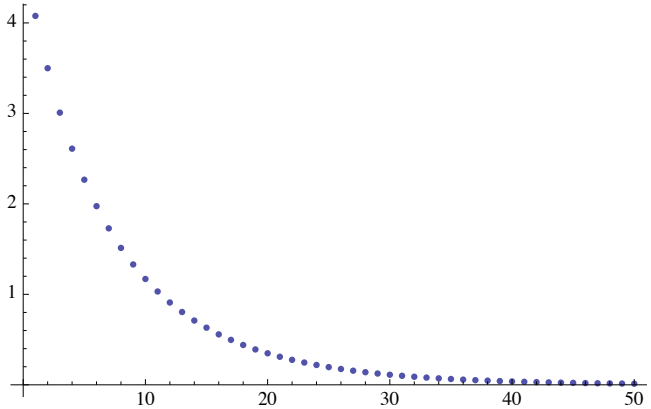
```
ValueIterPrint:= (valold = valnew; {valnew,errmax}=Bell
Print[{errmax,ListPlot[Transpose[valnew]]}])
```

```
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]]
```