Discrete Dynamic Programming: Gauss-Seidel

In[3237]:=
```
x = 0; Remove["Global`*"]
```

Outline of notebook

This notebook consists of two basic parts. First, I specify a particular problem, a simple 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.

This notebook uses Gauss-Seidel updating of the value function. This is appropriate when solving infinite-horizon problems, for which this is much faster than value-function iteration.

Initializations: An Optimal Growth Problem

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

Grids:

We specify a set of grid points for the endogenous state in kgrid
We specify a set of grid points for the exogenous state in θgrid

Payoff function:

We specify a function, payoff[ik, iθ, ikp], which computes the period payoff if in state (ik, iθ) and choose to move to endogenous state ikp tomorrow.

Transition probabilities:

We specify a Markov transition matrix, Prb[iθ, jθ], that gives the probability of being in exogenous state jθ if the current exogenous state is iθ.

Grid of states

We use a simple growth model as an example.

In[3238]:=
```
(* kgrid is the grid used in our Bellman code, and kindex indexes that list *)
kgrid = Table[kmin + (i - 1) κ, {i, 1, nk}];
kindex = Range[1, nk];
```

Table: Iterator {i, 1, nk} does not have appropriate bounds.

Range: Range specification in Range[1, nk] does not have appropriate bounds.

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

In[3240]:=
```
kgridf[ik_] = kmin + (ik - 1) κ
```

Out[3240]=
```
kmin + (-1 + ik) κ
```

Exogenous states:

The productivity states move exogenously. Create vector, θgrid, of productivity states

In[3241]:=
```
nθ = 2;
```

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

In[3242]:=
```
θgrid[iθ_] = θmin + δθ (iθ - 1);
```

Construct payoff function

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

Production function

In[3243]:=
```
f[k_, θ_] = k + θ (1 - β) k^α / (β α)
```

Out[3243]=
$$k + \frac{k^{\alpha}\ (1-\beta)\ \theta}{\alpha\ \beta}$$

We need to define the production function in terms of the state indices

In[3244]:=
```
fg[ik_, iθ_] = f[kgridf[ik], θgrid[iθ]] // Expand
```

Out[3244]=
$$\text{kmin} - \kappa + \text{ik}\ \kappa + \frac{\delta\theta\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha} - \frac{\text{i}\theta\ \delta\theta\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha} - \frac{\delta\theta\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha\ \beta} + \frac{\text{i}\theta\ \delta\theta\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha\ \beta} - \frac{\theta\text{min}\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha} + \frac{\theta\text{min}\ (\text{kmin} + (-1+\text{ik})\ \kappa)^{\alpha}}{\alpha\ \beta}$$

Utility function

In[3245]:=
```
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. *)
```

Define the payoff function in terms of

(ik, iθ) the indices of the current state, and

ikp index of kplus, the next periods capital stock, our choice variable

```
payoff[ik_, iθ_, ikp_] = util[fg[ik, iθ] - κ ikp];
```

Note that κ ik is the current capital stock, κ (ikp - ik) is the net investment, and κ ikp is next period's capital stock

Parameter Values

capital grid

In[3248]:=
```
(*kmin (kmax) is the minimum (max) capital stock*)
kmin = .5;  kmax = 1.5;
(* nx=nk=number of capital stocks *)
nx = nk = 101;
(* κ is the uniform difference between successive states.The
 possible choices of net investment will be integer multiples of κ. *)
κ = (kmax - kmin)/(nk - 1);
```

θ grid

$\delta\theta$ is the step size in θ values, and θmin is the minimum value.

In[3251]:=
```
δθ = 0.2;
θmin = 1 - δθ / 2;
```

Utility function

In[3253]:=
```
γ = 2; β = .95;
```

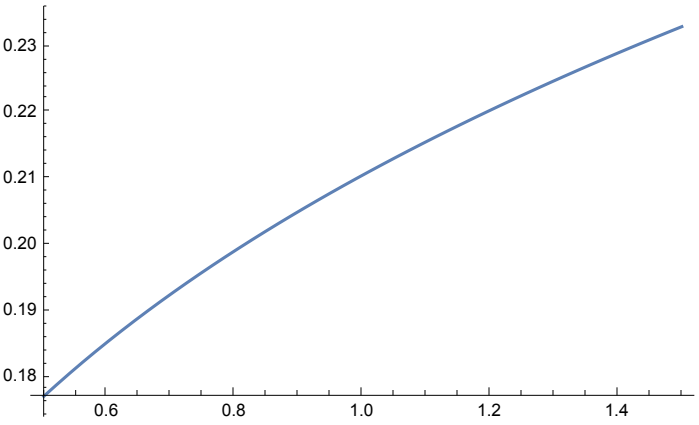
Production function

In[3254]:=
```
α = .25; γ = 2; β = .95;
```

Plot net output

In[3255]:=
```
Plot[f[k, 1] - k, {k, kmin, kmax}]
```

Out[3255]=



Define marginal product of capital and check that k = θ = 1 is the deterministic steady state

In[3256]:=
```
fk[k_, θ_] = D[f[k, θ], k];
β fk[1, 1]
```

Out[3257]=
```
1.
```

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 the θgrid[[i]] value to the θgrid[[j]] value. We initially make a degenerate choice

In[3258]:=
```
Prb[1, 1] = Prb[2, 2] = 3 / 4;
Prb[1, 2] = Prb[2, 1] = 1 / 4;
```

Number of iterations

In[3260]:=
```
numIts = 20;
```

Initial guess

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

In[3261]:=
```
valnew = valold = valinit = Table[0, {i, 1, nx}, {j, 1, nθ}];
```

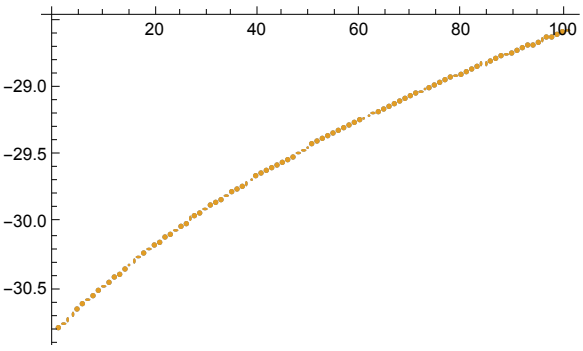
In many cases, one has a much better initial guess for the value function. This is the point where valinit should be set to that alternative. Otherwise, valinit will be the matrix of zeros.

In[3262]:=
```
Clear[BellmanInit]
fbell[iz_] := payoff[iz, iθ, iz] + β Sum[ Prb[iθ, iθp] valold[[iz, iθp]], {iθp, 1, nθ}];
BellmanInit :=
 (Do[
   (* We record the value of zero investment forever in the theta state. *)
   valnew[[ix, iθ]] = payoff[ix, 1, ix] / (1 - β),
   (* We cycle through all the endogenous and exogenous states. *)
   {ix, 1, nx}, {iθ, 1, nθ}];
  (* The output is the new value function plus
   the difference between the old and new value functions. *)
  {valnew, Max[Abs[valnew - valold]]})
```

In[3265]:=
```
valinit = BellmanInit[[1]];
```

In[3266]:=
```
ListPlot[Transpose[valinit]]
```

Out[3266]=



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 kgrid[i] and the current exogenous state θ is θgrid[j]

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

In[3267]:=
```
errmax := Max[Abs[valnew-valold]]
```

Code that looks like basic C or Fortran, simple GJ

The following defines the routine for one iteration of Gauss-Seidel updating. It takes the old value function, valold, and initializes the new value function, valnew, to be valnew=valold. valnew is then used in each Bellman update for each state.

In[3268]:=
```
Clear[BellmanGJ]
fbell[iz_] := payoff[ix, iθ, iz] + β Sum[ Prb[iθ, iθp] valold[[iz, iθp]], {iθp, 1, nθ}];
BellmanGJ :=
 (Do[
   (* The following steps through the choices for xnext,
   evaluating the value of each choice, and picking the max. *)
   vals = fbell /@ kindex;
   (* We record the maximum value in the new value function. *)
   valnew[[ix, iθ]] = Max[vals],
   (* We cycle through all the endogenous and exogenous states. *)
   {ix, 1, nx}, {iθ, 1, nθ}];
  (* The output is the new value function plus
   the 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 righthand 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

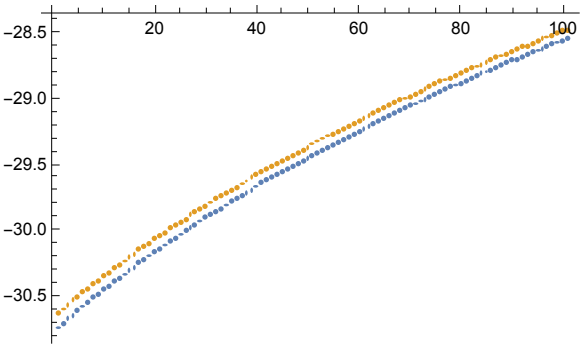
In[3271]:=
```
valold = valinit; valnew = valold;
BellmanGJ; // AbsoluteTiming
errmax
```

Out[3272]= {0.644746, Null}

Out[3273]= 0.161551

In[3274]:=
```
ListPlot[Transpose[valnew]]
```

Out[3274]=



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

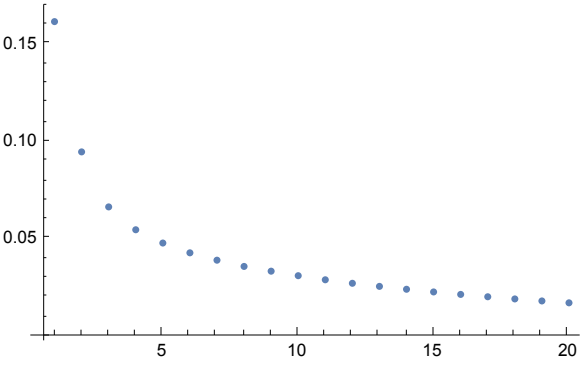
In[3275]:=
```
errr = Table[0, {numIts}];
```

In[3276]:=
```
valnew = valinit;
```

In[3277]:=
```
Do[valold = valnew; erer = BellmanGJ[[2]];
 errr[[i]] = erer, {i, 1, numIts}]
```

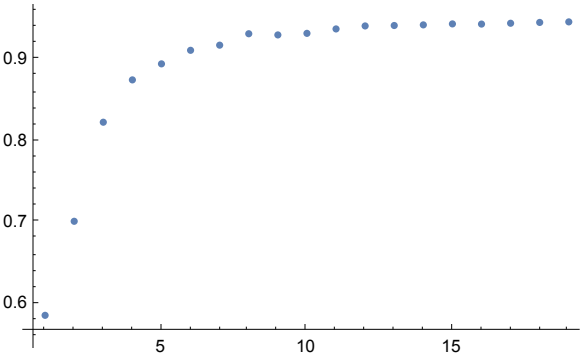
In[3278]:=
```
ListPlot[errr]
```

Out[3278]=



In[3279]:=
```
convergenceGJ = Take[errr, {2, numIts}] / Take[errr, {1, numIts - 1}] // ListPlot
```

Out[3279]=



Compute the distance to true value function

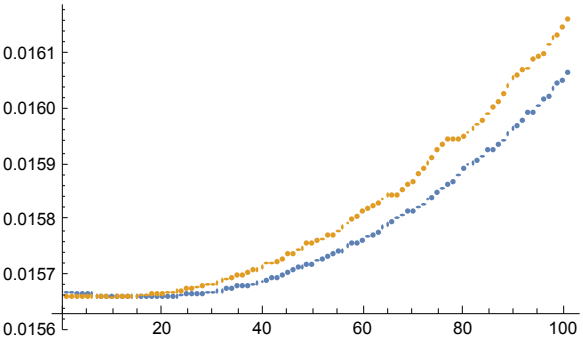
In[3280]:=
```
valold = valnew; BellmanGJ;
GJerror = errmax / (1 - β)
```

Out[3281]=
```
0.323222
```

Let's graph the differences between valold and valnew.

In[3282]:=
```
GJdiffplot = ListPlot[Transpose[valnew - valold]]
```

Out[3282]=



Code that looks like basic C or Fortran, simple GS

The following defines the routine for one iteration of Gauss-Seidel updating. It takes the old value function, valold, and initializes the new value function, valnew, to be valnew=valold. valnew is then used in each Bellman update for each state.

In[3283]:=
```
Clear[BellmanGS, valnew, valold]
fbell[iz_] := payoff[ix, iθ, iz] + β Sum[ Prb[iθ, iθp] valnew[[iz, iθp]], {iθp, 1, nθ}];
BellmanGS :=
 (Do[
   (* The following steps through the choices for xnext,
   evaluating the value of each choice, and picking the max. *)
   (* The key Gauss-Seidel difference:
      valnew is used instead of valold on the RHS of the valtry computation. *)
   vals = fbell /@ kindex;
   (* We record the maximum value in the new value function. *)
   valnew[[ix, iθ]] = Max[vals],
   (* We cycle through all the endogenous and exogenous states. *)
   {iθ, 1, nθ}, {ix, 1, nx}];
  (* The output is the new value function plus
   the 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 righthand 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
Later iterations sets the old value function to be the last calculated valnew.

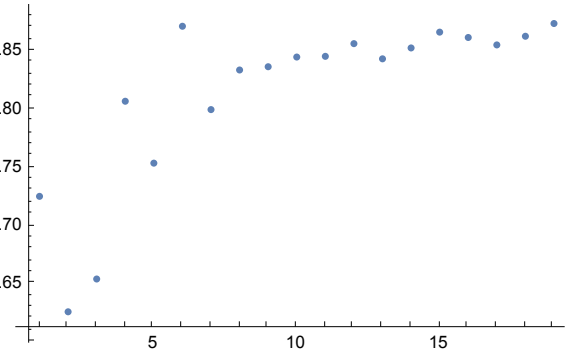
In[3286]:=
```
errr = Table[0, {numIts}];
```

In[3287]:=
```
valnew = valinit;
```

In[3288]:=
```
Do[valold = valnew; erer = BellmanGS[[2]];
 errr[[i]] = erer, {i, 1, numIts}]
```

In[3289]:=
```
convergenceGS = Take[errr, {2, numIts}] / Take[errr, {1, numIts - 1}] // ListPlot
```

Out[3289]=



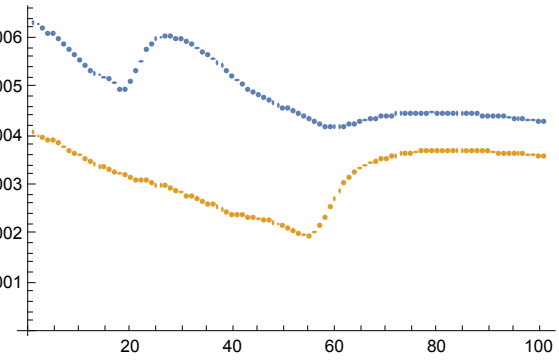
In[3290]:=
```
valold = valnew; BellmanGJ;
GSerror = errmax / (1 - β)
```

Out[3291]=
```
0.126451
```

Let's graph the differences between valold and valnew.

In[3292]:=
```
GSdiffplot = ListPlot[Transpose[valnew - valold]]
```

Out[3292]=



Code that looks like basic C or Fortran, Alternating Sweep GS

The following defines the routine for two passes of Gauss-Seidel through the states, one where we go from first to last state, and the next where we go from last to first.

In[3293]:=
```
Clear[BellmanGSAlt, valold, valnew]
BellmanGSAlt :=
 (Do[valGuess = -10^10; maxNext = nx; minNext = 1;
   (* We compute the value of each choice for xnext, and pick the max. *)
   (* In the first pass, ix goes from 1 to nx. *)
   Do[
    valtry =
     payoff[ix, iθ, xnext] + β Sum[ Prb[iθ, iθp] valnew[[xnext, iθp]], {iθp, 1, nθ}];
    If[valtry > valGuess, valGuess = valtry],
    {xnext, minNext, maxNext}];
   (* We record the maximum value in the new value function. *)
   valnew[[ix, iθ]] = valGuess,
   (* We cycle through all the endogenous and exogenous states. *)
   {ix, 1, nx}, {iθ, 1, nθ}];
  valmid = valnew;
  (* In the second pass, ix goes from nx to 1. *)
  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] valnew[[xnext, iθp]], {iθp, 1, nθ}];
    If[valtry > valGuess, valGuess = valtry],
    {xnext, minNext, maxNext}];
   (* We record the maximum value in the new value function. *)
   valnew[[ix, iθ]] = valGuess,
   (* We cycle through all the endogenous and exogenous states. *)
   {iθ, 1, nθ}, {ix, nx, 1, -1}];
  (* The output is the new value function plus
   the 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 righthand 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

In[3295]:=
```
valold = valinit; valnew = valold; BellmanGSAlt;
errmax
```

Out[3296]= 1.04767

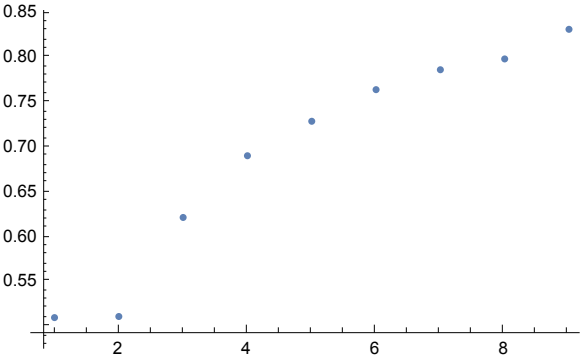
In[3297]:=
```
errrgs = Table[0, {i, 1, numIts / 2}];
```

In[3298]:=
```
valnew = valinit;
```

In[3299]:=
```
Do[valold = valnew; erer = BellmanGSAlt[[2]];
 errrgs[[i]] = erer, {i, 1, numIts / 2}]
```

In[3300]:=
```
convergenceGSAlt =
 Take[errrgs, {2, numIts / 2}] / Take[errrgs, {1, numIts / 2 - 1}] // Sqrt // ListPlot
```

Out[3300]=



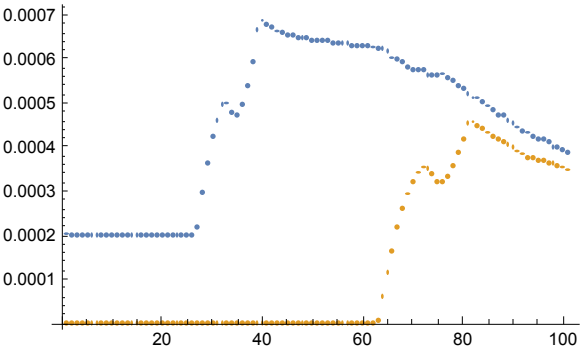
In[3301]:=
```
valold = valnew; BellmanGJ;
GSalterror = errmax / (1 - β)
```

Out[3302]= 0.0137557

Let's graph the differences between valold and valnew.

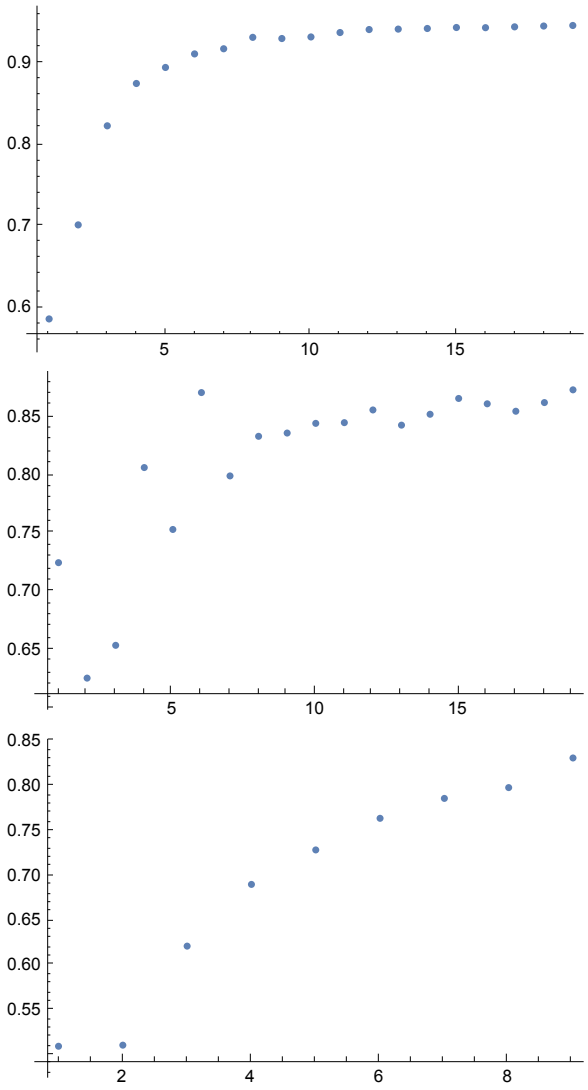
In[3303]:=
```
GSaltdiffplot = ListPlot[Transpose[valnew - valold]]
```

Out[3303]=



In[3311]:= `{convergenceGJ, convergenceGS, convergenceGSAlt} // Column`

Out[3311]=



In[3305]:= `{GJerror, GSerror, GSalterror}`

Out[3305]= `{0.323222, 0.126451, 0.0137557}`

In[3312]:=
```
{GJdiffplot, GSdiffplot, GSaltdiffplot} // Column
```

Out[3312]=