COMPUTATIONAL ECONOMICS

Revised Edition

Dynamic Programming with Discrete State Space

David A. Kendrick
Ruben P. Mercado
Hans M. Amman

1 Introduction

In this chapter we return to the growth model; however this time instead of using nonlinear programming to solve the model we employ a dynamic programming technique with discrete state space. This means that the state variable, rather than taking on any value in a range is restricted to a grid point of values. Thus, for example, the capital stock in a growth model solved with nonlinear programming might be allowed to take on any value in the range from 100 to 180. In the same growth model solved with dynamic programming with discrete state space the capital stock would only be allowed to take on the values 100, 200, 300, 400, 500, 600, 700 or 800. This at first might seem to be a restriction that is difficult to live with; however, modern computers are sufficiently fast that the grid of points can be finely divided and instead of the eight points above have 800 points like $(100, 101, 102, ..., 800)$ or maybe even 8000 points like (100.0, 100.1, 100.2,...,800.0).

More about the methodology later - first let's summarize the growth model that was used in the Chapter on the *Ramsey Model* and modify it somewhat for use in this chapter.

2 The Model

The production side of the economy is specified in a stylized form by means of an aggregate production function

$$Y_t = \theta K_t^{\alpha} \tag{1}$$

where

$$\begin{aligned} Y_t &= \text{output in period} \\ \theta &= \text{a technology parameter} \\ K_t &= \text{the capital stock in period} \\ \alpha &= \text{exponent of capital in the production function} \end{aligned}$$

This is the widely used form of a production function except that the function usually includes both capital and labor inputs. However, for the sake of simplicity, the production function in this model includes only capital. Consider next the capital accumulation constraint

$$K_{t+1} = (1 - \delta)K_t + Y_t - C_t \tag{2}$$

where

$$\begin{aligned} \delta &= \text{depreciation rate} \\ C_t &= \text{consumption in period } t \end{aligned}$$

which says that the capital stock next period will be the one minus the depreciation rate of the capital stock this period plus the difference between output and consumption which is saving or investment. We did not use the depreciation rate in the model in Chapter 1 so for the time being we will set it equal to zero in order to have a model in this chapter that is comparable to the one in Chapter on the *Ramsey Model*. Thus equation (2) becomes

$$K_{t+1} = K_t + Y_t - C_t \tag{3}$$

Also we will make use of the income identity

$$Y_t = C_t + I_t \tag{4}$$

$$I_t \quad = \quad \text{investment in period } t$$

Then comparing equation (3) to equation (4) we see that in this model investment is equal to the change in the capital stock from one period to the next, i.e.

$$K_{t+1} = K_t + I_t \tag{5}$$

For the methodology of this chapter it is convenient to substitute equation (1) into equation (3) to obtain

$$K_{t+1} = K_t + \theta K_t^{\alpha} - C_t \tag{6}$$

so that the model contains a single state equation in the state variable and a single control variable. In addition, the model has an initial condition that specifies the size of the capital stock in the initial period.

$$K_0 = \text{given}$$

The model also includes a terminal condition that fixes a minimum amount of capital that must be left to the next generation after the time horizon covered by the model.

$$K_N \geq K^*$$

where

$$K^* \quad = \quad \begin{array}{l}\text{a lower bound on the amount of capital} \\ \text{required in the terminal period } N\end{array}$$

Finally, the model has a criterion function that is the discounted value of the utility that is obtained from consumption over all of the periods covered by the model. It is written in two steps. First the utility in each period is defined as

$$U(C_t) = \frac{1}{1-\tau} C_t^{1-\tau} \tag{7}$$

where

$$\begin{aligned} U(C_t) &= \text{the utility in period} t \text{as a function of consumption in that period} \\ \tau &= \text{a parameter in the utility function} \end{aligned} \tag{8}$$

Then the sum of the discounted utilities is specified as

$$J = \sum_{t=0}^{N-1} \beta^t U(C_t) \tag{9}$$

where

$$\begin{aligned} J &= \text{the criterion value} \\ \beta &= \text{the discount factor} \end{aligned}$$

and the substitution of equation (7) into equation (9) yields the criterion function

$$J = \sum_{t=0}^{N-1} \beta^t \frac{1}{1-\tau} C_t^{1-\tau} \tag{10}$$

In summary, the model consists of the criterion function (10), the capital accumulation equation (6) and the initial and terminal conditions (7) and (7) and can be stated as find

$(C_0, C_1, ..., C_{N-1})$ to maximize

$$J = \sum_{t=0}^{N-1} \beta^t \frac{1}{1-\tau} C_t^{1-\tau} \tag{11}$$

subject to

$$\begin{aligned} K_{t+1} &= K_t + \theta K_t^{\alpha} - C_t & (12) \\ K_0 &= \text{given} & (13) \\ K_N &\geq K^* & (14) \end{aligned}$$

So the essential problem is to choose those levels of consumption, over the time periods covered by the model that strike the right balance between consumption and the difference between output and consumption which is investment . Lower consumption in any given period means less utility in that period but more savings and therefore larger capital stocks and more production. The parameters used here are the same as those used in the model in Chapter on the *Ramsey Model*, namely $\alpha = 0.33$, $\beta = 0.98$, $\tau = 0.5$, $K_0 = 7.0$ and $K^* = 9.1$.

Also the model can be written in a more general form as find $(c_0, c_1, ..., c_{N-1})$ to maximize

$$V = \sum_{t=0}^{N-1} \beta^t u(c_t) \tag{15}$$

subject to

$$\begin{aligned} k_{t+1} &= f(k_t, c_t) & (16) \\ K_0 &= \text{given} & (17) \\ k_N &\geq k^* & (18) \end{aligned}$$

where k_t is the state vector with the single element and c_t is the control vector with the single element . The growth model in equations (15)-(18) is a dynamic programming problem and can be solved with dynamic programming with discrete state space. Therefore we turn next to a general discussion of that methodology.

3 The Methodology

The methodology of dynamic programming with discrete state space has a nice analogy to a method for solving Rubik's cube. This popular puzzle is a plastic block with nine color squares on each side and the ability to rotate the elements of the block. The object is to begin with a mix of colors on each side and rotate the elements of the cube until all the faces on each side are the same color.

I gave one of these puzzles to my son when he was a youngster and after a week or so he announced that he had learned how to solve it. I was greatly surprised and asked him to show me. He tossed me the cube to show me that it was indeed in a state with multiple colored faces on each side. He then retreated to his room and within a few minutes returned to show me the cube with the same color faces on each side.

It turned out that he had solved it backwards. He began with the solution with the same colors on each side and rotated it in a few steps that he memorized so that it appeared to me to be very difficult to solve. However, it was easy for him to simply reverse the steps and get back to the solution. So it is with dynamic programming - the way to solve these problems is to work backward. In this light another analogy is useful. Consider a plane flying across the Atlantic from New York to London facing a certain pattern of wind conditions across the ocean. The way to find the best route is not to begin with New York but rather to begin from London and work backwards across the Atlantic.

One begins at Heathrow Airport and backs out to an array of points some distance away from the airport. Then one finds the optimal cost from each of those points to the airport. Next one backs away another step and finds the optimal cost from each of those points to the closer set of points given the knowledge of the optimal cost of each of the closer points on in to Heathrow. The end goal of this solution procedure is to provide the pilots with a map that shows them in an array of grid points and tells them which direction they should go at each grid point in order to get across the ocean with the least fuel use given the wind patterns that day. Thus two key elements of this methodology are the grid of points and a solution procedure that works backwards. First consider the grid. In our growth model we do not have a two dimensional ocean but rather a single dimension for the capital stock so we need to create a grid for the capital stock in each time period. Therefore we define the grid for the capital stock in each time period. For our growth model it is convenient to set the first element in this grid to the given initial capital stock and the last one to the terminal capital stock.

The second key element is the backward solution method. For this we draw on the dynamic programming approach outlined by Bertsekas (2005, Chapter 1) (see also Adda and Cooper (2003, Chapter 2)). The key is the value function which is defined as

$$V_t(k_t) = \max_{c_t} \{u(c_t) + \beta V_{t+1}(k_{t+1})\} \tag{19}$$

Moreover we will use this value function in a recursion by starting with its value in the last

period, and working backwards to solve for and we will do these calculations over the set of grid points in the capital stock. The goals of this procedure will be to obtain a map that shows at each point in time and at each grid point what is the best grid point to move to in the next time period.

4 The Program

The first step in the program is to define the parameters that are used in the model with the **MATLAB** statements

```
alpha = 0.33
beta = 0.98
theta = 0.3
tau = 0.5
```

Also the initial capital stock k0 and the terminal target capital stock are defined with the statements.

```
k0 = 7
kN = 9.1
```

The program can be divided into three segments:

1. the definition of the grid and mesh for the capital stocks and other variables
2. the backward loop in the value function
3. the forward loop in the capital accumulation equation

The full program is listed in Appendix A.

4.1 Definition of the Grid and the Mesh

As discussed above, one key to the program is to define a grid for the state variable over the range of values that it is likely to take on. For the problem at hand it is useful to define the lower value of the grid as the initial capital stock `k0` and the upper end of the grid as the terminal capital stock `kN`. Then if the grid is to consist of n points one can define the step size, step, between the points in the grid with the MATLAB statements

```
n = 5
step = (kN-k0) / (n-1)
k_grid = [k0:step:kN]
```

The third of the three statements above creates a row vector, `k_grid`, whose first value is `k0` and last value is `kN` with `n-2` steps in between. For the current model `k0 = 7.0` and `kN = 9.1` so the row vector `k_grid` becomes

```
k_grid = [ 7.000  7.525  8.050  8.575  9.100]
```

Also, we can define a grid for output that corresponds to the same points in the grid for capital stocks. This is done with the statement

```
y_grid = theta * (k_grid.^alpha)
```

In this statement the combination of symbols .ˆis used to raise an array to a power. Thus every element in the `k_grid` row vector is raised to the power alpha. The next task in the program is to define investment. For this we use equation (5) above and solve it for I_t to obtain

$$I_t = K_{t+1} - Kt \tag{20}$$

So we need to create a grid for $Kt + 1$ and subtract from it a grid for K_t. The first step in doing this is to create a two dimensional array for capital that is filled with n rows of the capital grid. We call this the capital *mesh* and define with the **MATLAB** statements

```
aux = ones(n,1)
kt_mesh = aux * k_grid
```

The first of these two statements creates a column vector of ones that is n elements long. The second statement multiplies that column vector times the row vector `k_grid`. Since a column vector multiplied by a row vector yields a matrix the result of this operation is to create an n x n matrix in which each row is the row vector `k_grid`, i.e.

```
7.0000    7.5250    8.0500    8.5750    9.1000
7.0000    7.5250    8.0500    8.5750    9.1000
7.0000    7.5250    8.0500    8.5750    9.1000
7.0000    7.5250    8.0500    8.5750    9.1000
7.0000    7.5250    8.0500    8.5750    9.1000
```

Next we transpose the `kt_mesh` matrix and call it `kt1_mesh`. This is done with the statement

```
kt1_mesh = (kt_mesh)'
```

which yields the matrix

```
7.0000    7.0000    7.0000    7.0000    7.0000
7.5250    7.5250    7.5250    7.5250    7.5250
8.0500    8.0500    8.0500    8.0500    8.0500
8.5750    8.5750    8.5750    8.5750    8.5750
9.1000    9.1000    9.1000    9.1000    9.1000
```

The `kt1_mesh` matrix has a column vector version of `k_grid` in each column of the matrix. The difference between these two matrices then provides a mesh for investment by using the **MATLAB** statement

```
i_mesh = kt1_mesh - kt_mesh
```

which is shown below

```
0        -0.5250   -1.0500   -1.5750   -2.1000
0.5250         0   -0.5250   -1.0500   -1.5750
1.0500    0.5250         0   -0.5250   -1.0500
1.5750    1.0500    0.5250         0   -0.5250
2.1000    1.5750    1.0500    0.5250         0
```

The (i,j) elements in the `i_mesh` matrix then represent the amount of investment that would have to be incurred to move from the j position in the `k_grid` vector to position i in that vector. For example the (4,2) element in the `i_mesh` matrix is 8.5750 minus 7.5250 which is 1.0500. Thus it would require 1.05 of investment to move from the second position in the `k_grid` vector to the fourth position in that vector.

Discussions of moving from one position in a grid vector to another are unusual in computational economics but are basic to discrete state space models. The reason harks back to our discussion above about

the airplane flying across the Atlantic from New York to London. When the pilot is at each point in the mesh of points over the Atlantic he is asking himself what point he should move to in his next step - one that is due east or another that is northeast or another that is southeast. Just so in a growth model that is solved with discrete state space methods. At time the economy has an amount of capital that corresponds to the second point in the capital grid and the question is asked about whether it is best to simply stay at that position the next time period or to invest enough to advance to the third, or the fourth, or the fifth grid position. With the investment mesh now in hand the next step in the program is to define an equivalent consumption mesh using equation (4) above and solving it for to obtain

$$C_t = Y_t - I_t \tag{21}$$

This is done with the statement

```
c_mesh = y_mesh - i_mesh
```

which yields the matrix

```
    0.5702    1.1089    1.6471    2.1847    2.7217
    0.0452    0.5839    1.1221    1.6597    2.1967
   -0.4798    0.0589    0.5971    1.1347    1.6717
   -1.0048   -0.4661    0.0721    0.6097    1.1467
   -1.5298   -0.9911   -0.4529    0.0847    0.6217
```

This matrix indicates the amount of consumption that would occur if the economy moves from grid position j to grid position i. For example the (3,3) position indicates that consumption would be 0.5971 if the economy was at capital grid position 3 in a given period and the decision is made to make no investment and simply remain at that grid position in the next time period. Of course the negative consumption levels are not feasible so the following three lines of code are used to set the negative elements of the matrix to zero

```
cc = find (c_mesh < 0)
c_mesh(cc) = 0
clear cc
```

The **MATLAB** command *find* locates the elements in `c_mesh` that are less than zero and puts those (i,j) locations in the array `cc`. Those values are set to zero in the next line and the `cc` array is cleared in the third line. The use of the clear statement at this point in the program is important to avoid errors later in the program.

Now that the consumption level that corresponds to each move among the grid points has been calculated, next we can compute the amount of utility that would be associated with each of these moves. This is done by using equation (7) above, i.e.

$$U(C_t) = \frac{1}{1-\tau} C_t^{1-\tau} \tag{22}$$

with the statements

```
u_mesh = (1/(1-tau)) * (c_mesh.^(1-tau))
```

This statement yields the `u_mesh` matrix

```
    1.5102    2.1061    2.5668    2.9561    3.2995
    0.4251    1.5283    2.1186    2.5766    2.9643
         0    0.4856    1.5454    2.1304    2.5859
         0         0    0.5370    1.5616    2.1417
         0         0         0    0.5819    1.5770
```

However the zeroes in this matrix can cause computational problems so a couple of statements are used to replace the zeroes with large negative elements. These statements, which function as well to ensure that , are

```
        uu = find (u_mesh == 0)
%       u_mesh(uu) = - realmax
        u_mesh(uu) = -10
```

The second of the three statements above is commented out with the symbol %. It is suggested that the reader use the third statement instead of the second when first learning how the program operates and then remove the comment from the second statement and move it to the third. The value realmax is the largest value that the computer code can generate. The problem with using it while first learning the program is that it is so large that all other numbers are so small that they are not printed with the usual print statements. Thus it is difficult to understand what is happening in the program. In contrast the use of the third statement above yields the `u_mesh` matrix shown below.

```
    1.5102    2.1061    2.5668    2.9561    3.2995
    0.4251    1.5283    2.1186    2.5766    2.9643
  -10.0000    0.4856    1.5454    2.1304    2.5859
  -10.0000  -10.0000    0.5370    1.5616    2.1417
  -10.0000  -10.0000  -10.0000    0.5819    1.5770
```

This completes the preliminary work in the program and opens the door to the use of the value function as in equation (1.20) above.

4.2 The Backward Loop

The next step in the program is to make use of the value function from equation (19) and to work backward as in the solution for the Rubik's Cube discussed above. First we specialize that equation to the next to the last period to obtain

$$V_{N-1}(k_{N-1}) = \max_{c_{N-1}} \{u(c_{N-1}) + \beta V_N(k_N)\} \tag{23}$$

In order to do this we must define which can be done substituting equation (20) into equation (21) to obtain

$$C_t = Y_t - (K_{t+1} - K_t) \tag{24}$$

and representing this in **MATLAB** with

```
clast = y_grid' - (aux * kN - k_grid')
```

Notice here that clast is a column vector so we transpose the output grid row vector `y_grid`. Also, kN is a scalar and that is multiplied by the column vector of n ones in aux. Then the row vector `k_grid` is transposed to a column vector and subtracted from the product of aux and kN. The result is the column vector clast, i.e.

```
-1.5298
-0.9911
-0.4529
 0.0847
 0.6217
```

The values in this vector are those which are the consumption levels that correspond to terminal capital stock levels of the grid points in `k_grid`, i.e.

```
    7.000
    7.525
    8.050
    8.575
    9.100
```

The negative values in this vector are then set to zero to yield the clast vector

```
     0
     0
     0
0.0847
0.6217
```

Then the utility in the last period can be computed by using the utility function from equation (1.23) specialized to the last period as

$$U(C_{N-1}) = \frac{1}{1-\tau} C_{N-1}^{1-\tau} \tag{25}$$

and written in **MATLAB** with the statement

```
Vlast  = (1 / (1 - tau) )* (clast .^ (1 - tau))
```

This yields the Vlast column vector

```
     0
     0
     0
0.5819
1.5770
```

As we have done above, it is also useful here to set the zero elements to large negative numbers and this is done with statements

```
    vv = find(Vlast == 0)
%   Vlast(vv) = - realmax
    Vlast(vv) = -100
```

Here, as above the statement with realmax is commented out and replaced with a statement using -100 which is a number large enough to insure that the resulting calculations are correct and yet small enough that one can easily see the results in order to understand how the program operates. After an initial learning phase it would be advisable to remove the % comment marker from the statement with realmax and comment out the line below. After the initialization of the backward loop the next step is the loop itself. The first and last statements in this loop are

```
for i =  T - 1: -1 : 1
    % matlab code
end
```

Notice that the loop begins with period T - 1 and decrements by one each pass until is reaches the first period. The first statement in the loop computes the term

$$u(c_{N-1}) + \beta V(k_N) \tag{26}$$

from equation (23) using the statement

```
V = u_mesh + beta * (Vlast * (aux'))
```

Since both Vlast and aux are column vectors the expression `Vlast(aux')` is a column vector times a row vector which yields a matrix containing Vlast in each of its n columns. This is then multiplied by the discount factor beta and added to the utility mesh matrix, `u_mesh` to obtain the term over which the maximization is done in equation (1.24). Thus V is the matrix

```
  -96.4898  -95.8939  -95.4332  -95.0439  -94.7005
  -97.5749  -96.4717  -95.8814  -95.4234  -95.0357
 -108.0000  -97.5144  -96.4546  -95.8696  -95.4141
   -9.4297   -9.4297    1.1073    2.1319    2.7120
   -8.4545   -8.4545   -8.4545    2.1274    3.1225
```

The maximization in equation (23) is then done with the **MATLAB** statement

```
[VV, II] = max (V, [], 1)
```

The **MATLAB** `max` function is used here with three arguments. The general form of this function is `max (V, [], dim)` and it returns the largest elements along the dimension specified by the scalar dim. Thus in the use above the function returns the largest element along the first dimension (the rows) of the matrix by going down the rows of each column and returning the maximal element. With two elements on the left hand side of the function it returns not only the largest element in each column in the row vector `VV` but also the index of each of those numbers in the row vector `I`. Thus you can see by examination that the maximum value in each column of the matrix `V` above is in the row vector `VV` as

```
  -8.4545   -8.4545    1.1073    2.1319    3.1225
```

Also, since `II` is a row vector of the indices of the elements which have the largest value in each column it is

```
   5     5     4     4     5
```

The information in the row vector `II` above is crucial to the dynamic programming approach of using a discrete state space. It is the first step to providing the map for the pilots flying across the Atlantic or, as in our case, the economist solving a growth model. There is one element in this row vector for each point in the grid for the state space, `k_grid`

```
7.000  7.525  8.050  8.575  9.100
```

Since we are at the first step in the backward loop this result tells the economist the optimal moves to make in the next to last time period (T-1). Thus the vector `II` tells the economist that if in the next to last time period the solution should happen to be at capital accumulation that corresponds to the second column in `II` and thus to the second grid point in `k_grid`, i.e. to 7.525, then in the next step the optimal solution will be to move to the 5th grid point, i.e. to 9.100. In contrast, if the solution in this next to last time period corresponds to the 3rd column in `II`, i.e. is at a capital stock of 8.050 then the optimal solution for the last step would be to move to the 4th position in `k_grid`, namely to 8.575.

The next step in the program then is to store away the information in the vector `II` in a matrix `I` that has a row for each time period and a column for each grid point in `k_grid`. This is done with the statement

```
I (i, :) = II
```

This yields the matrix I

```
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
5     5     4     4     5
```

In this first pass through the backward loop the index `i` is equal to `T-1`, or in our case 5 so this statement has the effect of putting the row vector `II` into the fifth row of the matrix `I`. Thus the elements in the matrix I, i.e. I(i,j) provide the map for the pilots (economists) that we discussed above. The economist uses this matrix as a *look-up* table. If the economy in period i happens to be at grid point j in `k_grid` then in the next period the optimal step will be to move to grid point `I(i,j)` in `k_grid`. For example, if the economy in period 5 is at grid point 3 in `k_grid` then the optimal next step is to move to grid point 4. But enough of this for now! We will return to this subject again shortly after more of the matrix I is constructed.

Meanwhile, before we advance to the next time period in the backward loop we need to update the value Vlast with the statement

```
Vlast = VV'
```

Recall from above that `Vlast` before we started through the backward loop was the column vector

```
     0
     0
     0
0.5819
1.5770
```

and contains the value of the value function for the last time period. Now with the statement above we update it with the row vector VV transposed, thus Vlast now becomes

```
-8.4545
-8.4545
 1.1073
 2.1319
 3.1225
```

This column of numbers tells the economist the value of being at each of the grid points in k_grid in the next to last time period, values that will be used in the second pass through the backward loop. For this second pass through the backward loop consider all at once the six statements in this loop, i.e.

```
for i =  T - 1: -1 : 1
        i
        V = u_mesh + beta * (Vlast * (aux'))
        [VV, II] = max (V, [], 1)
        I (i, :) = II
        Vlast = VV'
end
```

In the second pass through this loop the index will have the value $T - 2$, or in our case 4. The first statement in the loop, namely

```
        i
```

serves no purpose other than to cause the time period to be printed which helps in deciphering the output of the program when one is first learning it. The rest of the lines are familiar from the discussion above. The effect of this second pass through the backward loop is to fill in the 4th row in the *look-up* matrix I so that it becomes

```
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
1     3     3     4     5
5     5     4     4     5
```

So if the economy in period 4 happens to be a grid point 2 it should be moved to grid point 3 in `k_grid`. Then in period 5 it will be a grid point 3 and it should move in that period to grid point 4. This would correspond to a capital path in periods 4 and 5 of

```
8.050  8.575
```

and since in this model the capital stock has to be at the terminal target of 9.1 in the last, or 6th period, the time path for capital in this case for the last three time periods would be

```
8.050  8.575  9.100
```

Similarly, after the completion of all five passes through the backward loop the I matrix will be fully filled out and will be

```
1     2     3     4     5
1     2     3     4     5
2     2     3     4     5
1     3     3     4     5
5     5     4     4     5
```

This matrix provides the full map for the economist and gives the optimal move in each time period if the economy happens to be in each of the grid points of `k_grid` at the beginning of that time period. This completes the discussion of the backward loop. Next we develop a forward loop using the I matrix above to construct the optimal path for capital accumulation.

4.3 The Forward Loop

The I matrix above can be used to compute the optimal path for the growth model for any initial condition in `k_grid`. However we need only one of those points, namely `k0` so the initialization of the forward loop is done with the statement

```
start = find (k_grid == k0)
```

So start becomes the integer corresponding to the position of `k0` in `k_grid`. In our case `k0` is 7 and `k_grid` is

```
 7.000  7.525  8.050  8.575  9.100
```

so start is set to one. his is followed by the forward loop which is

```
for i = 1:T-1
      i
      next(i) = I(i,start)
      k_path = k_grid(next(i))
      start = next(i)
end
```

In this case the loop goes forward from 1 to 5 and the first statement in the loop, namely i, is used for no purpose other than to print the time period in order to make the output easier to read when the reader is first using the program. The following statement, namely

```
next(i) = I(i, start)
```

tells the economist where to go in the ith time period in order to follow the optimal path, From the *I lookup* matrix above one can see that the `I(1,1)` element is one so `next(1)` becomes 1. The following line, i.e.

```
k_path(i) = k_grid(next(i))
```

then sets the first element in the vector `k_path` to the first element in `k_grid`, namely to 7.000. Then the starting point for the next pass through the forward loop is set with the statement

```
start = next(i)
```

So for the second pass through the forward loop start will be 1 again. After the second pass through the forward loop next becomes the row vector

```
1     1
```

after the third pass (third time period) next becomes

```
1    1    2
```

On can see this by examining the `I` matrix again

```
 1     2     3     4     5
 1     2     3     4     5
 2     2     3     4     5
 1     3     3     4     5
 5     5     4     4     5
```

The third row in the `I` matrix corresponds to the third time period. The start for that time period (the capital stock from the previous time period) is in the first position in `k_grid` so we add to next in the this period the $(3, 1)$ element of the `I` matrix which is 2.

After the fourth pass the next row vector is

```
1    1    2    3
```

and this occurs because the $(4, 2)$ element of the `I` matrix is 3, i.e. in period 4 when one starts the period from the 2nd position in the capital grid one should move to the 3rd position in `k_grid`. The corresponding `k_path` row vector is

```
7.0000    7.0000    7.5250    8.0500
```

After the completion of the 5th and last pass through the forward loop the row vector next is

```
1    1    2    3    4
```

and the corresponding `k_path` row vector is

```
7.0000    7.0000    7.5250    8.0500    8.5750
```

This is the optimal path for capital accumulation for periods 1 through 5. Thus it remains only to add the capital stock for the first and last time periods to `k_path` and this is done with the statement

```
k_path = [k0  k_path  kN]
```

to yield the final `k_path` which is

```
7.0000 7.0000 7.0000 7.5250 8.0500 8.5750 9.1000
```

The program is then completed with the following three statements which are used to plot the optimal path for the capital stock, i.e.

```
figure(1)
plot(k_path)
title('Path of capital')
```

which yields the graph shown in Figure 1.

We have purposely used a grid with only a few points in the discussion in this section in order to minimize the amount of output and thus make the program easier to understand. However, the accuracy of the result can be increased substantially by using a finer grid for `k_grid` with more grid points. This is done in the following section.

Figure 1: The optimal path for the capital stock.

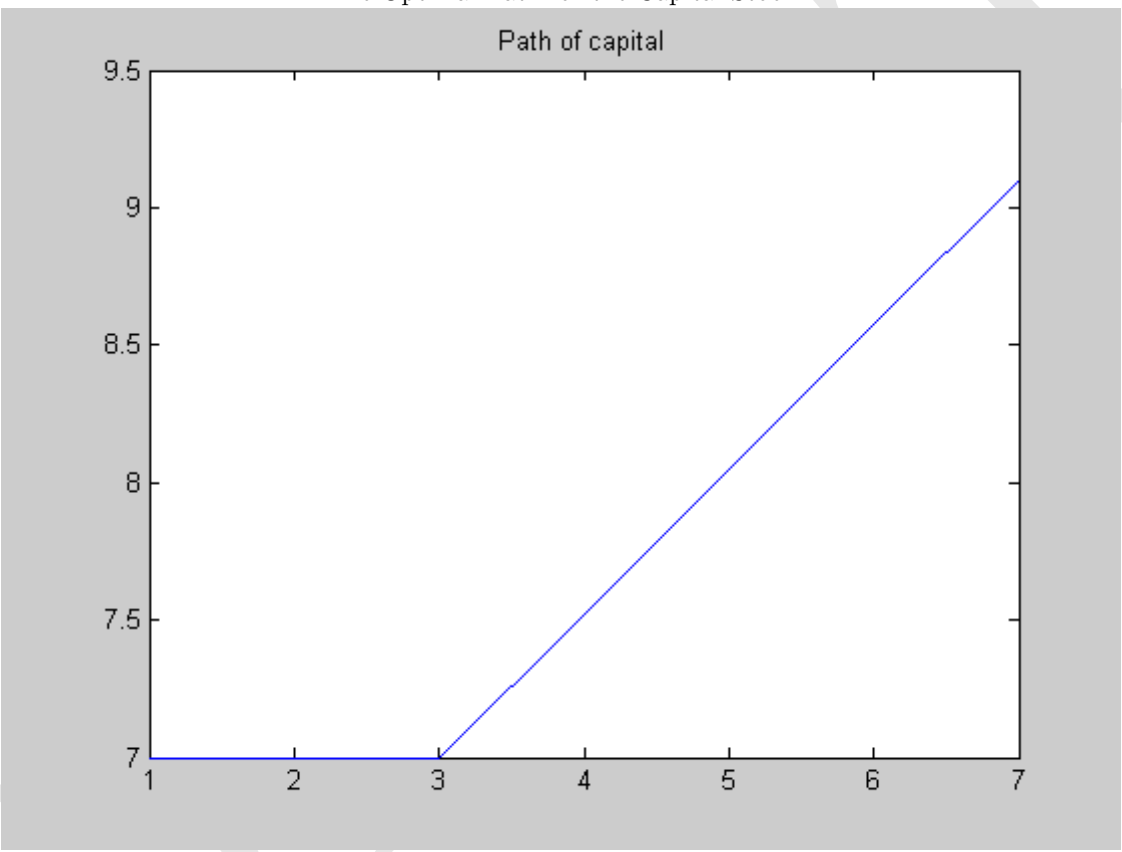
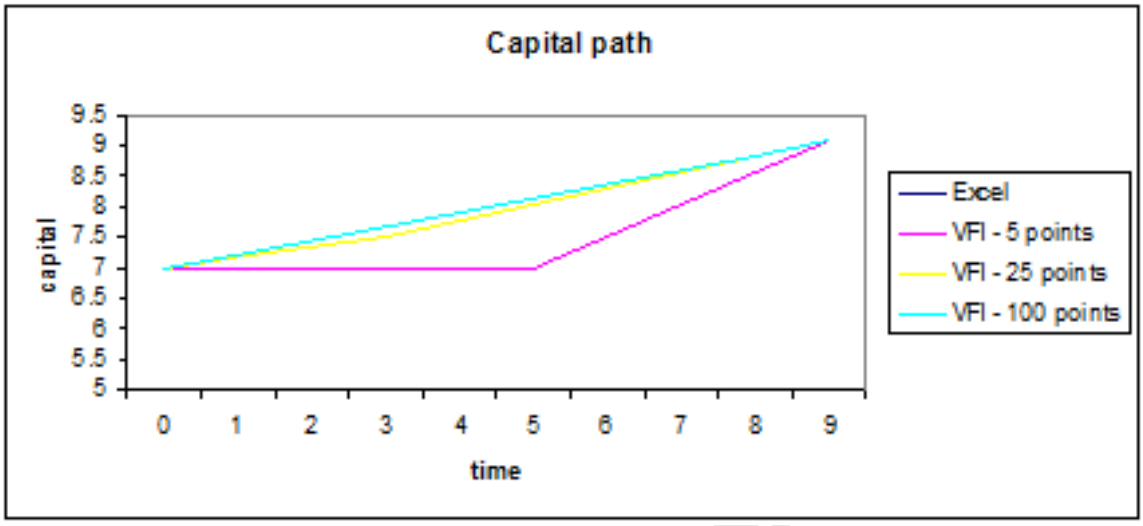


Figure 2: The Capital Path with 5, 25 and 100 Grid Points and the Excel Resul

The Capital Path with 5, 25 and 100 Grid Points and the Excel Result



5 Results with a Finer Grid

We have run the program above with grids of 5, 25 and 100 points and then plotted the results along with the result we obtained in Chapter 1 where this growth model was computed using nonlinear programming and a continuous state variable by employing the Solver in Excel. The results are shown below in Figure 2.

In Figure 2 one can see that a relatively small number of grid points are needed in order to closely approximate the solution obtained with the Solver in Excel. Thus in this case the discrete state space approach yields almost the same result as the nonlinear programming approach with a continuous state variable.

References

Adda, J. and Cooper, R.: 2003, *Dynamics Economics: Quantitative Methods and Applications*, MIT Press, Cambridge, Massachusetts.

Bertsekas, D. P.: 2005, *Dynamic programming and optimal control*, Vol. 1, 3rd edn, Athena Scientific, Boston, USA.

Stockey, N. L. and Lucas, R. E.: 1989, *Recursive methods in economic dynamics*, Harvard University Press, Cambridge, Massachussets.

Appendix

A Solving with DP in Discrete State Space

```
%Value function iteration for an infinite period and uncertainty
%model is based on the Chakravarty model as in Kendrick, Mercado and
%Amman (2006)
%Code written by Firat Yaman and modified by David Kendrick
%See the mathematics in Adda and Cooper(2003) and in
%Stokey and Lucas with Prescott (1989)

%clears command window and variables
clc;clear all;

%alpha and theta are production coefficients, beta discount factor,
%k_grid is a grid for capital, tau is a utility
%parameter, delta is depreciation, k0 and kN define first and last
%point on the grid

alpha = 0.33;
beta = 0.98;
theta = 0.3;
tau = 0.5;
delta=0.1;
k0 = .1;
kN = 2.1;
n = 20;
step = (kN-k0) / (n-1);
k_grid = [k0:step:kN];

iterlim = 5;
tolera = 0.000001;

z_low = 0.90;
z_high = 1.10;

%probability of going from low to low, low to high
P(1,1)=0.5;P(1,2)=1-P(1,1);
%probability of going from high to low, high to high
P(2,1)=0.1;P(2,2)=1-P(2,1);

%Output - y-grid is the grid for output
y_grid_l = z_low*theta*(k_grid.^alpha);
y_grid_h = z_high*theta*(k_grid.^alpha);

%vector of ones
aux = ones(n,1);

%current capital stocks vary along a row in kt_mesh
%future capital stocks vary along a column in kt1_mesh
%i_mesh is the corresponding mesh for investment
%since i = k(t+1) - (1-delta) * k(t)

kt_mesh = aux*k_grid;
kt1_mesh = (aux*k_grid)';
i_mesh = kt1_mesh - (1-delta) * kt_mesh;

y_mesh_l = aux*y_grid_l;
```

```
y_mesh_h = aux*y_grid_h;

%then consumption is c(t)= y(t)-i(t)
%to create the consumption mesh array
c_mesh_l = y_mesh_l - i_mesh;
c_mesh_h = y_mesh_h - i_mesh;

%replace negative consumption with zero
cc = find(c_mesh_l<0);
c_mesh_l(cc) = 0;
clear cc;
cc = find(c_mesh_h<0);
c_mesh_h(cc) = 0;
clear cc;

%utility; to avoid negative and zero consumption we asign the lowest
%possible utility to entries with zero consumption
u_mesh_l = (1/(1-tau))*(c_mesh_l.^(1-tau));
uu = find(u_mesh_l == 0);
%u_mesh(uu) = -realmax
u_mesh_l(uu) = -10;
u_mesh_h = (1/(1-tau))*(c_mesh_h.^(1-tau));
uu = find(u_mesh_h == 0);
%u_mesh(uu) = -realmax
u_mesh_h(uu) = -10;

%first guess for the value function; a natural starting point is one
%where there is no investment and consumption therefore equals
%output.
clast_l = y_grid_l';
Vlast_l = (1/(1-tau))*(clast_l.^(1-tau));
clear cc vv uu;
clast_h = y_grid_h';
Vlast_h = (1/(1-tau))*(clast_h.^(1-tau));
clear cc vv uu;

%iteration, decision rules for all capital states; from the
%utility matrix expanded by the next-period value, this picks for every
%column the highest entry (that is, for every state of capital the best
%future state of capital); tol is the convergence criterion
tol = 1
iter = 0
while ( (tol > tolera) & (iter < iterlim) )
    iter = iter + 1;
    V_l=u_mesh_l + beta*(P(1,1)*Vlast_l*(aux')+P(1,2)*Vlast_h*(aux'));
    V_h=u_mesh_h + beta*(P(2,1)*Vlast_l*(aux')+P(2,2)*Vlast_h*(aux'));
    [VV_l,I_l] = max(V_l,[],1);
    [VV_h,I_h] = max(V_h,[],1);
    tol_l=max(abs(VV_l-Vlast_l'));
    tol_h=max(abs(VV_h-Vlast_h'));
    tol=max(tol_l,tol_h);
    Vlast_l = VV_l';
    Vlast_h = VV_h';
end

iter
tol
Vlast_l
Vlast_h
I_l
I_h
```

```
%plotting the decision rule for the capital stock, the steady state is
%where the steady state line crosses the decision rule
index=1:n;
figure(1)
plot(index,index,'-',index,I_l,'--',index,I_h,':');
title('Path of capital');
legend('steady state','decision rule low','decision rule high');
```