Chapter 7

Dynamic Choice on an Infinite Horizon

> "Bigger than the biggest thing ever and then some. Much bigger than that in fact, really amazingly immense, a totally stunning size, real 'wow, that's big', time. Infinity is just so big that by comparison, bigness itself looks really titchy. Gigantic multiplied by colossal multiplied by staggeringly huge is the sort of concept we're trying to get across here."
>
> *Douglas Adams, "The Hitchkiker's Guide to the Galaxy"*

If you're following this book linearly, in the previous chapter you'll have gone through a number of issues involved in determining optimal choice across time periods, including an outline of the ideas behind Bellman's functional equation. In setting up all of these solutions, we've relied on having some known point from which to anchor things: typically the fact that no value remains if goods are left unconsumed after the final period of time. Once we have this final period to act as an anchor, we are able to iterate backwards, and in this way calculate the value function—and as a result optimal decisions—in all time periods.

But, you might ask, how would we move ahead were we not to have such a point to work from? What if we are solving a problem that doesn't have a terminal point at all? There are many times in microeconometric problems where this may be a relevant concern. For example, it seems unlikely that many firms would plan to close up shop at a defined point in the future. Indeed, even in the case of individuals deciding over their lifetime, the terminal point may be so many periods away[1] (hopefully!), so uncertain, and discounted in such a way that acting as if the horizon is infinite may be the most reasonable approach when searching for a solution.

Problems of this type—those that never end and hence have no terminal point as an anchor—will require another class of solution. Fortunately, such a situation can be well-served using the ideas motivated by Bellman (1957) which we discussed in chapter 6. In the sections ahead we will have a look at this class of solutions, and set up some implementations in MATLAB. Once again though, there are many applications, and we will only scratch the surface. We hope that this outline should provide you with a strong foothold in how to set up a wider range of these problems, but direct the interested reader to further resources such as Acemoglu (2008); Adda and Cooper (2003); Ljungqvist and Sargent (2000); Judd (1998); Dixit (1990) and the many papers cited therein.[2]

In this chapter we return to the problem outlined in Chapter 6 which represents a firm's decision over how much to invest now, and how much to save for the future. This problem has a long history in economics, being described by Ramsey as early as 1928 (although applied to countries rather than firms; some examples for firms include Bond and Söderbom (2005); Fafchamps et al. (2011)). The objective is to maximise total discounted utility, subject to the

[1] Of course, the idea of a 'period' is quite nebulous. For example, when defining these problems, should we consider a period to be a year? Or a month? Or a day? Generally we will allow the context of a problem to dictate the length of a period. For example in optimal child-bearing decisions a woman can only become pregnant at most once every 9 months, in job search a searcher can receive job offers which much greater frequency (depending upon search intensity), and so forth.

[2] Much work on dynamic programming with an infinite horizon has been done in macroeconomics, for example considering country growth rates and investment decisions. There is a large literature in this area, including resources on the application of these problems in computer languages (see for example the excellent references of Stachurski (2009) and Sargent and Stachurski (2013) in Python, or Collard's online lecture notes on value function iteration in MATLAB with a macroeconomic focus.)

flow equation for capital:

$$\max_{\{c_t\}_{t=1}^{\infty},\{k_t\}_{t=2}^{\infty}} \sum_{t=1}^{\infty} \beta^{t-1} \ln(c_t) \qquad \text{subject to} \qquad \begin{aligned} k_{t+1} &= \theta k_t^{\alpha} - c_t \\ c_t &\geq 0 \\ k_t &\geq 0. \end{aligned} \tag{7.1}$$

You will note here that we are assuming quite a particular functional form: that of log utility and Cobb-Douglas production. There is a reason that we make these assumptions. Such a functional form allows for us to find both an analytical and a numerical solution. Typically finding an analytical solution to dynamic problems of this type is not possible or very difficult—meaning that we *have* to revert to numerical tools like MATLAB. However, in this case having both solutions is quite handy. We can see how we would resolve this in MATLAB, while also having the exact solution against which we can compare our results.

7.1 Value Functions and Infinite Solutions

7.1.1 A Rough Outline

In the previous chapter we introduced the notion of Bellman's functional equation to solve dynamic optimisation problems (see for example equation 6.8). The basic idea is that we break down an optimisation problem which runs over many periods into two periods: now (ie the current period), and the future, where the value of the future is represented by a 'functional equation' which we have been labelling V. This looks something like:

$$V(k) = \max_{c,\tilde{k}} \left\{ u(c) + \beta V(\tilde{k}) \right\} \tag{7.2}$$

where here rather than rely on time subscripts we are using a tilde over a variable (such as $\tilde{k}$) to signify future values, and no tilde (ie k) to denote current values.[3] Substituting the functional forms we assumed in (7.1), into our Bellman

[3] This notation is not without reason. We use it given that time does not actually enter into the Bellman equation at all. In other words, the Bellman equation is **stationary**.

equation implies that (7.2) will end up looking like:

$$V(k) = \max_{c,\tilde{k}} \left\{ \ln(\theta k^{\alpha} - \tilde{k}) + \beta V(\tilde{k}) \right\}. \tag{7.3}$$

So far this looks much like what we have done so far in finite horizon problems. However as we suggested in the introduction to this chapter, in *in*finite horizon problems, we have (by definition) no final period, so backwards recursion from V_T is no longer a viable option. Fortunately we have a way forward using the tools we've already been looking at. Much work exists to show that if we choose an arbitrary value function—say for example $V(k) = 0 \times k = 0$, and iterate on this value function, we will eventually converge on the true value function.

If you will permit us to be somewhat verbose on this point, this implies that we choose some arbitrary initial function, in this case a vector of zeros, and plug this into the right hand side of (7.3) as $V(\tilde{k})$. Resolving (7.3) will give us a new value function (ie the left-hand side of (7.3)). If we then plug this new value function back into the right-hand side of the equation, we will once again get a new value function, which we yet again substitute into (7.3) *ad infinitum* – or rather, until the value function on the left-hand side is equal to the value function on the right-hand side. At this point our convergence is complete, and we know that we have the true value function. Essentially then, we are searching for a fixed point which implies that we have determined a function $V(\cdot)$ which summarises the value of behaving optimally forever more. You may note that in this case *the same* $V(\cdot)$ enters both sides of the functional equation (7.3) implying stationarity, or that optimal consumption just depends upon capital, and *not* upon the time period in which the firm finds itself.

In more compact form, we are interested in calculating:

$$\Gamma V(k) = \max_{\tilde{k}} \left\{ \ln(\theta k^{\alpha} - \tilde{k}) + \beta V(\tilde{k}) \right\} \qquad \forall\ k, \tag{7.4}$$

where Γ is an operator representing this process of iteration on the value function until the point that $\Gamma V(k) = V(k)$. We do not think that this book is the place to provide a great deal of detail on the maths (and certainly not the mathematical proofs!) behind this idea. However, we *do* provide an appendix to this chapter where we show analytically how value function iteration works. If

you are after more precise mathematical details about the process of value function we would refer you to one of the previously cited texts such as Ljungqvist and Sargent (2000), or alternatively, to Bertsekas (1976, 2005), as from here on we choose to focus entirely on the *computational* aspects of the problem. In order to do so, we would ask that you convince yourselves (or perhaps you are already familiar with the maths behind dynamic optimisation) that this particular Bellman equation (7.3) can be satisfied by a value function of the form:

$$V(k) = \frac{\alpha}{1 - \beta\alpha} \ln k + F \tag{7.5}$$

where F just represents a constant, and that this value function implies the following policy function:

$$c(k) = \theta k^{\alpha}(1 - \beta\alpha). \tag{7.6}$$

If you are not entirely convinced, or if you simply wish to brush up on your math here, we direct you to the aforementioned appendix to this chapter where we show that this is the case.

7.1.2 Computation

The Value Function

In moving to computation then, the question becomes how we can actually instrumentalise the iteration process $\Gamma V(k)$. From (7.4) a number of things perhaps stand out. We will likely have to calculate $\theta k^{\alpha} - \tilde{k}$, we will likely have to try this over a range (or grid) of different values of $\tilde{k}$, we will want to choose the 'best' outcome for $\tilde{k}$ in the sense that it maximises $\Gamma V(k)$, and finally, we will want to do this for 'all' values of k. Let's have a look at some code...

```
function [TV optK] = iterateVF(V,maxK)
  % iterateVF(V,maxK) takes a potential value function V and
  % performs an iteration, returning the updated proposed value
  % function TV.  When TV==V, we have found the true value
  % function.  The scalar maxK represents the maximum possible
  % amount of capital that can be consumed in one period
```

```
  %=============================================================
  %=== (1) Basic Parameters
  %=============================================================
  alpha   =  0.65; beta  =  0.9; theta  =  1.2;

  grid    =  length(V);
  K       =  linspace(1e-6,maxK,grid)';
  TV      =  zeros(length(V),1);

  %=============================================================
  %=== (2) Loop through and create new value function for each
  %===     possible capital value
  %=============================================================
  for  k = 1:grid
    c               = theta*K(k)^alpha-K(1:k);
    c(c<=0)         = 0;
    u               = log(c);
    [TV(k) optK(k)] = max(u + beta*V(1:k));
  end
return
```

This code provides one iteration of the value function, after being passed a proposed value function `V`, and given an upper bound for capital (`maxK`). The first section simply inputs our necessary parameters so we won't discuss this. The second section is the important part of this function. Firstly we loop over all possible values of k. This ensures that when we find our final solution, it will hold for all k. In this loop we calculate utility based on all possible values for $\tilde{k}$ – in the first line of the loop we define a vector of possible values for $\tilde{k}$: from lowest possible `K` value, right up to the entire amount `k`. From there we maximise the current iteration of the value function: here define a vector which gives us two outputs: `TV(k)` which is the maximised value function, and `optK(k)` which is the capital value associated with this maximum (note that if this is unclear to you, it may be worth reading the help file for `max`).

Now that we have defined a function `iterateVF` which allows us to make one

iteration of a value function, we will want to implement this by passing this a proposed value function, and receiving an updated value function. As per the appendix of this chapter, we will subscript proposed value functions with $j \in [0, \infty)$ to signal the iteration number. In this case V_0 will be our starting point (which we will arbitrarily define as $V_0(k) = 0$), and the result for the first iteration will be labelled as $V_1(k)$. In the following code we run through 10 value function iterations, in which case the final result will be $V_{10}(k)$.

```
%=====================================================================
%=== (1) Set parameters, plot analytical solution
%=====================================================================
Beta = 0.9; alpha = 0.65; theta = 1.2; aB = alpha*Beta;
K = linspace(1e-6,100,100);

E = alpha/(1-aB);
F = 1/(1-Beta)*(log(theta*(1-aB))+aB*log(aB*theta));
soln = E*log(K)+F;

plot(K,soln, '-k', 'LineWidth', 3)
axis([0 100 -15 10])
hold on
%=====================================================================
%=== (2) Plot 10 value function iterations
%=====================================================================
TV = [zeros(100,1) NaN(100,9)];
for iter = 1:10
    fprintf('Iteration number %d\n', iter)
    TV(:,iter+1)=iterateVF(TV(:,iter),100);
end

plot(K,TV)
xlabel('Amount of Capital', 'FontSize', 12)
ylabel('Value Function', 'FontSize', 12)
title('Value Function Iteration', 'FontSize', 14)
```

The above script stores our 10 value function iterations in the matrix `TV`, along

with the initial value function which is just a vector of zeros. Along with these value function iterations which we calculate in MATLAB, we also define the analytical solution which we can use as a comparison to our numerical solutions. The resulting output is presented in figure 7.1. You'll probably notice a few things in this figure. Firstly, the initial value functions (the thin coloured lines) seem to converge reasonably rapidly towards the true value function (the thick black line). However, there's clearly more work to do. Our 10^{th} iteration, $V_{10}(k)$, (the light blue line), is not *that* close to the target result.

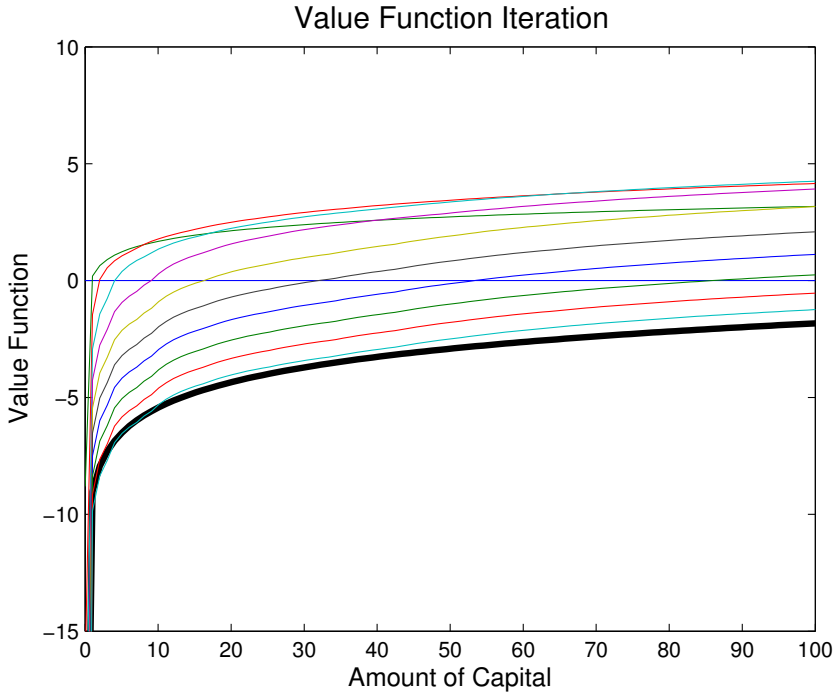

Figure 7.1: Convergence after 10 Iterations

In this case we can write another script in which we explicitly require the value function to converge before stopping. This time we won't use a loop which defines a certain number of iterations (as we did above in the code with `for iter=1:10`), but rather we will ask MATLAB to itearate on our code until a certain condition is met. The idea here is that our value function iteration will

have 'converged'[4] when $V_{j+1}(k) \approx V_j(k)\ \forall\ k$. In order to operationalise this in MATLAB we use a `while` loop. As long as the following convergence criterion is not met, we ask MATLAB to keep iterating on the value function.

$$||V_{j+1}(k) - V_j(k)|| \leq \varepsilon\ \forall\ k \tag{7.7}$$

In the code which follows our ε is labelled `crit` (which we define as 0.01), and at the end of each iteration we calculate $||V_{j+1}(k) - V_j(k)||$, which we call `conv`.

```
%=================================================================
%=== Convergence to the Value Function
%=================================================================
conv =  100;
crit =  1e-2;

K    =  linspace(1e-6,100,1000);
V    =  zeros(1000,1);
axis([0 100 -15 10])
hold on

cc   =  hot(70);
Iter =  0;

while conv>crit
   Iter      =  Iter+1
   [TV opt]  =  iterateVF(V,100);
   conv      =  max(abs(TV-V))
   plot(K,TV, 'color', cc(Iter,:))
   V         =  TV;
end
```

The output from this code is presented in figure 7.2, and here we see that our value function does indeed converge. In this graph we've used MATLAB's inbuilt

[4] We write converged in inverted commas here to imply that it is a slight abuse of nomenclature. In numerical iterations we will never have true convergence of the value function. Rather, contiguous value functions will move closer and closer to one another as we iterate towards infinity until the distance between them is extremely small.

"colormap" `hot`, which results in higher iterations on the value function being 'hotter' colours. As with all the code outputs in this book, we suggest you look up any functions that you aren't familiar with in MATLAB's help files or on the web, and encourage you to play around and see how the results react to different input parameters. What happens to the following code if you start with an alternative $V_0(k)$ for example? How about if you make a more finely spaced capital grid?

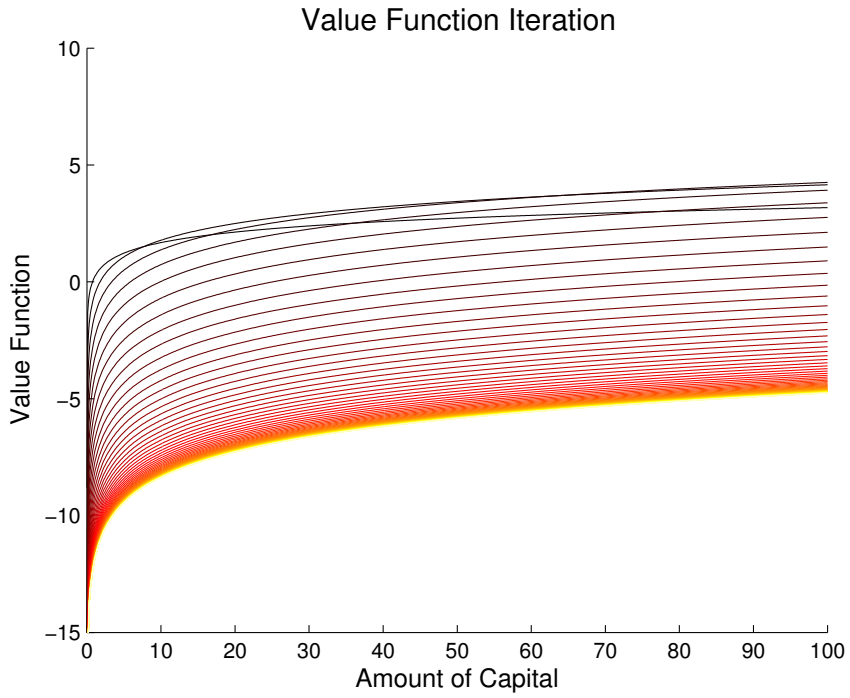

Figure 7.2: Convergence to the True Value Function

The Policy Function

After all of this work to find the value function in problems with an infinite horizon, you may be wondering how we actually determine what our firm's optimal behaviour is. More specifically, how much should the firm consume at a given point in time, and how much should it save? Fortunately, finding this

policy function—that is $c(k)$, or the mapping from capital to consumption—is reasonably straightforward.

You may remember that in the function `iterateVF`, we solved for both the value function at each point on our capital grid, along with the corresponding optimal amount of capital consumption at this point. If you want to refresh your memory, have a look at the final line of the loop in section 2 of this code. You'll also notice that we return this optimal capital vector when we define the `Iterate_VF` (to review functions and returning vectors, see chapter 1). And finally, when we run our optimal convergence code, for each iteration we save two vectors: `TV`, the value function, and `opt`, the optimal amount of capital consumption.

In this case, generating the policy function is just a question of applying these optimal capital values to our capital grid `K`. In the five lines of code which follow we show how we generate figure 7.3b. This shows both our numerically calculated policy function (the solid blue plot), along with the exact analytical result we derived in appendix 7.A.

```
>> aB  = 0.65*0.9;  theta = 1.2;  alpha = 0.65;
>> plot(K,K(opt),K,aB*theta*K.^alpha, '--r', 'LineWidth', 3)
>> xlabel('Amount of Capital', 'FontSize', 12)
>> ylabel('Optimal k_{t+1}', 'FontSize', 12)
>> title('Policy Function for Capital Consumption', 'FontSize', 14)
```

Exercise: Create similar graphs to these optimal k_{t+1} graphs displayed as 7.3a and 7.3b (with both analytical and numerical predictions) for optimal consumption at t based upon the amount of capital in hand at t.

7.2 Policy Iterations and Faster Solutions

In the previous section we saw that iterating on the value function in MATLAB is quite an effective way to find the numerical solution to these infinite horizon

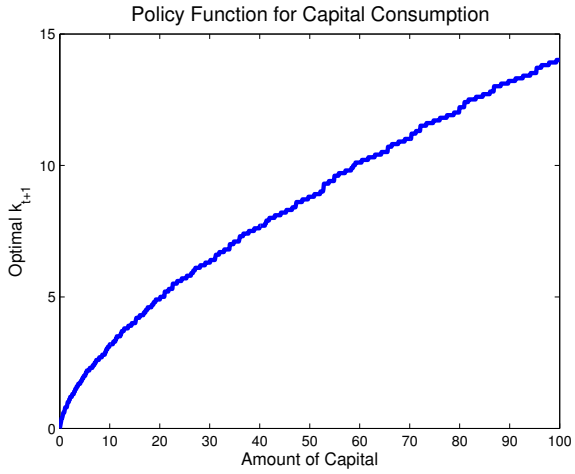

Figure 7.3a: Calculated Best Path

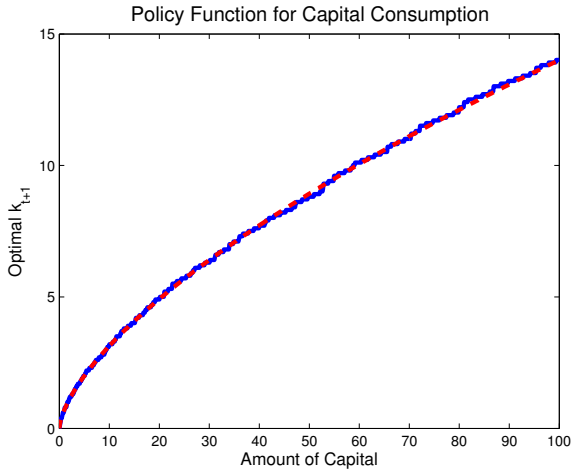

Figure 7.3b: True Best Path

problems. However, this value function iteration is not necessarily computationally cheap. In some cases this may not concern you. If the 'traditional' value function iteration works for your particular problem you may be happy to learn and apply this, rather than focusing on more efficient refinements. However, in other cases you may find that it is worth your while to spend some time learning how to further optimise your code. If, for example, you expect to work with a very large state space, you may find that each additional iteration on the value function takes a long time to complete. If this is the case, the remainder of this section is for you...

As we saw with the code `ConvergeGraph.m`, we required 66 iterations before V_{j+1} was close enough to V_j for us to consider that the function had 'converged'. What's more, in this code, we define 'convergence' as a situation where $||V_{j+1} - V_j|| < 0.01$. Were we to set a more rigorous convergence criterion (which we control in the second line of `ConvergeGraph.m`), we will require (perhaps many) more iterations to solve the problem.[5]

One particularly useful alternative manner to solve this problem is the so-called **Howard Improvement Algorithm** or policy function iteration (Sargent, 1987). Rather than iterate to calculate the value function, and then from

[5] For example, setting a convergence criterion of 1e-6 means that our problem now takes 153 iterations to converge.

this the policy function (figure 7.3b), the Howard Improvement Algorithm alows us to directly solve for the policy function. Fortunately, as we will see, this algorithm typically converges to the true policy function in much fewer steps than the value function iteration in section 7.1.2.

In broad terms, the policy function iteration looks like the following, where the process is initialized by setting some initial arbitrary value function $V_j = V_0$, and defining some stopping criterion ε:

(i) Based upon V_j, determine optimal consumption for each k, giving a proposed 'policy function', $c_j(k)$

(ii) Calculate the payoff associated with this policy function, $u(c_j(k))$

(iii) Calculate the value of following this policy function forever, V_{j+1}

(iv) If $||V_{j+1} - V_j|| < \varepsilon$ stop, or else return to step (i) for another iteration

The increase in the efficiency of this routine comes from calculating a value function associated with following the policy function *forever* in step (iii). In the traditional value function iterations we've been calculating so far, the calculated policy function is only followed for one period before again iterating to calculate V_{j+1}. While policy function iteration generally takes many fewer steps than value function iteration, there is one computationally heavy step involved in calculating the value function from the policy function. To see this, consider solving for the unknown V_j in the following computation[6]:

$$\begin{aligned} V_j &= u(c_j(k)) + \beta Q_j V_j \\ \Rightarrow V_j &= (I - \beta Q_j)^{-1} u(c_j(k)), \end{aligned} \tag{7.8}$$

where I is the identity matrix, and Q a matrix which keeps track of the capital stock associated with a given V_j.[7] At each step of (7.8), which corresponds to item (iii) on the above list, V_j is calculated by matrix left division; this can be a computationally demanding process.

[6] Here we borrow the notation of Judd (1998), and direct you to his discussion on pp. 411–417 should you be interested in further details

[7] In the case of a stochastic infinite horizon model, Q acts as a transition matrix describing the probability of each realisation in the stochastic portion of the model. We do not go into that here, and refer you once again to resources such as Judd (1998) if you're interested in these sort of details.

Below we lay out how the above enumerated list would look in MATLAB code. We suspect that by now you will be familiar with many of these commands, so will spare you a step-by-step discussion, and will leave it to you to examine this at the command line. We will however point out the use of MATLAB's `sparse` routines, given that the matrix Q in (7.8) is made up largely of zeros.

```
function [V,opt] = Iterate_Policy(V, maxK)
   % Iterate_Policy(V, maxK) takes an aribitrary value function
   % V and iterates over the policy function c(k). At each step
   % it calculates an updated policy function c_j(k), and a
   % corresponding value function V_j(k), which is the value of.
   % following c_j(k) forever.
   %
   % see also iterateVF

   %=============================================================
   %=== (1) Parameters
   %=============================================================
   alpha  =  0.65; beta  =  0.9;  theta  =  1.2;
   grid   =  length(V);
   K      =  linspace(1e-6,maxK,grid)';
   opt    =  NaN(grid,1);

   %=============================================================
   %=== (2) Calculate optimal k for V
   %=============================================================
   for k  =  1:grid
       c               =  theta*K(k)^alpha-K(1:k);
       c(c<=0)         =  0;
       u               =  log(c);
       [V1,opt(k)]     =  max(u+beta*V(1:k));
   end

   kopt =  K(opt);
   c    =  theta*K.^alpha-kopt;
   u    =  log(c);
```

```
    %=============================================================
    %=== (3) Invert k, u to find V_{j+1}
    %=============================================================
    Q    =  sparse(grid,grid);

    for  k  =  1:grid
       Q(z,opt(z)) = 1;
    end
    TV  =  (speye(grid)-beta*Q)\u;
    V   =  TV;
return
```

Having now written an iteration for this policy function improvement step, we can ask MATLAB to loop over this a number of times until our numerical policy function has converged. Were we to do this at the command line, we would proceed as below. The first four lines set graphing parameters and graph the analytical solution, while the `for` loop iterates over the policy function 7 times, starting with the defined $V_0 = 0$. Figure 7.4 presents output, and it appears as if after only 7 iterations we are already very close to the true policy function!

```
>> cmap  = cool(7);
>> V     = zeros(1000,1);
>> K     = linspace(1e-6,100,1000);
>> aB    = 0.65*0.9; theta = 1.2; alpha = 0.65;
>> plot(K, aB*theta*K.^alpha, '-k','LineWidth',3)
>> hold all
>> for l = 1:7
[V,k]    = Iterate_Policy(V,100);
plot(K,K(k), 'color', cmap(l,:))
end
>> legend('Analytical', 'Iter 1', 'Iter 2', 'Iter 3', 'Iter 4', ...
'Iter 5', 'Iter 6', 'Iter 7', 'Location', 'NorthWest')
>> xlabel('Amount of Capital')
>> ylabel('Optimal c_t')
>> title('Policy Function Iteration and Optimal Consumption')
```

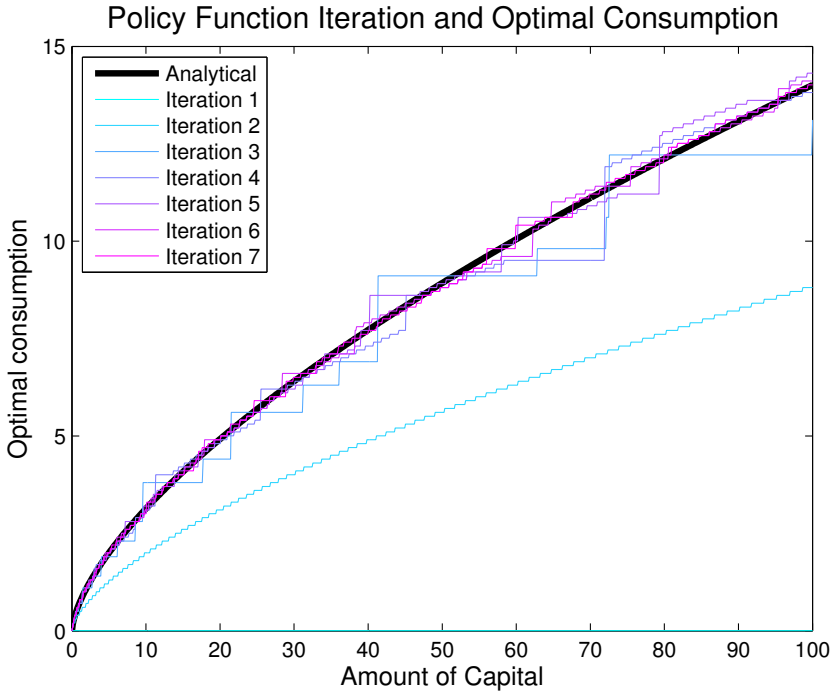

Figure 7.4: Faster Convergence to the Policy Function

7.3 Solving for Structural Parameters Using GMM

So far in this and the preceding chapter we have essentially been simulating models and assuming that we know the relevant underlying structural parameters. We have assumed that we know an individual's discount rate β, and the nature of the production technology of the good that the individual or firm is producing (that is to say we know the parameters γ and θ). For now we will refer to this group of parameters as $\mathbf{\Omega}$:

$$\mathbf{\Omega} = (\beta, \theta, \gamma) \tag{7.9}$$

While these simulations are interesting and useful to predict behaviour under a certain parametric representation of the world, often our interest will be in

inverting this problem. Rather than assume that we know the true underlying parameter set $\mathbf{\Omega}$ and then use this to generate data, frequently we will have data on household behaviour, and from this will be interested in estimating these underlying parameters. This is typically the type of problem that we are used to working with in microeconometrics: that of observing data and recovering estimates to summarise the underlying data generating process.

So, the question now becomes whether the methods we have been looking at here lend themselves to microeconometric modelling. Given the title of this book, you will not be surprised to learn that the answer is a robust 'yes'. Indeed, Keane et al. (2011) point out that "[i]t is a truism that any dynamic optimization model that can be (numerically) solved can be estimated."

7.3.1 A Brief Introduction to Generalised Method of Moments

Perhaps the most flexible ways of estimating such models, and indeed a large range of econometric problems, is by using method of moments, or generalised method of moments (GMM). Without going into too great detail,[8] when using GMM, we define a number of **population moments**. These are expressions which are true in our model.[9] We then estimate our parameters using the principle of analogy. This involves setting the identical **sample moments** to zero in our observed draw of data.

Before jumping into GMM with the dynamic models we've been simulating in this and the previous chapter, it's perhaps worth working through a very familiar example. To do this, we will return to linear regression, much like the OLS function we defined to estimate (1.1) way back in Chapter 1.

[8] There are *many* excellent references to turn to if you are interested in further details on method of moments based estimation. For example, Hall (2005); Cameron and Trivedi (2005), or for those particularly interested, the original Hansen (1982) article provide more extensive details.

[9] Of which we assume the observed data provides a representative picture.

From the basic Gauss Markov assumptions, we have that in the population:

$$E[\boldsymbol{\varepsilon}|\boldsymbol{X}] = 0. \tag{7.10}$$

From this, we can also show that:

$$E[\boldsymbol{X}\boldsymbol{\varepsilon}] = E[\boldsymbol{X}(y - \boldsymbol{X}\boldsymbol{\beta})] = \mathbf{0}. \tag{7.11}$$

We can then use (7.11) to form our sample moment conditions, which gives one moment for each β parameter in our model. As you may be aware, these sample moments look like the following:

$$\boldsymbol{m} = \frac{1}{N}\left[\sum_{i=1}^{N} \boldsymbol{X_i}(y_i - \boldsymbol{X_i}\boldsymbol{\beta})\right] = \mathbf{0}. \tag{7.12}$$

This vector is what we'll refer to as our moment vector and will be of dimension $1 \times k$, where k is the number of independent ($\boldsymbol{X}$) variables in our model. The fundamental idea in GMM is that our estimates $\hat{\beta}$ should be those values which drive the weighted quadratic distance $\boldsymbol{m}W\boldsymbol{m}'$ to zero, (or as close to zero as possible if we have more moments than coefficients)[10]. For consistent estimation we simply need to ensure that our weight matrix W is semi-definite positive (such as an identity matrix), however for efficiency we may be interested in using other weight matrices such as those discussed by Hansen (1982).

This is all well and good in theory, but how does it actually work in MATLAB? Given that we are attempting to minimise $\boldsymbol{m}W\boldsymbol{m}'$, you may suspect that our code will involve minimising some function whose output is this value. If so, you'd be correct! Let's have a look at the below function which sets up these sample moments.

```
function Q = objective(beta,y,X)
% Q = objective(beta,y,X) calculates the moments of a linear
% regression model given input data y (a vector), X (a matrix
% including an intercept of all ones if desired), and the point
% estimates beta.  The true method of moments estimate of beta
% occurs when Q=0.
%
```

[10] We refer to this case as 'over-identification'.

```
% The series of moments that are being fitted here are:
% [E(X_1*u)=0  E(X_2*u)=0 ... E(X_k*u)=0]

%=================================================================
%=== (1) determine sample size N and number of coefficients k
%=================================================================
N = length(y);
k = size(X,2);

%=================================================================
%=== (2) Calculate u and generate the (arbitrary) weight matrix
%=================================================================
u = y - X*beta;
W = eye(k);

%=================================================================
%=== (3) Generate moment vector (1*3) in this case
%=================================================================
m = 1/N*u'*[X(:,1) X(:,2) X(:,3)];
Q = m*W*m';

return
```

This function accepts as arguments our observed y and X data, and a proposed value for the vector of parameter estimates $\hat{\beta}$. As per normal, step through each line on the command line in MATLAB to ensure that you're comfortable with the computations carried out, and to ensure that you're happy that this function returns a scalar value for Q. Now, to calculate our estimates, all we need to do is minimise this function. As we have seen in a number of earlier examples, MATLAB has a host of minimisation routines which are perfect for such calculations.

To test out this code we can return to the `auto.csv` file we worked with in chapter 1. Below we load this into MATLAB, and estimate $\hat{\beta}$ by GMM:

```
>> DataIn   = dlmread('auto.csv');
>> X        = [ones(74,1) DataIn(:,2:3)];
>> y        = DataIn(:,1);
>> [beta,Q] = fminsearch(@(B) objective(B,y,X),[10,0,0]', ...
       optimset('TolX',1e-9));

beta =

  39.4397
  -0.0001
  -0.0058

Q =

  6.3475e-20
```

We see above that by minimising this objective function we get very close to $Q = 0$, and recover the identical point estimates that we found in previous trials. You'll notice that we've specified some particular optimisation settings in our call to `fminsearch`. In any estimation such as this it's very important to carefully set our minimisation criteria. For example, try estimating without this setting. Do you recover the correct point estimates? Fortunately, we have a very safe way to know when our optimisation settings are 'sensitive enough' in this case. Given that each sample moment should be asymptotically equal to 0, our weighted result Q should also be approximately 0. In any situation where $Q \ncong 0$, we'll have reason to think that our GMM estimates have not been successful in setting the sample moments equal to zero, and that our optimisation settings likely need fine tuning.

Exercise: The above code has estimated β using just-identified GMM. However, in this case, we could also consider estimating via tradition method of moments, which simply involves setting each moment exactly equal to zero rather than minimising the quadratic distance. How would you code a method of moments estimator for β in MATLAB? *Hint*: The `fsolve` function may be useful in this case.

7.3.2 Applying GMM to Our Dynamic Model

The dynamic models which we have been generating and simulating in this and the previous chapter are entirely amenable to estimation in a similar way via GMM. While parameter estimates in a linear regression model can be generated by forming moments based on the Gauss-Markov assumptions, our dynamic models can be estimated if we believe that we can make reasonable assumptions about expectations of the stochastic elements of these models.

Perhaps the trickiest part of this process is in motivating and justifying specific moment conditions. To fix ideas in our heads, we will return to the example we have been using all along of a producer/consumer with log utility. Specifically, we will return to the example we have discussed in section 6.4 of the previous chapter. If you are yet to go through this section, however, there is no need to despair. We will simply ask you to convince youself that this example results in observations of the "true behaviour" of 100 individuals based upon their realisations of stochastic shocks ε_t at each point in time. Note that although we did simulate this data, from here onwards we will just be acting *as if* this were real data which we received as researchers considering a particular type of microeconometric issue. Importantly, this implies that we do not know the true data generating process, the values of all parameters in this process, or the nature of the stochastic elements in this 'world'.

Method of moments estimation starts with an assumption that *something* in the population is equal to zero. Let's assume that we have reason to believe that the expected value of the stochastic error term is zero in each period:

$$\mathbb{E}[\varepsilon_t] = 0 \quad \forall \quad t. \tag{7.13}$$

You will note that this is simply an assumption about *one moment* (the mean of a parameter) in the real world. Here we simply assume that we know something about the mean of this unobserved term, and do not assume that we know anything else about its distribution.[11]

[11] This is, perhaps, one of the nicest things about method of moments based estimation. Rather than making a full distributional assumption regarding the nature of the error term, we just make an assumption about one meaningful point in the distribution. This is less demanding than what we have been assuming in earlier estimation techniques where maximum likelihood was used based upon a distributional assumption. We return to this point in section 7.3.3.

Let's also assume that we know the general form of the problem. We'll imagine that we know that our agent is a utility maximizer, subject to a stochastic flow equation for capital:

$$\max_{\{c_t\}_{t=1}^{T}} \sum_{t=1}^{T} \beta^{t-1} u(c_t) \qquad \text{subject to} \qquad k_{t+1} = f(k_t, c_t) + \varepsilon_{t+1}. \tag{7.14}$$

Now, by combining (7.13) with (7.14), we have immediate candidates for moment conditions. The first comes from simply rearranging the capital flow equation, expressing in terms of ε, and taking expectations of both sides. The second comes from the Euler equations. These state that the marginal rate of substitution of consumption between periods should be equal to the marginal return of saved capital. These moment conditions look like the following[12]:

$$\mathbb{E}[k_{t+1} - f(k_t, c_t)] = 0 \tag{7.15}$$

$$\mathbb{E}\left[\frac{u'(c_t)}{\beta u'(c_{t+1})} - f'(k_t)\right] = 0. \tag{7.16}$$

With population moment conditions in hand, we can now fit the sample analogue of these moments using our 'data' on the 100 individuals we simulated earlier. Our goal is to estimate the parameters of the production technology $f(k_t, c_t)$. We start by assuming (correctly) that the functional form for utility is $\ln(c_t)$, and the for production is $\theta(c_t - k_t)^{\alpha}$.[13] Ideally then, we'd like to form these moments, and estimate $\hat{\mathbf{\Omega}}$ from equation (7.9). Unfortunately however, we will find we still have a problem were to try to do this. Typically in dynamic models like the one we are working with here, the discount factor β is not identified without placing strong restrictions on other primitives in the model.[14] Normally then, we will assume some plausible value for β, plug this into our estimation, and based upon this assumption be able to identify the remaining parameters.

Let's try setting up these moment conditions in MATLAB:

[12] Or, with our assumed functional form that $f(k_t, c_t) = \theta(k_t - c_t)^{\alpha} + \varepsilon$, they look like:

$$\mathbb{E}[k_{t+1} - \theta(k_t - c_t)^{\alpha}] = 0$$

$$\mathbb{E}\left[\frac{c_{t+1}}{\beta c_t} - \alpha\theta(k_t - c_t)^{\alpha-1}\right] = 0.$$

[13] This functional form also allows for the flow equation we used in the earlier case that $k_{t+1} = k_t - c_t$.

[14] We will not go into this here, however an exposition can be found in Rust (1994a,b).

```
function [Q] = dynamicMoments(ct,ctp,kt,ktp,params)
  % dynamicMoments(ct,ct+1,kt,kt+1,params) returns the quadra-
  % tic distance (scalar) based on dynamic model moments and
  % specified values of alpha and theta
  alpha  =  params(1);
  theta  =  params(2);
  beta   =  0.9;
  k      =  size(params,2);

  %=============================================================
  % Form moments
  %=============================================================
  m1  = mean([ktp - theta*(kt - ct).^alpha]);
  m2  = mean([(ctp./(beta*ct))-alpha*theta*(kt-ct).^(alpha-1)]);

  %=============================================================
  % Create weight matrix and quadratic distance
  %=============================================================
  W      = eye(k);
  Q      = [m1 m2]*W*[m1 m2]';
return
```

You'll probably notice a few things that we're doing here. Firstly, the moments `m1` and `m2` are based on the functional form outlined above, and we've assumed a value for β. Secondly, we're forming *only* two moments here. You will note, however, from (7.13) that we have assumed that $\varepsilon = 0$ in each of our T time periods, meaning that we could create an overidentified system of moments. We leave this case as an exercise below.

So, to estimate we'll consider just arbitrarily choosing data from one time period and using method of moments. We'll start by re-simulating our 'sample' data, and then use `fminunc`—MATLAB's unconstrained minimisation routine—to minimize our objective function $Q(\Omega)$:

```
>> finiteStochastic;
>> simulateStochastic;
>> opt        = optimset('TolFun', 1E-20, 'TolX', 1E-20);
>> [Omega, Q] = fminunc(@(p) dynamicMoments(con(:,4),con(:,5),...
                                             kap(:,4),kap(:,5),...
                                             p),[1, 1], opt);

Omega =

   0.9832   1.1895

Q =

   2.7558e-08
```

Perhaps as expected given our somewhat 'serendipitous' set of assumptions, we see that this estimation technique works well to recover estimates for α and θ. Our estimates of 0.9832 and 1.1895 are close to the true population values of 0.98 and 1.20, and we'll let you check whether you can improve these by using a larger set of moment conditions to estimate.

Exercise: In the above example we estimated $\hat{\alpha}$ and $\hat{\theta}$ by using two moment conditions based upon ε_5. Generalise this GMM estimation so that rather than using two conditions you use all the (T-1)*2 moments available from ε_t.

7.3.3 Final Thoughts on Estimation

Before closing this chapter we think it's worthwhile to briefly discuss a few additional points on estimating these dynamic models. Firstly, although we motivate estimation via GMM in this chapter, there is nothing in this framework which suggests that this must be the only (or indeed even the best) way to estimate these models. Alternative ways to estimate these models include method of simulated moments (MSM) techniques, and even techniques which do not involve defining moments at all, such as maximum likelihood. While

techniques based around full distributional assumptions such as ML are liable to be less accurate if these assumptions turn out not to hold in reality, in many cases these assumptions will already be built into the structure of our model, meaning that ML is an entirely reasonable and efficient estimation technique.

This brings us to a more general point about these 'structural' estimation techniques. While a more structural approach allows us to pursue highly ambitious questions which may be outside the scope of reduced form estimation techniques, the assumptions which we require to estimate these models are generally much stronger (hence the emphasis on structure!). We won't belabour this point too much, as it is partially outside the scope of a book on computational and microeconometric methods, and especially one of this length. However, we do encourage you to experiment with the code in this chapter to see how it performs under alternative (and less serendipitous) assumptions regarding functional form, discount rate, and so forth, and certainly to turn to more expert opinion if you are interested, such as Keane et al.'s (2011) handbook chapter, and their excellent closing remarks on "how credible are DCDP models?".

7.4 Review

COMMAND	BRIEF DESCRIPTION
linspace	A linearly spaced vector based of (user-defined) values
hot	A colourmap of black, red and yellow for use in visual outputs
fsolve	Solves a system of equations
sparse	Store sparse matrix in a computationally more efficient way
speye	A sparse identity matrix, with only the diagonal stored in memory
legend	Add a legend to a plot

Table 7.1: Chapter 7 commands

7.A Analytically Iterating the Value Function

In what follows, we use V_j to signify the value function, where the subscript $j \in [0, \infty)$ represents the iteration on the value function. Importantly, this number does not have any link to time periods, simply telling us how many times we have iterated over V, and hence how close we are to our solution. From Stokey and Lucas (1989) we know that under a relatively innocuous set of assumptions, the contraction mapping theorem implies that as $j \to \infty$, $\Gamma V \to V$ (or that our value function will converge). To start the iterations we define an initial value function[15]:

$$V_0(k) = 0 \ \forall \ k.$$

We treat V_0 as a proposed solution, where a proposed solution is only verified as the true solution if it is determined that $V_{j+1} = V_j$, otherwise V_{j+1} becomes the new proposed solution, and iteration continues. So, starting from V_0 the first iteration is defined by maximising the functional equation:

$$V_1(k) = \max_{\tilde{k}} \{\ln(c) + \beta V_0(\tilde{k})\} \qquad \text{s.t.} \qquad c = \theta k^\alpha - \tilde{k}. \tag{7.17}$$

Here we use k and $\tilde{k}$ to denote capital in the current and subsequent periods respectively. In this case given that for all k the value of V_0 will be 0, it is optimal to consume all capital, giving a utility maximising consumption of $c^* = \theta k^\alpha$.

[15] This is arbitrary in the sense that starting the iteration from any resolvable value function will still lead us to the true solution.

Substituting this optimal solution into our value function (7.17) gives:

$$\begin{aligned} V_1(k) &= \ln(c^*) + \beta V(\tilde{k}^*) \\ &= \ln(\theta k^\alpha) + \beta 0 \\ &= \ln\theta + \alpha \ln k, \end{aligned} \tag{7.18}$$

and given that $V_1(k) \neq V_0(k)$ we know that our proposed V_0 is not the solution to the Bellman equation.

Now, having the result from the first iteration, we are able to iterate again, and continue the process of iteration until $V_j = V_{j+1}$, in which case we have arrived at our solution. For our second iteration we continue as above:

$$V_2(k) = \max_{\tilde{k}} \{\ln(c) + \beta V_1(\tilde{k})\} \qquad \text{s.t.} \qquad c = \theta k^\alpha - \tilde{k}. \tag{7.19}$$

Maximising (7.19) gives us a first order condition of the following form:

$$\frac{1}{\theta k^\alpha - \tilde{k}} = \frac{\beta\alpha}{\tilde{k}},$$

which, by rearranging implies that $\tilde{k}^* = \frac{\beta\alpha}{1+\beta\alpha}\theta k^\alpha$, and substituting into the flow equation that $c = \theta k^\alpha - \tilde{k}$ gives $c^* = \left(\frac{1}{1+\beta\alpha}\right)\theta k^\alpha$. Substituting these optimal values back into our value function gives that

$$\begin{aligned} V_2(k) &= \ln c^* + \beta V_1(\tilde{k}^*) \\ &= \ln\left[\left(\frac{1}{1+\beta\alpha}\right)\theta k^\alpha\right] + \beta\left[\ln\theta + \alpha\ln\left(\frac{\beta\alpha}{1+\beta\alpha}\theta k^\alpha\right)\right] \\ &= \alpha(1+\beta\alpha)\ln k + \ln\left(\frac{\theta}{1+\beta\alpha}\right) + \beta\ln\theta + \beta\alpha\left(\frac{\beta\alpha}{1+\beta\alpha}\theta\right) \\ &= E_1 \ln k + F_1 \end{aligned}$$

where in the second line the functional form for V_1 comes from (7.18), E_1 and F_1 just denote constants, and once again we can verify that $V_1(k) \neq V_2(k)$.

Now, similarly we can iterate again to calculate $V_3(k)$:

$$\begin{aligned} V_3(k) &= \max_{\tilde{k}} \{\ln(c) + \beta V_2(\tilde{k})\} \qquad \text{s.t.} \qquad c = \theta k^\alpha - \tilde{k}, \\ &= \max_{\tilde{k}} \{\ln(\theta k^\alpha - \tilde{k}) + \beta[\alpha(1+\beta\alpha)\ln\tilde{k} + F_1]\} \end{aligned}$$

and here the relevant first order condition for the above equations is:

$$\frac{1}{\theta k^{\alpha} - \tilde{k}} = \frac{\beta\alpha(1+\beta\alpha)}{\tilde{k}}.$$

Rearranging this FOC gives $\tilde{k} = \frac{\beta\alpha+\beta^2\alpha^2}{1+\beta\alpha+\beta^2\alpha^2}\theta k^{\alpha}$ and $c = \theta k^{\alpha} - \tilde{k} = \left(\frac{1}{1+\beta\alpha+\beta^2\alpha^2}\right)\theta k^{\alpha}$. We can then substitute these into our value function, giving that $V_3(k)$ is:

$$\begin{aligned} V_3(k) &= \ln c^* + \beta V_2(\tilde{k}^*) \\ &= \ln\left[\left(\frac{1}{1+\beta\alpha+\beta^2\alpha^2}\right)\theta k^{\alpha}\right] + \beta\left[\alpha(1+\beta\alpha)\ln\frac{\beta\alpha+\beta^2\alpha^2}{1+\beta\alpha+\beta^2\alpha^2}\theta k^{\alpha} + F_2\right] \\ &= \alpha(1+\beta\alpha+\beta^2\alpha^2)\ln k + F_2 \\ &= E_2 \ln k + F_2 \end{aligned}$$

where once again E_2, F_2 just denote constants.[16]

Here, yet again, we see that $V_3(k) \neq V_2(k)$, however we do start to see a pattern emerging. Indeed, were we to keep iterating *ad infinitum*, we would find that for each iteration j the solution would look like $V_j(k) = E_j \ln k + F_j$. In order to actually resolve this fully, we could keep iterating, forming $V_4(k), V_5(k), \ldots, V_{\infty}(k)$ as above, or we can take advantage of the algebra of geometric series. For the first constant E_j, the limit is as follows:

$$\lim_{j\to\infty} E_j = \alpha[1+\beta\alpha(1+\beta\alpha+\beta^2\alpha^2+\ldots+\beta^{j-1}\alpha^{j-1}] = \frac{\alpha}{1-\alpha\beta}, \qquad (7.20)$$

while for F we can break this down into a number of steps. From F_1 and F_2 we begin to see that the general form of F_j is:

$$\begin{aligned} F_j = \ln\left(\frac{\theta}{1+\beta\alpha+\ldots+\beta^{j-1}\alpha^{j-1}}\right) + \beta\ln\left(\frac{\theta}{1+\beta\alpha+\ldots+\beta^{j-2}\alpha^{j-2}}\right) + \ldots + \beta^{j-1}\ln\theta \\ +\beta\alpha(1+\beta\alpha+\ldots+\beta^{j-2}\alpha^{j-2})\ln\left(\frac{\beta\alpha+\ldots\beta^{j-2}\alpha^{j-2}}{1+\beta\alpha+\ldots\beta^{j-2}\alpha^{j-2}}\alpha\beta\theta\right) \\ +\beta(\beta\alpha)(1+\beta\alpha+\ldots+\beta^{j-3}\alpha^{j-3})\ln\left(\frac{\beta\alpha+\ldots\beta^{j-3}\alpha^{j-3}}{1+\beta\alpha+\ldots\beta^{j-2}\alpha^{j-2}}\alpha\beta\theta\right) \\ +\ldots+\beta^{j-2}(\beta\alpha)\ln\left(\frac{1}{1+\beta\alpha}\alpha\beta\theta\right). \end{aligned}$$

[16] If you wish to do the algebra for F_2, feel free! If your algebra is correct (and we haven't made any mistakes) you will find something like: $F_2 = \ln\left(\frac{\theta}{1+\beta\alpha+\beta^2\alpha^2}\right) + \beta\ln\left(\frac{\theta}{1+\beta\alpha}\right) + \beta^2\ln\theta + \beta^2\alpha\left(\frac{\beta\alpha}{1+\beta\alpha}\theta\right) + \beta\alpha\ln\left(\frac{\beta\alpha+\beta^2\alpha^2}{1+\beta\alpha+\beta^2\alpha^2}\theta\right)$.

This can be simplified into what is essentially two geometric series. The first line of the above equation as:

$$\begin{aligned}\lim_{j\to\infty}\sum_{t=0}^{j-1}\beta^t \ln\left(\frac{1}{1+\beta\alpha+\ldots+\beta^{j-1}\alpha^{j-1}}\theta\right) &= \lim_{j\to\infty}\sum_{t=0}^{j-1}\beta^t \ln\left[\theta(1-\beta\alpha)\right] \\ &= \frac{1}{1-\beta}\ln\left[\theta(1-\beta\alpha)\right]\end{aligned}$$

while the final three lines are:

$$\begin{aligned}\lim_{j\to\infty}\sum_{t=0}^{j-2}\beta^t\beta\alpha(1+\beta\alpha+\ldots+\beta^{j-2}\alpha^{j-2})\ln\left(\frac{\beta\alpha+\ldots\beta^{j-2}\alpha^{j-2}}{1+\beta\alpha+\ldots\beta^{j-2}\alpha^{j-2}}\alpha\beta\theta\right) \\ = \lim_{j\to\infty}\sum_{t=0}^{j-2}\beta^t\frac{\beta\alpha}{1-\beta\alpha}\ln(\beta\alpha\theta) \\ = \frac{1}{1-\beta}\left[\frac{\beta\alpha}{1-\beta\alpha}\ln(\beta\alpha\theta)\right],\end{aligned}$$

in which case we have that

$$\lim_{j\to\infty} F_j = \frac{\ln[\theta(1-\alpha\beta)]}{1-\beta} + \frac{\beta\alpha\ln(\beta\alpha\theta)}{1-\beta}.$$

Of course, this has been an awful lot of algebra, and we might be concerned that we haven't actually found the closed form solution to this value function. Thankfully we've already come across a way we can check this solution: all we need to do is show that iterating once again on the above value function gives us an identical value function (a fixed point). Let's give it a try[17]...

$$V_{\infty+1}(k) = \max_{\tilde{k}}\{\ln(c)+\beta V_\infty(\tilde{k})\} \qquad \text{s.t.} \qquad c = \theta k^\alpha - \tilde{k}. \tag{7.21}$$

As we have done above, we can form the first order condition for (7.21), which gives:

$$\frac{1}{\theta k^\alpha - \tilde{k}} = \frac{\beta\alpha}{(1-\beta\alpha)\tilde{k}}.$$

From here we can rearrange for $\tilde{k}^* = \beta\alpha\theta k^\alpha$, and $c^* = \theta k^\alpha(1-\beta\alpha)$. If we

[17] And please excuse our abuse of notation in (7.21).

substitute these optimal values into our value function we have:

$$\begin{aligned} V_{\infty+1}(k) &= \ln c^* + \beta V_\infty(\tilde{k}^*) \\ &= \ln\left[\theta k^\alpha (1-\beta\alpha)\right] + \beta\left[\frac{\alpha}{1-\alpha\beta}\ln(\beta\alpha\theta k^\alpha) + \frac{\ln[\theta(1-\alpha\beta)]}{1-\beta} + \frac{\beta\alpha\ln(\beta\alpha\theta)}{1-\beta}\right] \\ &= \frac{\alpha}{1-\alpha\beta}\ln k + \frac{\ln[\theta(1-\alpha\beta)]}{1-\beta} + \frac{\beta\alpha\ln(\beta\alpha\theta)}{1-\beta} \end{aligned}$$

and indeed, we find that $V_{\infty+1}(k) = V_\infty(k)$, indicating that we have actually iterated onto the true solution.