Chapter 6

Dynamic Choice on a Finite Horizon

"The thousand times that he had proved it meant nothing. Now he was proving it again. Each time was a new time and he never thought about the past when he was doing it."

Ernest Hemingway, "The Old Man and the Sea"

6.1 Introduction

The steady march of time casts a pall over many economic decisions. The choice to do something today may preclude, restrict, encourage or necessitate certain choices in the future. Among *many* other areas: searching for work, investing in human capital, installing or removing physical capital, and choosing a partner on the marriage market are all decisions which are frequently considered from a <u>dynamic</u> point of view.

While the mathematics behind these dynamic choices are slightly more complex than traditional single-period optimisation problems we discussed in chapter 2, the basic idea is—we think—not too much more demanding. When determining behaviour over a number of periods, a decision maker should aim to equalise the discounted marginal utility *at each point in time*. This is analagous to a consumer's aim to equalise the cost-adjusted marginal utility of each unit of consumption in a static environment[1]. Effectively, while in a static sense we expect that a consumer could not increase utility by rearranging consumption between goods, in a dynamic sense we would expect that such an improvement could not be made by rearranging consumption over time. Indeed, this 'no-arbitrage' type condition has a special name in dynamic optimisation: the **Euler equation**. This is a point we return to when setting up our problems and code in the sections which follow.

The aim of this and the following chapter is not to provide a comprehensive overview of the theory behind dynamic programming. If you are interested in such a review, we would suggest having a look at a number of text-book-length analyses such as that of Adda and Cooper (2003), or chapters of Dixit (1990) and Acemoglu (2008). We *do* however aim to provide a number of self-contained applied examples of dynamic optimisation in MATLAB, along with the corresponding theory behind these situations. While we hope that this will give you a good foot-hold into programming and thinking about these types of problems, applications of this type are vast, so further reading (and experimentation in MATLAB!) is likely to be very useful.

6.2 Dynamic Decisions

6.2.1 Household Consumption

To fix the idea in our heads, let's consider a particular dynamic choice: namely that of a household deciding how much of a particular endowment to consume. Given that households exist over various periods of time, and—we assume—aim

[1] or likewise, for a firm to equalise the marginal output per cost of physical capital and labour in typical (static) optimisation problems.

to maximise total lifetime utility, this is effectively a dynamic decision. For the time being we assume that the household is only interested in their utility from consumption for the next T periods, perhaps because the good will spoil after this time. We also assume that utility from consumption is additively separable over time[2]. This then gives us a familiar utility function of the form:

$$U = \sum_{t=1}^{T} \beta^{t-1} u(c_t), \tag{6.1}$$

where β represents the household's discount factor.

At the beginning of the first period our household will have some stock of the good which we will call k_1, and which it apportions over time as it sees fit. Thus, in each of the T periods the household will consume c_t, giving us a flow equation of the form:

$$k_{t+1} = k_t - c_t. \tag{6.2}$$

This transition equation keeps track of capital, our **state variable** (think $\underline{\text{st}}$ate=$\underline{\text{st}}$ock). From the above we see that the state in any given period depends only upon the state at the beginning of the period, and the decision the individual makes with respect to the choice variable, c, (also known as the **control variable**). From (6.2) we start to see the dynamics of the optimisation problem very clearly. The remaining stock of k in a given period depends upon consumption in the preceding period and the level of k in the preceding period, which itself depends upon decisions made earlier in life (that is unless we are in the initial period in which k_1 is given). Equations (6.1) and (6.2), along with the non-negativity constraints $c_t \geq 0$ and $k_t \geq 0$ allow us to completely characterise the household's (dynamic) problem as:

$$\max_{\{c_t\}_1^T} \sum_{t=1}^{T} \beta^{t-1} u(c_t) \qquad \text{s.t.} \qquad \begin{aligned} \textstyle\sum_{t=1}^{T} c_t + k_{T+1} &= k_1 \\ c_t &\geq 0 \\ k_t &\geq 0. \end{aligned} \tag{6.3}$$

Here you will notice that we have rearranged the series of flow equations (6.2)

[2] This may turn out to be untrue, but we do not entertain this possibility for the time being. Such an assumption is equivalent to assuming that "the marginal rate of substitution between lunch and dinner is independent of the amount of breakfast", an analogy that Dixit (1990) attributes to Henry Wan.

into one equation for ease of presentation (and later ease of computation).

This problem looks remarkably similar to (static) optimisation problems we have already tackled in earlier chapters of this book. Indeed, if we know the form of $u(c_t)$, the discount factor β, the initial endowment k_1, and the number of periods T, we should be able to resolve this problem reasonably easily by using MATLAB's `fmincon` function. Let's define these and have a look. We'll assume for now a log normal utility function $u(c_t) = \ln(c_t)$, 10 time periods, a discount factor of $\beta = 0.9$ and an initial endowment of $k_1 = 100$.

As per normal then, we can consider setting up a MATLAB function to minimise. Consider something along the lines of:

```
function V = flowUtility(T,Beta,C)
   % flowUtility(T,Beta,C) takes T periods of consumption of
   % size C (a Tx1 vector), and calculates the total
   % utility of consumption assuming an additively separable
   % utility function and discount rate Beta.

   c  =  C(:,1);

   t  =  [1:1:T];
   V  =  Beta.^(t-1)*log(c)
   V  =  -V

return
```

If we pass this function the appropriate parameters it will return to us a scalar value `V`, corresponding to total utility from the T periods of consumption. In the final line of this script you will notice that we convert our (positive) utility `V` into a negative value, as although we are interested in *maximising* utility, `fmincon` is a *minimisation* function.

Having written the function we aim to minimise, and having defined all the necessary parameters, let's solve for consumption c_t in each period.

```
>> beta  = 0.9;
>> T     = 10;
>> k1    = 100;
>> lb    = zeros(10,1);
>> ub    = 100*ones(10,1);
>> guess = 10*ones(1,10);
>> A     = ones(1,10);
>> opt   = optimset('TolFun', 1E-20, 'TolX', 1E-20, ...
           'algorithm', 'sqp');
>> c     = fmincon(@(C) flowUtility(T,Beta,C), guess, ...
           A, k1, [], [], lb, ub, [], opt)

c =

   15.3534
   13.8181
   12.4363
   11.1926
   10.0734
    9.0660
    8.1594
    7.3435
    6.6091
    5.9842
```

We see here that along with those parameters we defined above (`beta`, `T`, and `k1`), we pass additional arguments to `fmincon`. Our non-negativity constraint for c is defined as a lower-bound (`lb`) of zero in each of the ten periods, and similarly an upper-bound is defined, as consumption can never exceed 100 (the full amount of the endowment) in any period.[3] As usual, we pass an initial starting point to MATLAB as the vector `guess` (somewhat lazily defining this as equal consumption in all periods). Finally, we set up the flow constraint that total consumption must not exceed the full endowment `k1`. We do this using the vector `A`, defining that $\mathbf{A} \cdot \mathbf{c} \leq k_1$. Remember if at any time you are unsure of what's going on in this command, you can consult MATLAB's help files by

[3] Strictly speaking, there is nothing which *requires* us to include `ub` and `lb`. These will be implied by the flow constraint and utility maximisation (respectively). However, it doesn't hurt, and is consistent with out so we include it here anyway.

typing `help fmincon`), or working through particular steps at the command line.

The resulting vector of values shows us expected consumption in each period. As expected given that $\beta < 1$, we see a downward sloping consumption profile, and we can also confirm that this answer has our household consuming the entirety of their endowment k_1 in line with non-satiation:

```
>> sum(c)

ans =

   100
```

Sensitivity to Input Parameters In the above example we have made a number of reasonably particular assumptions upon which our optimal consumption depends. Principally, we have assumed that the (one) household that we are considering has a particular discount rate, and that their utility from consumption in each period obeys a given functional form. Having written and solved the above optimisation problem once, there is nothing that stops us from doing this many times to see how these underlying parameters and primitive functions affect our results. In the following code we 'simulate' consumption over time for a range of households: those which are remarkably impatient ($\beta = 0.05$), to those which place absolutely no additional weight on their current utility ($\beta = 1$).

```
%==============================================================
%=== (1) Calculate optimal consumption for 0<Beta<1
%==============================================================
result  = NaN(10,20);

for Beta=1:20
    beta_use=Beta/20;
    result(:,Beta)=fmincon(@(C) flowUtility(T,beta_use,C),...
        guess,A,k1,[],[],lb,ub,[],opt);
end
```

```
%=================================================================
%=== (2) Graphical output
%=================================================================
time  =  1:10;
beta  =  0.05:0.05:1;

subplot(1,2,1)
plot(time, result, 'LineWidth', 2)
xlabel('Time', 'FontSize', 12)
ylabel('Consumption', 'FontSize', 12)

subplot(1,2,2)
surf(time, beta, result')
xlabel('Time', 'FontSize', 12)
ylabel('Beta', 'FontSize', 12)
zlabel('Consumption', 'FontSize', 12)
```

You'll notice here that we have effectively solved the problem 20 times, by generating a `for` loop. Whilst `for` loops are not necessarily the fastest way to write code in MATLAB, in some cases this will be the simplest and clearest way to set-up our code[4]. After resolving this problem for each of 20 households with differing degrees of patience, we will likely be interested in visualising these results in some way. We suggest a couple in the second block of the above code. These outputs are available as figure 6.1. In the left hand figure we simply present two-dimensional consumption profiles over time for our range of βs, while the right-hand panel takes advantage of MATLAB's 3-D graphing capabilities.

Exercise: How do our results depend upon the utility function we specify?

Perhaps this leads you to ask how we can actually determine β (and other

[4] This is a point which we will return to extensively in chapter 8, where we discuss the benefits of vectorisation, how to speed up code via parallelising loops, and a number of other exciting tricks that MATLAB offers.

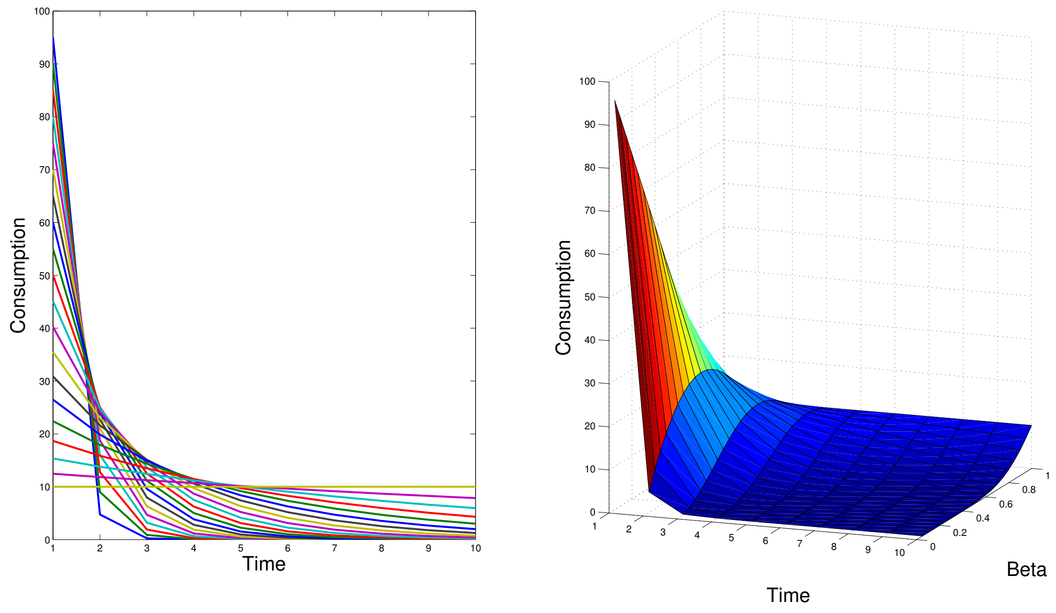

Figure 6.1: Sensitivity of Consumption to Discount Rate

relevant parameters in problems of this type). This is something we will return to more fully in section 6.4 where we discuss taking these microeconomic concepts—namely dynamic optimisation—and building them into microeconometric models. While the programs we have written above have been based upon a single (or representative) household, we can use the power of Matlab to extend these to microeconometric applications which are based on many households who face the same optimisation problem, but who may have different preferences and different discount rates.

So far this entire process hasn't been too much more difficult than standard static optimisation. Before getting too excited about this, we should probably point out that this result is quite artificial for a number of reasons (or perhaps you are ahead of us here...). Principally, we have been able to reduce the 'dynamism' of the problem by rearranging our flow constraints (6.2) into one simple constraint, as presented in the first line of (6.3). In the subsection which follows we will relax this, and consider a slightly more realistic example in which the flow constraints cannot be rearranged into such a nice linear format.

6.2.2 A Small Firm

Let's imagine now that our household from the previous subsection is actually both a producer and a consumer. We could think of this as being a small farming household, and so its consumption decisions in one period affect what it produces in the following periods. Specifically, the flow equation now takes the form:

$$k_{t+1} = f(k_t - c_t, \theta). \tag{6.4}$$

In this case current production depends upon the stock of k remaining from the previous period, (and some technology parameter θ which represents an idiosyncratic time-invariant measure of efficiency). The dynamic nature of the problem essentially remains the same, as the household/firm should decide how much to consume each period to maximise its total utility flow. We can then define the analogous maximisation problem to (6.3):

$$\max_{\{c_t\}_1^T} \sum_{t=1}^{T} \beta^{t-1} u(c_t) \qquad \text{s.t.} \qquad \begin{aligned} &\textstyle\sum_{t=1}^{T} \left[k_{t+1} = f(k_t - c_t, \theta)\right] \\ &c_t \geq 0 \\ &k_t \geq 0. \end{aligned} \tag{6.5}$$

As opposed to the problem in the previous subsection, we can no longer necessarily set up our maximisation problem with one simple linear flow constraint (as we did with the condition $\mathbf{A} \cdot \mathbf{c} = k_1$). The constraint on the first line of 6.5 will likely be both non-linear, and have a highly 'dynamic' nature. This dynamism comes about given that high consumption in early periods not only runs down the stock of k, but also affects the households' ability to produce more k in the future.

By defining the functional form of $f(\cdot)$, we can see how this change affects the setup—and the result—of the problem. Let's assume, for example, that production is adequately captured by a Cobb-Douglas specification:

$$k_{t+1} = f(k_t - c_t, \theta) = \theta(k_t - c_t)^{\alpha}. \tag{6.6}$$

Now, at least without forming a complex and highly recursive equation from (6.6), setting-up a single constraint is not possible. If we wish to solve this dynamic maximisation directly using MATLAB's optimisers, we then must form a series of T (non-linear) constraints to account for (6.6) in each period.

Fortunately for problems of this type, `fmincon` allows us to define and solve for non-linear constraints. Much as we can form linear constraints of the form $\mathbf{A}\cdot\mathbf{c} \leq k_1$ and $\mathbf{Aeq}\cdot\mathbf{c} = k_1$, we can form non-linear constraints $\mathbf{D}(\mathbf{c}) \leq \mathbf{k}$ and $\mathbf{Deq}(\mathbf{c}) = \mathbf{k}$. We pass `fmincon` these non-linear constraints as a function, in precisely the same way that we have been defining and pointing to the objective function we wish to maximise. Rather than belabour this point in words, let's work through an example below in code:

```
function [d,deq] = flowConstraint(CK,T,K1,theta,alpha)
  % flowConstraint(C,k,T,K1,theta,alpha) sets up the system
  % of constraints k_{t+1}=\theta (k_t-c_t)^\alpha.  It
  % requires CK, a Tx2 matrix of consumption and capital
  % values in each of the T periods, K1, the stock of k at
  % the beginning of the first period, and the production
  % function parameters alpha and theta.

  cap   =   CK(:,2);
  c     =   CK(:,1);
  k     =   [K1; cap];

  for t = 1:T
       deq(t)  =  k(t+1) - theta * (k(t) - c(t))^alpha;
  end
  d    =  [];

return
```

We suggest you run through the above code carefully to ensure that it makes sense to you. There are a number of things going on here, some of which may not be entirely obvious. For example, it is important to note that non-linear constraints *must* return two outputs (which we call `d` and `deq` in the first line of our function). These correspond to strict non-linear equalities, and non-linear inequalities. Given that (6.4) is a system of T *equality* constraints, we just define an empty vector for the inequality constraints `d`.

Try experimenting with the `flowconstraint` function by passing it the entire set of arguments. Try situations in which `deq` equals zero (perhaps where $\alpha = 1$ and $\theta = 1$) and where `deq` $\neq 0$. *A useful hint*: in order to see both outputs (`d` and `deq`) you must request these explicitly from MATLAB.

Thus, having defined our non-linear constraints which make production in one period depend upon remaining capital from previous periods, and having defined the utility function we wish to maximise in the previous section, we can make a call to `fmincon`. As long as you still have the parameters from the previous example stored in memory (namely `beta`, `T`, and `k1`), we can enter the remaining code below:

```
>> theta = 1.2;
>> alpha = 0.98;
>> lb    = zeros(10,2);
>> ub    = 100*ones(10,2);
>> guess = [10*ones(10,1), [90:-10:0]'];
>> opt   = ('TolFun', 1E-20, 'TolX', 1E-20, 'algorithm','sqp',...
             'MaxFunEvals', 100000,'MaxIter', 2000);
>> Result = fmincon(@(CK) flowUtility(T,beta,CK), guess,[],[],...
    [],[],lb,ub, @(CK) flowConstraint(CK,T,k1,theta,alpha),opt)

Result =

   16.5011   91.7125
   15.9856   83.3386
   15.5165   74.8041
   15.0944   66.0245
   14.7213   56.9014
   14.4010   47.3159
   14.1408   37.1173
   13.9543   26.1024
   13.8695   13.9624
   13.9624    0.0000
```

We see that our optimisation returns to us a two column matrix with $T = 10$ rows. This two column matrix is our result for `CK`: consumption and remaining capital in each period. This is of course precisely what we have asked MATLAB to give us by using the 'function handle' `@(CK)` in the `fmincon` command.

Finally, perhaps we are interested in producing graphical output rather than simply having tabular output in the form of columns. We track consumption and remaining capital in the following graph (and accompanying code). Once again we see that as expected, all capital is consumed, and, given the particular technology parameters defined, that our household/firm has an approximately downward sloping consumption profile.

```
>> plot(T, Result(:,1), '--r', T, Result(:,2), 'linewidth', 2)
>> xlabel('Time', 'FontSize', 14)
>> ylabel('C_t,k_t', 'FontSize', 14)
>> legend('Consumption', 'Capital Remaining', 'Location', 'NorthEast')
>> title({'Firm Consumption and Investment', ...
    '\beta=0.9, \theta=1.2, \alpha=0.98'}, 'FontSize', 16)
>> print -depsc DynamicBehaviour
```

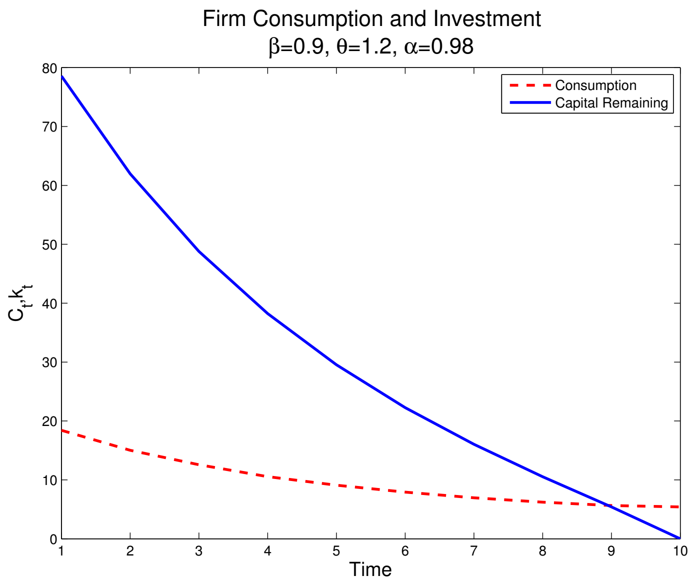

Figure 6.2: Dynamic behaviour of a household firm

6.3 Introducing the Value Function

When we resolve these optimisation problems using MATLAB's solvers (the `fmincon` function), we are solving for consumption directly, and effecitvely in one shot. This is what Adda and Cooper (2003) refer to as **Direct Attack**, and corresponds to Stokey and Lucas (1989)'s sequence problem. The basic intuition is that MATLAB (or indeed you, the MATLAB user,) define(s) a vector of possible values for consumption at each period, and then the optimal outcome is found by shifting consumption between periods until further utility gains are exhausted. If we were to solve (6.3) algebraically for the first-order conditions, we would find that this implies a series of equations:

$$\begin{aligned} u'(c_t) &= \beta u'(c_{t+1}) \\ u'(c_t) &= \beta^2 u'(c_{t+2}) \end{aligned}$$

and so forth. These are the Euler equations, and when these hold this suggests that further gains from rearranging consumption over time may be unable to be made.

This method of (numerical) direct attack is not the only way to consider resolving dynamic optimisation in MATLAB. We could also consider using **Dynamic Programming**. The idea behind dynamic programming is that rather than solving a complex optimisation which considers consumption in T periods all at once, we can break this down into a number of much simpler optimisation problems. Intuitively, from the point of view of the household firm in the final period, the problem is quite simple: maximise utility given the total amount of remaining k. Of course, given that k has no value to the household beyond the T^{th} period (we assume that the good spoils), their optimal behaviour is to consume all remaining k. Then, having 'resolved' for the optimal final period behaviour, we can consider the consumer's behaviour in period $T-1$. In this case their goal is maximise the value of current consumption and the discounted value of future consumption. Given that we know the value of future consumption (we resolved this in the first stage), this is just a matter of how much to consume in $T-1$, and how much to save for T which will then be consumed optimally. We can then continue this process, moving to period $T-2$, where the decision becomes how much to consume in this period, and how much to consume in the future ($T-1$ and T). Thus, we solve a series of T optimisation problems which are effectively two period in nature: now and the future.

The important element of dynamic programming is that at each step we can form a single summary statistic for 'the future', which is simply the value of all remaining capital when consumed optimally. Thus, to solve for any given period, we must have already solved for 'the future'. For this reason in dynamic programming we follow a process of backwards induction: we first solve for the final period, then for the penultimate which depends upon the final, then for the second last which depends upon the penultimate and the final, and continue this until we arrive at the first period. Once solving for optimal behaviour in each subproblem of two periods, we combine all these optimal subproblems to find the final solution.

This may all seem slightly abstract, so we will introduce some mathematical notation, and then see how it works in practice in Matlab. We first introduce the idea of the value function. This summarises the value to the household of a given amount of capital, assuming that this capital is used optimally. Whilst we can't say much about the value function yet, one thing we *do* know, is that the value of any capital which remains beyond the final period is zero, given that it spoils. This allows us to define the following:

$$V(k_{T+1}) = 0 \qquad \forall\ k. \tag{6.7}$$

Effectively, for for any amount of remaining k after the final period, the future value flow to the household is zero. This then gives us a place to start for our backwards inductions. When making their optimal decision in period T, the household resolves:

$$V(k_T) = \max_{c_T} \{u(c_T) + \beta V(k_{T+1})\} \tag{6.8}$$

where $k_{T+1} = k_T - c_T$. Given that we already have our terminal condition from (6.7), the equation 6.8 can be solved for any k_t.[5] This solution is precisely our value function for period T: that is, it tells us the total value to the household of entering period T with some amount K_T, and then behaving optimally. This in turn allows us to consider the decision in k_{T-1}:

$$V(k_{T-1}) = \max_{c_{T-1}} \{u(c_{T-1}) + \beta V(k_T)\}. \tag{6.9}$$

[5] The solution for any value of k will be $k_T = c_T$. Why?

Here we start to see the process of backwards induction emerging. Once we solve (6.9), we can then move on and solve for $V_{T-2}(k)$, and continue (or in reality instruct MATLAB to continue) until it finally reaches $V_1(k)$.

Let's have a look at how a problem like this would be resolved in practice:

```
%============================================================
%=== (1) Prompt user to input parameters
%============================================================

Beta        =  input('Input Beta:');
T           =  input('Input time:');
K1          =  input('Input initial capital:');
grid        =  input('Input fineness of grid:');

K           =  0:grid:K1;
V           =  [NaN(length(K),T), zeros(length(K),1)];

%============================================================
%=== (2) Loop over possible values of k_{t} and k_{t+1}
%============================================================
aux    =   NaN(length(K),length(K),T);
for t  =   T:-1:1
    for inK   =  1:length(K)
        for outK  =  1:(inK)
            c                 = K(inK)-K(outK);
            aux(inK,outK,t)  = log(c)+Beta*V(outK,t+1);
        end
    end
    V(:,t)=max(aux(:,:,t),[],2);
end

%============================================================
%=== (3) Calculate optimal results going forward
%============================================================
vf   = NaN(T,1);
kap  = [K1; NaN(T,1)];
```

```
con  = NaN(T,1);

for t=1:T
    vf(t)      =  V(find(K==kap(t)),t);
    kap(t+1)   =  K(find(aux(find(K==kap(t)),:,t)==vf(t)));
    con(t)     =  kap(t)-kap(t+1);
end

%============================================================
%=== (4) Display results
%============================================================
[kap([1:T],:),con]
subplot(2,1,1)
plot([1:1:T],[con, kap([2:T+1],:)], 'LineWidth', 2)
ylabel('Consumption, Capital', 'FontSize', 12)
xlabel('Time', 'FontSize', 12)
legend('Consumption', 'Capital')

subplot(2,1,2)
plot([1:1:T], vf, 'Color', 'red', 'LineWidth', 2)
ylabel('Value Function', 'FontSize', 12)
xlabel('Time', 'FontSize', 12)
```

In order to get a handle on what this code is doing, it's probably best to start in section (2). Ignore for the time being the outer loop (which starts with `for t = T:-1:1`), and focus on the inner two loops. Here we define a matrix called `aux` (in reality a 3-dimensional matrix, but we'll get to that...), which looks very similar to the formula to the value function we've written down in (6.8) and (6.9). This is essentially what `aux` is. It shows us—for each possible entry value of k—the value to the household of a choosing a particular exit value of capital (and corrsponding level of consumption). For example, if we enter a given period with 20 units of k_t, the household could choose to consume 20 now and 0 in the future, 19 now and 1 in the future, 18 now and 2 in the future, and so forth. So, `aux` tells us the value for all possible entry values of k_t *and* all possible exit values of k_{t+1}.

Once we've calculated this matrix for all possible input and output capital values, we can calculate the optimal decision for each possible input capital. This is what we do 3 lines below the `aux` matrix. The matrix `V()` tells us the best possible behaviour for any given k_T—for example, were we to arrive to a given period with $k_T = 20$, we may find that the optimal choice is to consume 5 now, and save 15 for the future (and so can discard the other 20 possible combinations). You may wonder why we bother doing this for each possible input capital value. For example, why is it important to know what the household would do if it were to arrive with 20 units, if in reality it arrives and has 19 units of capital? The reason why we have to program such a computationally intensive process is that we won't know what decisions are made for consumption (and hence capital) in each period until we completely solve the model, and to be able to solve the model we must know the what the value function looks like in future periods.[6] We call the vector `V` the value *function* because it is a solution for all possible values of k. While it may not necessarily offer a closed form solution in the form of an algebraic function, numerically at least MATLAB allows us to calculate the output value for V over some domain of k—precisely the definition of a function with which we are familiar.

This brings us to the heart of dynamic programming (at least when considering problems with a finite horizon). When resolving, we must iterate backwards to solve the model, and only then can we iterate forward to obtain the objective function. This is why you see two loops involving time (those starting with `for t =`...) in the above code. The first of these loops calculates the value function starting in period T and counting backwards to the first period. Once having calculated the value function, the second loop determines how much capital to consume in each period, starting from period 1 and ending in period T. The reason we must start in period 1 in this case is because this is the only period where we know with certainty what the beginning capital will be. For example, in the code in sections 6.2.1 and we know that the initial endowment or stock of capital was 100. Effectively we need something like this initial condition to pin down the solution.

Now that we've discussed a few of the more cental parts of the above code, let's run it and see what happens. In order to test how this compares to a direct attack using MATLAB's native optimisers, we will use the same values as earlier

[6] In the next section we will also discuss another reason why such a process can be useful.

when prompted by our code to input the parameters:

```
 >> backwards_induc
Input Beta:0.9
Input time:10
Input initial capital:100
Input fineness of grid:0.25

ans =

  100.0000   15.5000
   84.5000   13.7500
   70.7500   12.5000
   58.2500   11.2500
   47.0000   10.0000
   37.0000    9.0000
   28.0000    8.2500
   19.7500    7.2500
   12.5000    6.5000
    6.0000    6.0000
```

When we compare the consumption values—which are returned as the second column of the above output—with those calculated from the direct attack in section 6.2.1, we will note that we have lost some precision. This occurs due to the fact that here we had to discretise the possible values of k_t and k_{t+1}. You will note that in section (1) of the code we define the possible values of capital as `K = 0:grid:K1;`. This allows the household to choose any possible values between 0 and full amount `K1`, in steps of size `grid`. Given that in the above example we have specified a `grid` of 0.25, we are losing some precision in our solution. Depending upon the computational resources you have available, you may wish to see how this solution changes as we define finer and finer grids over which to search.[7]

[7] For general interest, with a grid size of 0.05 we find that `Result(:,2)' = 15.3500 13.8000 12.4500 11.2000 10.1000 9.0500 8.1500 7.3500 6.6000 5.9500`.

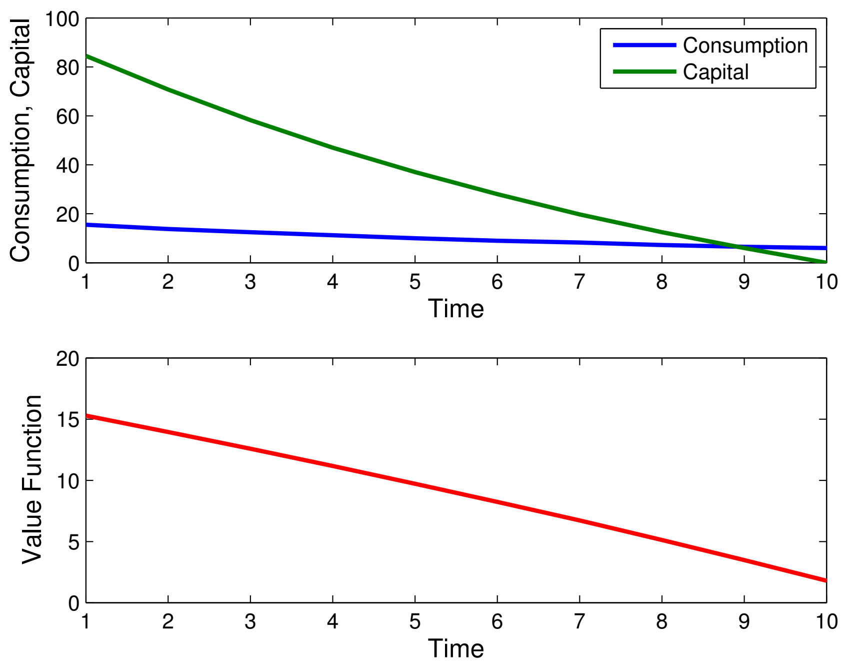

Figure 6.3: Resolving dynamic behaviour using a value function

Exercise: In the above code we have solved for a flow equation of the form $k_{t+1} = k_t - c_t$. Modify the code to solve for $k_{t+1} = \theta(k_t - c_t)^\alpha$ (and confirm that this is correct by referring to the results from section 6.2.1). *A useful hint:* this can be incorporated into the above code with two fairly minor changes. Concentrate only on the formulas which define consumption (`c` and `con`).

6.3.1 Computational Accuracy, Curse of Dimensionality and Memoization

In the above example we have seen that the accuracy of the solution of these types of problems depends upon the grid size we ask MATLAB to search over. Of course, if we want a highly accurate solution, we could just specify a very fine grid, such as an increment of 0.001. The problem with such an approach however, is that this will very quickly become quite demanding on the processing capacity of many personal computers unless we are quite careful about the way we write our code (and not long after, even *if* we are quite careful).

The main bottleneck in this code comes about as we need to calculate an intermediate step for each possible capital pair combination (as we see in the `aux` matrix in the last code example). For example, if we specify a grid search of 0.1 (which allows us to calculate optimal consumption to one decimal place), we see that:

```
>> size(aux)

ans =

       1001 1001 10
```

Similarly, when we try with a grid size 0.01, we have that `size(aux)=10001 10001 10`, and were we patient (or brave?) enough to try with a grid size of 0.001, we would be dealing with a matrix with 100 billion individual elements. Whilst perhaps we would be willing to accept waiting a reasonable amount of time to solve this one problem very accurately, it is unlikely that we could afford such a luxury if we were resolving this for many households (rather than one), or if rather than dealing with one state variable we were dealing with multiple. Such a situation is well known in Dynamic Programming, and was labelled the **curse of dimensionality** by Bellman (1957) when introducing the principles we've laid out above. Whilst we will not delve too deeply into solutions to this problem here, we will provide some discussion of alternative optimisation techniques in chapter 7, and refer the interested reader to the large body of work on dynamic optimisation and interpolation, perhaps starting with Keane and Wolpin (1994) or the text-book exposition of Adda and Cooper (2003).

Given the somewhat demanding form in which we have resolved the above dynamic programming problem, we likely want to avoid solving it repeatedly were we to use this in microeconometric applications with various individuals. MATLAB, and computational-based numerical solutions in general, offer a way to do precisely this. **Memoization** (Michie, 1968) refers to the process of 'remembering', rather than re-computing, results for use in subsequent analysis. As Michie suggests, "It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution. A simple but effective rote-learning facility can be provided within the framework of a suitable programming language". Above in step 2 of our code we entirely solve the problem for any possible starting value and finishing value of capital in each period. Only once we have completely solved this problem (and stored all possible solutions in `aux` and V!) do we actually determine what the household does when starting with a capital value of $k_1 = 100$. However, if we suddenly become interested in a household whose $k_1 = 50$, we have memoised our solution from step 2, so need to do no further backwards induction. Similarly, if our household unexpectedly receives an additional amount of capital in between periods 3 and 4, we simply follow our memoised solution, avoiding the main cost of calculation. We will see the importance and flexibility of such a situation below, where we consider stochastic dynamic programming.

6.4 Shocks, Uncertainty, and Microeconometrics

Having gone through the above code and discussion, this more or less gives us the tools required to set up any finite horizon dynamic programming problem. Perhaps the most obvious remaining question is how to build stochastic elements into these types of problems. Almost all economic processes are stochastic in some sense, whether it be due to uncertainty about what will happen to state variables like capital in future periods, uncertainty about what the decision maker will want to do in future periods, uncertainty in external parameters and events, measurement error in observational data, and so forth. Indeed, microeconometrics is entirely based upon the existence of unobservable elements. Once we try to fit these dynamic models to data, such an extension will be fundamental, as in real data, there will be shocks to a dynamic process.

The problems we have looked at so far have all been deterministic in the sense that we knew with certainty what the decision maker would face in future periods. For example, in the household consumption example we knew that future capital would just be current capital minus current consumption, while in the firm example we knew that capital—our state variable—follows a Cobb-Douglas process: $k_{t+1} = \theta(k_t - c_t)^\alpha$. A much more relasistic situation might be one where future capital follows something like the aforementioned process, but which is subject to positive or negative shocks.

Let's imagine for the moment that evolution of capital is subject to a stochastic shock, so our transition equation (6.6) is now revised as:

$$k_{t+1} = f(k_t - c_t, \theta, \varepsilon_{t+1}) = \theta(k_t - c_t)^\alpha + \varepsilon_{t+1}. \tag{6.10}$$

Here ε could be thought of as an unknown return on investment, which is only resolved in the subsequent period. However, at moment t, the decision maker will be entirely aware of current k_t, all technology parameters, and we assume, the distribution of possible ε_{t+1} (and hence k_{t+1}).

Now, rather than deciding between consumption now and consumption in the future, the decision must be framed in terms of consumption now and *expected* consumption in the future. This suggests a re-writing of the dynamic programming problem as:

$$V(k_t) = \max_{c_t}\{u(c_t) + \beta\mathbb{E}[V(k_{t+1})]\} \tag{6.11}$$

where k_{t+1} is represented in (6.10), and the expectation is taken over the distribution of ε_{t+1}.

In previous sections, we have illustrated these concepts by assuming certain vaules and functional forms for our key parameters and functions. Similarly, here we will assume a reasonably simple structure for the ε term. Specifically, we assume that ε takes two possible states: low and high. We will assume that each state occurs with a probability of $\frac{1}{2}$, and that returns in each case are $\varepsilon \in \{-2, 2\}$.[8] Hence, we can effectively fully characterise the stochastic portion of this problem with two vectors: a vector of returns: [-2, 2], and a vector of probabilities [0.5, 0.5]. Of course, were we to enter these vectors into MATLAB, finding the expectation of ε should be a relatively straightforward process.

Now, with the incorporation of stochastic elements into the optimisation prob-

[8] For those interested in a more extensive discussion of modelling stochastic processes with dependence in dynamic systems, we point you to any of a multitude of resources which discuss the Markov property and Markov chain for shocks. Stachurski (2009) for example provides a very nice overview.

lem, we should rewrite the backwards induction to allow us to compute the value function `V` for each possible future value of k at each time period. Fundamentally, this involves an additional step, as rather than just having that $k_{t+1} = \theta(k_t - c_t)^\alpha$, we have that $\mathbb{E}[k_{t+1}] = \theta(k_t - c_t)^\alpha + \sum_j^J \pi_j \times \varepsilon_{t+1,j}$, a formula we translate to MATLAB in the line which calculates `EnextK`.

```
clear; clc
%================================================================
%=== (1) Setup parameters
%================================================================
epsilon  =  [2 -2];  PI    =  [0.5 0.5];
Beta     =  0.9;
alpha    =  0.98;
K1       =  100;
grid     =  0.2;
T        =  10;
theta    =  1.2;

K        =  0:grid:K1+max(epsilon);
V        =  [NaN(length(K),T), zeros(length(K),1)];
aux      =  NaN(length(K),length(K),T);

%================================================================
%=== (2) Loop over possible values of c, k and epsilon
%================================================================
for t = T:-1:1
   fprintf('Currently in period %d\n', t)
   for inK = 1:length(K)
      for outK = 1:inK
         c               =  K(inK)-(K(outK)/theta)^(1/alpha);
         EnextK          =  theta*(K(inK)-c)^alpha+epsilon*PI';
         position        =  round(EnextK/grid + 1);
         aux(inK,outK,t) =  log(c)+Beta*V(position,t+1);
      end
   end
   V(:,t)=max(aux(:,:,t),[],2);
end
```

Following on from the earlier code that we have gone through, much of the above looks similar, but there are two additional lines in the second block of code. Firstly, the line `EnextK ...` which we alluded to above. You'll note that the difference here is that we take the expectation ε by multiplying the matrix of possible shocks (ε) by the transpose of the probability of a given shock occurring (π). Secondly, we add a line to ensure that this predicted k_{t+1} value will lie in the future value function. Given that we do not necessarily want to restrict the values of ε and π in any way, but we do need to limit the state space for ease of computation, `position` just rounds $\mathbb{E}[k_{t+1}]$ to the closest value in our capital grid. In dynamic problems with continuous state variables, discretisation steps such as this are necessary for computation.

If we run the above code in Matlab, this calculates the value function at each of the (ten) time periods, and for each possible (optimal) capital–consumption pair. Having run this once, we can save these results (that is to say memoise the above function), and then consider particular realisations of the shocks, and the resulting consumption paths. As a decision maker's particular consumption path depends upon the values of ε at each point in time, there is not *one* optimal result. In the below code we consider 100 different individuals, and specific draws from the distribution of ε.

```
%==========================================================
%=== (1) Setup parameters, simulate shocks
%==========================================================
people               =   100;

epsilon              =   randi(2,people,T+1);
epsilon(epsilon==1)  =   -2;

vf                   =   NaN(people,T);
kap                  =   [K1*ones(people,1) NaN(people,T)];
con                  =   NaN(people,T);

%==========================================================
%=== (2) Determine consumption based on simulated shocks
%==========================================================
for p=1:people
```

```
    for t=1:T
        position    =   round(kap(p,t)/grid+1);
        vf(p,t)     =   V(position,t);
        kap(p,t+1)  =   K(find(aux(position,:,t)==vf(p,t)));
        con(p,t)    =   theta*kap(p,t)^alpha-kap(p,t+1);
        kap(p,t+1)  =   kap(p,t+1)+epsilon(p,t+1);
    end
end

%==============================================================
%=== (3) Output
%==============================================================
plot([1:1:T], con)
ylabel('Consumption', 'FontSize', 12)
xlabel('Time', 'FontSize', 12)
title('Simulated Consumption Paths', 'FontSize', 16)

figure(2)
hist(sum(con,2))
title('Lifetime Consumption', 'FontSize', 16)
```

Here we calculate each person's actual outcome for consumption (`con`), capital (`kap`) and the resulting value function (`vf`) at each point in time. The loop in the second block of code outlines the optimal decision for each firm. Starting at period 1 (and $k_1 = 100$), the optimal future value function is calculated (from our memoised `V`), which implies the optimal consumption and optimal capital with which to exit the period. Then, prior to making the decision in the following period, the shock ε is realised, giving the actual capital the decision maker has to work with.

Figure 6.4 describes these optimal consumption paths for our 100 simulated decision makers. As you would perhaps expect, in period 1 (where each firm has the same capital endowment) all firms decide to act in precisely the same way. In following periods however, depending on the actual realisations of ε (and hence k), optimal behaviour diverges.

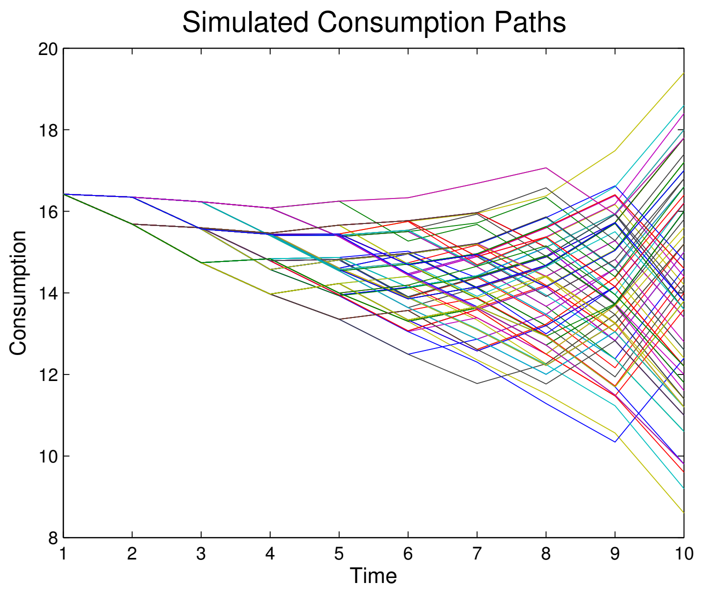

Figure 6.4: Simulated Consumption in a Stochastic Model

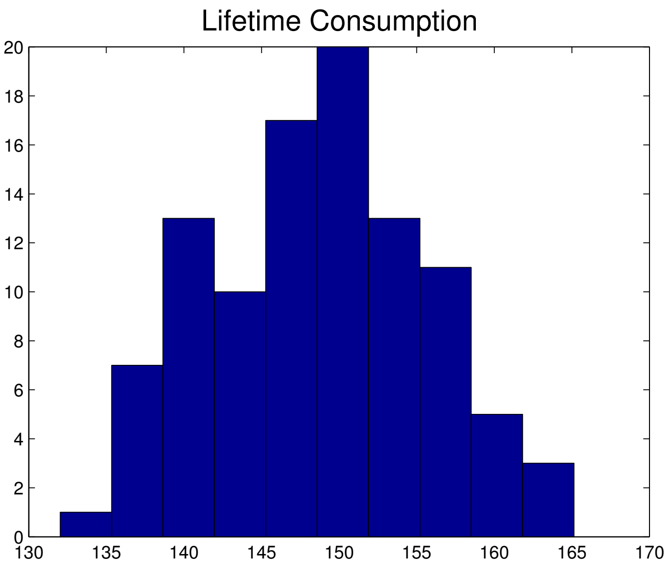

Figure 6.5: Total Simulated Consumption in a Stochastic Model

6.5 Review

COMMAND	BRIEF DESCRIPTION
subplot	Display multiple graphs on one output
input	Prompt user input from the keyboard
find	find location(s) of exact coincidence in a matrix
print	print to disk the item currently in graphical memory
clc	clear results screen
figure	output various figures in a MATLAB script

Table 6.1: Chapter 6 commands