Lifecycle Consumption Model Description

Peter Maxted

August 27, 2017

Contents

1 Introduction

*** All Matlab Code and written documents are works in progress. Please let me know of any errors that you find or sections that are unclear.*

The purpose of this document is to give a detailed description of the Matlab code used to simulate simple buffer-stock consumption models. For users looking for a quick summary, there is a much smaller README.txt file also included in the code folder.

Section 2 outlines the general form of the consumption model that this Matlab code is designed to simulate. Section 3 focuses at a descriptive level on the algorithm I use to simulate the consumption model. More detail on the code itself is given in Section 4. Section 4 is relatively short, however, since the majority of the Matlab coding details are left to the comments in the files themselves. Instead, Section 4 is meant to discuss the purpose of each Matlab file, and how each file relates to the algorithm of Section 3. Finally, the code allows for exponential discounting, sophisticated quasi-hyperbolic discounting, and naive quasi-hyperbolic discounting. These time preferences are outlined in Section 5. Throughout this file I will try to refer to variables that actually exist in the Matlab code with the formatting `<variable>`.

2 General Consumption Model

2.1 Dynamic Budget Constraint

First I outline the general consumption model that this code is intended to simulate. Starting with the dynamic budget constraint,

$$x_{t+1} = R(x_t - c_t) + \overline{y}_{t+1} + \epsilon_t, \tag{1}$$

where x_t is cash on hand and c_t is consumption in period t, $R-1$ is the interest rate, $\overline{y}_t$ is expected income in period t, and ϵ_t is an i.i.d. noise term in the income process. This code imposes the restriction that $c_t > 0$. Though not required, the Matlab code allows for liquidity constraints such that $c_t \leq x_t - \frac{L}{R}$ for some exogenous liquidity constraint L (L is denoted `<borrowingLimit>` in the code).[1]

[1] If borrowing constraint L is set by the user, the code forces the agent to choose consumption such that $R(x_t - c_t) \geq L$. Rearranging this condition yields $c_t \leq x_t - \frac{L}{R}$. By the dynamic budget constraint, $L < 0$ means that the agent is allowed to take on debt.

2.2 Income Process

The Matlab code assumes that each agent lives for only a finite number of years, denoted by variable `<yearsAlive>`. The code also allows for this finite life to be split into work and retirement, given by variables `<yearsWork>` and `<yearsRet>`, respectively. Income can only be earned during working years, and cannot be earned in retirement. Denote the income process $y_t = \overline{y}_t + \epsilon_t$. Included in the Matlab code are two choices for the income process during working years. One option is deterministic — $y_t = 1$ for all periods t in which the agent is working. The other option is stochastic, and is parameterized such that $y_t \sim^{iid} Uniform[0.5, 1.5]$. This parameterization implies that $\overline{y}_t = 1$ and $\epsilon_t \sim^{iid} Uniform[-0.5, 0.5]$ for all t in which the agent is working. Retirement in the model simply means $y_t = 0$.

2.3 Utility and Value

The consumer is given log-utility in the Matlab code.[2] For this section I'll stick to exponential discounting, but the code also allows for both sophisticated and naive quasi-hyperbolic discounting (more on this later in the document). Because each agent lives for only finite time, the code simulates a non-stationary Bellman equation. Specifically, in each period t the agent solves:

$$V(x,t) = \max_c u(c) + \delta E_t[V(R(x-c) + y_{t+1}, t+1)] \tag{2}$$

Value function $V(\cdot,\cdot)$ is written with two inputs to make clear that there are two state variables, assets x and the period t. There is one choice variable, which is consumption c. Further, I assume that for all $t > yearsAlive$,

$$V(x,t) = \begin{cases} -\infty & \text{if } x < 0 \\ 0 & \text{if } x \geq 0 \end{cases}.$$

This assumption on the value function ensures two useful properties: (i) the optimizing agent will never die with negative assets (often called a *No-Ponzi Condition*); and (ii) the optimizing agent will fully consume all remaining assets in the final period of life (i.e., the policy function when $t = yearsAlive$ is to set $c_{yearsAlive} = x_{yearsAlive}$). Explaining (i), if the agent consumes more than cash on hand x in the final year of life, then the agent will pass on negative assets to the next period and earn a value of $-\infty$, which is clearly suboptimal. Explaining (ii), since $V(x,t) = 0$ when $x \geq 0$, the agent yields no utility from passing off

[2] This can easily be changed. See the batch file line `<util = @(x) log(x);>`, which defines the utility function.

positive assets to the next period. Since the agent does earn utility from consumption in the final period of life, it will therefore be optimal to consume all remaining assets.[3]

One issue worth squaring away now is that you'll see the consumption model described above assumes that agents are finitely-lived. However, we may also want to use the Matlab simulation for infinitely-lived agents. Luckily, the Bellman equation (2) is a contraction mapping when $\delta < 1$.[4] Therefore the simulation gets arbitrarily close to a stationary, infinitely-lived, consumption model as `yearsAlive` increases. Thus, the user interested in simulating an infinitely-lived stationary consumption model should set `yearsAlive` to a very large value as this will approximate an infinitely-lived agent.

3 Consumption Model Simulation

Now that the analytical model has been outlined, the next step is to discuss how the consumption model is simulated in Matlab. The simulation can be split broadly into 3 steps — (1) model calibration; (2) policy and value function calculation; (3) simulation of individual profiles. In the model calibration phase, the user enters the model parameters that they want to simulate (e.g., R and δ). In the policy and value function calculation stage, the code uses backward induction (as described in Lecture 5 of 2010C) to numerically calculate the policy and value functions for the calibration specified by the user in step (1). Finally, the simulation uses these policy functions to generate consumption profiles for 5000 simulated agents. If the model is deterministic then all agents will follow exactly the same path of consumption. However, if income is stochastic then each of the 5000 agents will follow a potentially different path governed by their random income realizations. Each of these steps is expanded below.

[3] The assumption that the agent derives no utility from passing off wealth after death is a very simple assumption. I make this assumption so that the code aligns with the Value Function Iteration methodology discussed in Lecture 5 of 2010C. However, one could add realism to the model by assuming that the agent generates a bequest value from passing wealth off to her children. For interested students, there is a large economic literature on bequests and the effect of *bequest motives* on lifecycle consumption/savings decisions. In fact, for 2010C students it will come up next quarter when discussing Ricardian Equivalence with finite lives / overlapping generations. A simple Google search of 'bequest motive' will get any interested student started.

[4] This statement assumes that the agent is an exponential discounter.

3.1 Step 1: Model Calibration

→ *Relevant Matlab file for Step 1 is bufferstock_batch.m*

As long as the user is willing to stick to the consumption model outlined in Section 2, all model calibration takes place in the file bufferstock_batch.m.[5] The parameters that the user needs to input can be broadly split into two categories — model fundamentals and simulation parameters. Model fundamentals, such as δ, R, $yearsAlive$, etc., are part of the analytical consumption model described in Section 2. Simulation parameters, discussed in much more detail shortly, govern how the analytical consumption model is simulated. For example, the variable `<xmax>` determines the maximum value of cash on hand x for which the policy function is calculated (`<xmax>` is purely a coding requirement — in the analytical model of Section 2 there is no upper bound on assets). In general there is a tradeoff between simulation accuracy and speed. Simulation parameters govern this tradeoff.

Model fundamentals were discussed in Section 2, and should be calibrated to fit the needs of the user. Much more discussion, however, is needed for the simulation parameters labeled `<xjump>` and `<xmax>` . The key difference between the analytical model of Section 2 and the simulated Matlab model is that assets x are continuous in the analytical model, while x must be discretized in the simulation. What does this mean? In the analytical model we allow x_t to be *any* number in the interval $[L, \infty)$. In the Matlab simulation, we force assets x_t to lie on the discrete grid $[L,\ L + xjump,\ L + 2(xjump),\ ...,\ xmax]$.[6] The discrete set $\{L,\ L+xjump,\ L+2(xjump),\ ...,\ xmax\}$ will be referred to as the asset grid, or simply *the grid*. This process of creating a discrete grid on which assets must lie is known as *discretizing the state space*. The variable `<xjump>` measures the step size of the discrete grid, and the variable `<xmax>` controls the maximum size of the asset grid. Variable `<borrowingLimit>` (referred to as L above) controls the minimum value of the asset grid (though I think of `<borrowingLimit>` as a model fundamental instead of a simulation parameter). Finally, though likely unclear at this point, the reason for discretizing the state space will become apparent when I discuss the algorithm for calculating consumption and policy functions (Step 2).

3.2 Step 2: Policy and Value Function Calculation

→ *Relevant Matlab file for Step 2 is backwardInduction_general.m*

[5] Nonetheless, the code is written to be extended by users looking to add more functionality.

[6] For example, in Matlab the code `[0:1:10]` produces the vector [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

3.2.1 High-Level Description of Algorithm

Now I will discuss the algorithm for numerically calculating policy and value functions for the model calibrated in Step 1. Recall that there are two state variables — period t and assets x. Both of these state variables are discrete. Period t is in the set $\{1, 2, ..., yearsAlive\}$ and assets x are in the set $\{L,\ L + xjump,\ L + 2(xjump),\ ...,\ xmax\}$ (recall this is called the grid). There are $yearsAlive$ number of time periods, and denote the size of the grid by `<xlen>`. Since there are `<yearsAlive>` possible values for state variable t, and `<xlen>` possible values for state variable x, in total there are `yearsAlive` $\times$ `xlen` total possible state variable combinations.[7]

The algorithm used in this Matlab code calculates optimal behavior at *every* possible state variable combination (x, t). We need a place to start this calculation, and this code starts in the final period of life ($t = yearsAlive$). The reason to start at the end of life is as follows. The Bellman equation (2) makes clear that in order to calculate behavior in period t we need to know the value function in period $t + 1$. If I were to start the calculation at the start of life ($t = 1$), I could not calculate optimal behavior without knowing the value function in $t = 2$. Trying to calculate the value function in period $t = 2$ leads to the same problem; I need to know the value function in $t = 3$. On the other hand, in Section 2.3 I *assumed* the value of $V(x, t)$ for all $t > yearsAlive$. Thus, I've seeded the model with an assumed value function for all $t > yearsAlive$, and I can use this assumption to kick off the backward induction.

Let's start in period $t = yearsAlive$. In this period, there are `<xlen>` total points at which we need to calculate the value function $V(x, yearsAlive)$ and the consumption function $c(x, yearsAlive)$. The Bellman equation is $V(x, t) = \max_c u(c) + \delta E_t[V(R(x-c)+y_{t+1}, t+1)]$. Since we assumed that for all $t > yearsAlive$,

$$V(x,t) = \begin{cases} -\infty & \text{if } x < 0 \\ 0 & \text{if } x \geq 0 \end{cases},$$

[7] For those really interested in numerical methods, a brief discussion of "The Curse of Dimensionality" is warranted here. The algorithm I use calculates optimal behavior at *every* possible combination of state variables. In the model discussed here, this means our Matlab code must calculate optimal behavior at `yearsAlive` $\times$ `xlen` number of points. Let's say this consumption model seems too simple, so you want to add illiquid assets (e.g., 401(k) savings) to the model, denoted by z. Illiquid asset z will need to be discretized into the set $\{0,\ zjump,\ 2zjump,\ ..., zmax\}$ which has length `zlen`. Now, rather than calculating optimal behavior at `yearsAlive`$\times$`xlen` points, the code must calculate optimal behavior at `yearsAlive`$\times$`xlen`$\times$`zlen` points. This could go on even further — say you want to add housing wealth, adding another `hlen` points to the grid. Now, optimal behavior must be calculated at `yearsAlive` $\times$ `xlen` $\times$ `zlen` $\times$ `hlen` points. In general, as we add more state variables the computational complexity of the problem grows exponentially.

the Bellman in period $t = yearsAlive$ reduces to $V(x, yearsAlive) = \max_c u(c)$ such that $c \leq x$. Maximization here isn't complicated — for any level of assets x the optimal consumption decision is for the agent to consume all remaining assets and earn a utility flow $u(x)$. Thus, we now have a policy function in period $t = yearsAlive$ of $c(x, yearsAlive) = x$. This simple policy function gives us a value function in period $t = yearsAlive$ of $V(x, yearsAlive) = u(x)$. As desired, in the period $t = yearsAlive$ we've now calculated the consumption policy function and the value function at all possible levels of x on the grid!

The next step in the algorithm is to use $V(x, yearsAlive)$, which we just solved for, in order to calculate $E_t[V(R(x-c) + y_{t+1}, t+1)]$ for $t = yearsAlive - 1$. In this simulation the agent is assumed to know the income process. Therefore the agent is able to calculate their expected continuation payoff $E_t[V(R(x-c) + y_{t+1}, t+1)]$ for any level of savings $x - c$ in period t.

Now that we've calculated optimal behavior in period $t = yearsAlive$ and used this to calculate $E_t[V(R(x-c) + y_{t+1}, t+1)]$ for $t = yearsAlive - 1$, the code moves backward one period in time to period $t = yearsAlive - 1$. The Bellman equation is now $V(x, t) = \max_c u(c) + \delta E_t[V(R(x-c) + y_{t+1}, t+1)]$. The benefit of backward induction is that we've already calculated $E_t[V(R(x-c) + y_{t+1}, t+1)]$ for $t = yearsAlive - 1$. Thus, for each level of assets x on the grid we simply need to find the value of consumption c to maximize $u(c) + \delta E_t[V(R(x-c) + y_{t+1}, t+1)]$.[8] Just as before, we've now calculated the consumption policy function $c(x, yearsAlive - 1)$ and the value function $V(x, yearsAlive - 1)$.

Next, the Matlab code uses $V(x, yearsAlive - 1)$ in order to calculate $E_t[V(R(x-c) + y_{t+1}, t+1)]$ for $t = yearsAlive - 2$. Again, the code then moves back one period in time to $t = yearsAlive - 2$. The exact process described above is then repeated, moving backward in time through periods $t = yearsAlive - 2, yearsAlive - 3, ..., 1$. After this process is complete, the code will have calculated a consumption policy function and a value function at each possible (x, t) pair. Thus, the code is now able to fully describe the behavior of an agent at any possible point in the state space (x, t). This is exactly what is needed for the simulation of individual consumption decisions, which is Step 3 of the code. Before describing Step 3, however, I will spend more time describing the Matlab coding aspects of this backward induction algorithm.

Overall, this backward induction can be summarized with the following pseudo-code:

[8] There is one important step that I'm glossing over here, which is that consumption c must be chosen to be a value such that $R(x-c)$ lies on the grid. Thus, for each level of assets x there are only a discrete set of possible consumption values, and the Matlab code picks the best one out of this finite set.

Algorithm 1: Numerical Calculation of Consumption and Value Functions:

input : Calibration from Step 1

output: Consumption policy function and value function at each (x, t)

Set $c(x, yearsAlive) = x$ and $V(x, yearsAlive) = ln(x)$;

Set $t = yearsAlive$;

Calculate $E_{t-1}V(R(x_{t-1} - c_{t-1}) + y_t, yearsAlive)$;

for $t = yearsAlive - 1$ **to** 1 **do**

  for $x \in \{L, L + xjump, L + 2\ xjump, ..., xmax\}$ **do**

    Pick c^* to maximize $ln(c) + \delta E_t[V(R(x - c^*) + y_{t+1}, t + 1)]$;

    Set $c(x, t) = c^*$. Set $V(x, t) = ln(c^*) + \delta E_t[V(R(x - c^*) + y_{t+1}, t + 1)]$;

    Calculate $E_{t-1}V(R(x_{t-1} - c_{t-1} + y_t, t)$;

  end

end

3.2.2 Aside on *Vectorization* in Matlab

The above pseudo-code, included for intuition, does not perfectly describe the actual algorithm in backwardInduction_general.m. The reason for this is a Matlab coding trick known as vectorization. The basic idea is that Matlab is fast with matrix operations and slow with for-loops. The pseudo-code above uses nested for-loops (for-loops over both t and x) while the Matlab code does not actually loop over x. Instead, for each period t I calculate optimal consumption for all values of x *at the same time*. This code is certainly more difficult to read, but it also improves runtime significantly.

My own personal tip is to first write Matlab code using for-loops, since for-loops are fairly intuitive. Then, once the code has been written and de-bugged, go back and re-write the code to take advantage of vectorization. Much more detail on vectorization can be found online for those who are interested.

3.3 Step 3: Simulation of Individual Behavior (Forward Induction)

→ *Relevant Matlab file for Step 3 is simulateDecisions.m*

Now that the code has calculated optimal behavior at every possible combination of state variables, we can use this information to simulate the behavior of 5000 artificial agents. Importantly, while the calculation of value and policy functions made use of backward induction, simulation of individual behavior will use forward induction. Specifically, the code

simulates the behavior of 5000 artificial agents by starting with an assumed level of assets in period $t = 1$, and then using consumption policy function $c(x, t)$ and the dynamic budget constraint (equation (1)) to create a consumption profile for each individual agent.

The first step of this process is to generate an income profile (a simulated income realization in each period t) for each of the 5000 artificial agents. In the case where income is deterministic, all agents will be given the same income realization of either $y_t = 1$ or $y_t = 0$, depending on whether the agent is in working years or retirement years. If income is stochastic then each agent, in each period t during which the agent works, will be given a random income realization in the set $\{0.5, 0.5 + xjump, 0.5 + 2 * xjump, ..., 1.5 - xjump, 1.5\}$. Thus, the realized income profile will potentially be different for each of the 5000 agents.

The code assumes that each agent starts in period $t = 1$ with just a single income realization. Letting i index an individual agent, this means that $x_1^i = y_1^i$ — each agent's asset level in period 1 is equal to her income realization in period 1. For each of the 5000 artificial agents, the code then uses the policy function calculated in Step 2 to determine the consumption choice in period $t = 1$. Moving forward in time, assets for any individual i in period $t = 2$ can be calculated using the dynamic budget constraint (1). Specifically, for any agent i we have that period $t = 2$ assets are given by

$$x_2^i = R(x_1^i - c_1^i) + y_2^i.$$

Now that we know the asset level x_2^i in period $t = 2$ for each individual i, we can again use the consumption policy function to calculate the consumption choice of each i in period two. Using these consumption choices, the dynamic budget constraint (1) can again be combined with simulated income y_3^i to give period $t = 3$ assets x_3^i. This process continues forward in time until the final period $t = yearsAlive$. Upon termination, the code will have simulated the consumption decisions of each of the 5000 agents in each period t.[9]

[9] If income is deterministic then each agent will make exactly the same choices in every period. In this case, there is no need to simulate the behavior of 5000 agents. However, in the case where income is stochastic each of the 5000 agents will be subject to a potentially different path of income realizations, and thus will make different decisions over the lifecycle. This is known as $ex - post$ heterogeneity. Specifically, each agent in the simulation has exactly the same preferences and will behave in exactly the same way at a given combination of state variables (x, t). However, stochastic income realizations mean that even though agents in the simulation are $ex - ante$ identical, there is heterogeneity $ex - post$ in each agent's individual time-path of assets x_t and consumption c_t.

3.3.1 Numerical Methods Aside on Simulation Step

The purpose of the simulation step is to generate a distribution of possible outcomes given the stochastic income process and the calculated policy function. This method of generating a distribution of outcomes by simulating consumption profiles for a large number of agents is quite common in the seminal heterogeneous agent literature (e.g., Aiyagari (1994), Krusell and Smith (1998)). However, I believe it is no longer accepted as the most accurate numerical method. Instead, the histogram method of Young (2010) should be more accurate for this consumption model. Coding the methodology of Young (2010) into the simulateDecisions.m file is something for future work.

4 Explanation of Matlab Code Files

I will refer to the file bufferstock_batch.m as the *batch file* from now on. The batch file has two main purposes — (i) it is the file in which the user enters their desired Model Calibration (Step 1); and (ii) it is a wrapper function that oversees Steps 2 and 3 of the simulation.[10] For those interested in using the Matlab code as-is, the batch file is the only file that will need to be edited.

4.0.1 Calibration

The batch file starts with 6 pre-specified cases of the general consumption model. For users interested in calibrating the model themselves, the variable `<step>` should be set to 0. Within the case 0 block, the variable `<yearsWork>` governs the number of years in which the agent earns an income and the variable `<yearsRet>` governs the number of years the agent is retired. The code automatically sets variable `<yearsAlive>` equal to `yearsWork` + `yearsRet`. When variable `<incomeCase>` equals 1 the agent earns a deterministic income of $y_t = 1$ in each working year. When `<incomeCase>` equals 2 the agent earns a stochastic income of $y_t \sim Uniform[0.5, 1.5]$ in each working year. `<delta>` is the discount factor. `<beta>` and `<beta_hat>` are used for simulating quasi-hyperbolic discounting. `<borrowingLimit>` is the lower bound on the grid, and `<xmax>` is the upper bound on the grid. Unless the user desires explicitly setting a binding lower bound on assets, I would recommend leaving `<borrowingLimit>` commented out. The Matlab code is built to default into setting `<borrowingLimit>` to a level at which it will not bind (i.e., just below the natural borrowing limit). The user should also feel free to change the interest variable `<R>` (set by default

[10] Though the batch file does not simulate the consumption model itself, the bath file calls the functions responsible for the simulation.

to 1.05), which is located further down in the batch file.

With slightly more caution, variables <eolRepay> and <xjump> can also be changed. By default the variable <eolRepay> equals 1, and this ensures that the agent dies with non-negative assets by seeding the simulation with the value function

$$V(x,t) = \begin{cases} -\infty & \text{if } x < 0 \\ 0 & \text{if } x \geq 0 \end{cases} \text{ for all } t > yearsAlive.$$

Alternatively, setting <eolRepay> equal to 0 seeds the simulation with the simpler value function $V(x,t) = 0$ for all $t > yearsAlive$.[11]

As discussed above, variable <xjump> governs the density of the grid. Picking the value of <xjump> involves a tradeoff between accuracy and speed. A larger value of <xjump> creates a sparser grid, and this means there are fewer points at which the Matlab code needs to calculate policy and value functions (in Step 2). However, the smaller the value of <xjump> the closer the discrete grid comes to replicating a continuous asset space.[12]

4.0.2 Wrapper Function for Steps 2 and 3

The second purpose of the batch file is to set up the Matlab workspace for Steps 2 and 3, and then call the functions responsible for carrying out these steps.

The set-up for Step 2 begins with the creation of structs, which are a type of Matlab storage structure. This simulation spans across multiple Matlab files, and structs are a useful way of passing variables across functions.

The batch file calls the functions buildIncome.m and buildX_.m . These are fairly short files — buildIncome calibrates the income process and buildX_ creates the grid. These two functions could have been written directly into the batch file. However, I separated them out into functions in order to keep the batch file a bit cleaner.

Once the model is fully calibrated, the batch file calls the function backwardInduction_general.m. This is the function which calculates value and policy functions at each state combination (x,t) (Step 2). Once this function terminates, the batch file then calls

[11] From the standpoint of a consumption model it typically makes sense to impose a No-Ponzi condition, which is why <eolRepay> equals 1 by default. However, for 2010c students interested in seeing iterative solutions to Bellman equations it may be more intuitive to begin backward induction from the value function $V(x,t) = 0$.

[12] An interesting exercise here is to plot <V__(:, 1)> as a function of <xjump>. Recall that state space discretization limits consumption to a discrete set of values. Thus, a large value for <xjump> means that the agent's consumption choice set is quite small. As <xjump> shrinks the agent in the discretized environment becomes better able to replicate her optimal consumption level.

the function simulateDecisions.m. SimulateDecisions is the function that simulates individual behavior for 5000 heterogeneous agents in the model (Step 3). Both of these files are (hopefully) well commented, and follow the algorithm described in Sections 3.2 and 3.3. For this reason I will not expand on them further in this document.

Finally, the batch file ends with sample code for plotting the output of the model. This is just sample code included to serve as an example, and is welcome to be used / edited as desired.

5 Extension: Quasi-hyperbolic Discounting

Thus far I have only focused on exponential discounting, but the Matlab code does also allow for both sophisticated and naive quasi-hyperbolic discounting (e.g., Laibson (1997), O'Donoghue and Rabin (1999) for more on quasi-hyperbolic preferences). I'll outline both in this extension. Note that this extension assumes the reader has some knowledge of sophisticated and naive quasi-hyperbolic discounting.

5.1 Changes to Value Function

Starting with the general consumption model of Section 2, the introduction of quasi-hyperbolic discounting changes the value function of the agent. Specifically, the *sophisticated* present-biased agent's behavior is now described by the set of equations:

$$\begin{aligned} V(x,t) &= u(c(x,t)) + \delta E_t \left[V(R(x-c(x,t)) + y_{t+1}, t+1)\right] \\ W(x,t) &= u(c(x,t)) + \beta\delta E_t \left[V(R(x-c(x,t)) + y_{t+1}, t+1)\right] \\ c(x,t) &= \text{argmax}_c \; u(c) + \beta\delta E_t \left[V(R(x-c) + y_{t+1}, t+1)\right] \end{aligned}$$

Function $V(x,t)$ is the continuation-value function, and accumulates utils exponentially. Function $W(x,t)$ is the current-value function, and accumulates utils quasi-hyperbolically.

The code also allows for naive present-bias. The *naive* present-biased agent's behavior is described by the set of equations:

$$\begin{aligned} V^E(x,t) &= u(c^E(x,t)) + \delta E_t \left[V^E(R(x-c^E(x,t)) + y_{t+1}, t+1)\right] \\ c^E(x,t) &= argmax_c \; u(c) + \beta^E\delta E_t \left[V^E(R(x-c) + y_{t+1}, t+1)\right] \\ c(x,t) &= argmax_c \; u(c) + \beta\delta E_t \left[V^E(R(x-c) + y_{t+1}, t+1)\right] \end{aligned}$$

I use superscript E to denote the false expectation of the naive agent. Thus, β^E is the naive agent's false belief of her present-bias coefficient. Full naivete is modeled by $\beta^E = 1$, in which

case the agent believes herself to be an exponential discounter. Partial native is modeled by $\beta^E \in (\beta, 1)$. The function $c(x,t)$ describes the *actual* consumption decision of the agent. However, c^E describes the *expected* consumption decision of the agent, since the agent is naive about his true present bias coefficient β. For the purposes of backward induction, note that the expected value function V^E is calculated using expected consumption c^E instead of actual consumption c.

When simulating a model with quasi-hyperbolic discounting it is not uncommon to see *consumption pathologies* in the consumption policy function. These pathologies are violations of continuity and monotonicity of the consumption function. For more, see Harris and Laibson (2003).

5.2 Coding Details

The batch file allows for the user to calibrate a model with quasi-hyperbolic discounting through the two variables `<beta>` and `<beta_hat>`. Note that `<beta>` and `<beta_hat>` correspond to β and β^E, respectively. Unless specified otherwise, the default value of `<beta>` is 1 (exponential discounting) and the default value of `<beta_hat>` is whatever value `<beta>` is set to (sophistication). For the user of the code, all that is required to implement quasi-hyperbolic discounting is the setting of variables `<beta>` and `<beta_hat>` in the batch file.

The implementation of quasi-hyperbolic discounting in the file backwardIndinction_general.m follows pretty directly from the formulas outlined in Section 5.1. In the case of sophisticated quasi-hyperbolic discounting, the only change from the algorithm described in Section 3.2 is that the utility earned in all future periods must be discounted by factor β when calculating the optimal level of consumption. Naive quasi-hyperbolic discounting is slightly more complicated, as the code needs to calculate both the *realized* consumption policy function and the *expected* consumption policy function.[13]

References

Aiyagari, S Rao, "Uninsured idiosyncratic risk and aggregate saving," *The Quarterly Journal of Economics*, 1994, *109* (3), 659–684.

Harris, Christopher and David Laibson, "Hyberbolic Discounting and Consumption," *Econometric Society Monographs*, 2003, *35*, 258–297.

[13]Function backwardIndinction_general only returns the realized consumption policy function. However, the expected consumption policy function also needs to be calculated in order to generate the expected value function V^E.

Krusell, Per and Anthony A Smith Jr, "Income and wealth heterogeneity in the macroeconomy," *Journal of political Economy*, 1998, *106* (5), 867–896.

Laibson, David, "Golden Eggs and Hyperbolic Discounting," *Quarterly Journal of Economics*, 1997, *62*, 443–479.

O'Donoghue, Ted and Matthew Rabin, "Doing it Now or Later," *American Economic Review*, 1999, *89*, 103–124.

Young, Eric R, "Solving the incomplete markets model with aggregate uncertainty using the Krusell–Smith algorithm and non-stochastic simulations," *Journal of Economic Dynamics and Control*, 2010, *34* (1), 36–41.