THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ASSET PRICING IN A LUCAS FRAMEWORK WITH BOUNDEDLY RATIONAL, HETEROGENEOUS AGENTS

By

ANDREW J. CULHAM

A Dissertation submitted to the
Department of Mathematics
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2007

The members of the Committee approve the Dissertation of Andrew J. Culham defended on June 29, 2007.

Paul M. Beaumont
Professor Co-Directing Dissertation

Alec N. Kercheval
Professor Co-Directing Dissertation

Don Schlagenhauf
Outside Committee Member

Yevgeny Goncharov
Committee Member

David Kopriva
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

I dedicate this dissertation to my parents, whose love and support made my success possible.

ACKNOWLEDGEMENTS

It is difficult to overstate my gratitude to my dissertation advisors, Dr. Paul Beaumont and Dr. Alec Kercheval. Most doctoral students consider themselves lucky to have an academic advisor who possesses the intelligence and patience necessary to guide them to the completion of their degree. I was extremely lucky to have two advisors who possess these qualities. In the past three years, Paul and Alec have provided me with the inspiration and support I desperately needed to complete this dissertation. I consider myself extremely fortunate to know these gentlemen and to have had the opportunity to work alongside them for the past three years.

I would also like to thank my committee members, Dr. Don Schlagenhauf, Dr. Yevgeny Goncharov and Dr. David Kopriva for taking the time out of their extremely busy schedules to assist me in any way I needed. In particular, I owe a special thanks to Dr. Kopriva for many discussions on numerical methods. I would also like to thank Dr. Bettye-Anne Case who initially convinced me to stay here at FSU and pursue the doctoral degree. She provided excellent advice and I was lucky to have her as my advisor for the first two years I was here.

Last, and certainly not least, I would like to thank my family and friends for their continued support all these years. This has been a long trip and it would have been much longer had I not had these wonderful people supporting me. My family, although far away, have always stood by me and have provided me with endless love and encouragement. I also owe my good friend Mike Hansen thanks for always reminding me that I made the right choice in pursuing this degree. Finally, I have made many dear friends during my time in Tallahassee. In particular, I am forever indebted to Rebecca Clements, Shivakumar "Shibi" Rajagopolan and Clayton Webster for believing in me and helping me have alot of fun when I needed it the most. I hope we are able to stay in touch no matter where we end up.

-Andy

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The standard dynamic general equilibrium model of financial markets does a poor job of explaining the empirical facts observed in real market data. The common assumptions of homogeneous investors and rational expectations equilibrium are thought to be major factors leading to this poor performance. In an attempt to relax these assumptions, the literature has seen the emergence of agent-based computational models where artificial economies are populated with agents who trade in stylized asset markets. Although they offer a great deal of flexibility, the theoretical community has often criticized these agent-based models because the agents are too limited in their analytical abilities.

In this work, we create an artificial market with a single risky asset and populate it with fully optimizing, forward looking, infinitely lived, heterogeneous agents. We restrict the state space of our agents by not allowing them to observe the aggregate distribution of wealth so they are required to compute their conditional demand functions while simultaneously learning the equations of motion for the aggregate state variables. We develop an efficient and flexible model code that can be used to explore a wide number of asset pricing questions while remaining consistent with conventional asset pricing theory. We validate our model and code against known analytical solutions as well as against a new analytical result for agents with differing discount rates.

Our simulation results for general cases without known analytical solutions show that, in general, agents' asset holdings converge to a steady-state distribution and the agents are able to learn the equilibrium prices despite the restricted state space. Further work will be necessary to determine whether the exceptional cases have some fundamental theoretical explanation or can be attributed to numerical issues. We conjecture that convergence to the equilibrium is global and that the market-clearing price acts to guide the agents' forecasts

toward that equilibrium.

CHAPTER 1

INTRODUCTION

One of the most studied models in the financial economics literature over the past thirty years has been the Lucas asset pricing model (Lucas, 1978). In this model there is a stock (or "tree") that produces a random dividend (or "fruit") in each period, and the agents buy and sell shares of the tree in a competitive market. One share entitles the agent to the fruit of one tree in the next period. In this framework, production is both exogenous and stochastic. That is, the dividend is determined randomly and independently from the market. The agents may or may not be assumed to know the distribution of the dividends. The assumption of exogenous production allows us to focus not on the problem of the firm, but solely on the agent's problem, and therefore eliminates one half of a difficult problem.

Despite its popularity, the model does a poor job of explaining even the most basic empirical properties of actual market data (Campbell and Cochrane, 1999; Mehra and Prescott, 1985; Mehra, 2003; LeBaron, 2004). The reason for this poor performance is that, in addition to the assumption of exogenous production, typically agents are also assumed to be identical to have rational expectations. Both of these assumptions are quite restrictive. Although we will not calibrate to data here, this work provides a computational platform that would allow us to study the asset pricing puzzles discussed in the above references.

In an attempt to relax the restrictive assumptions of homogeneous agents and rational expectations, recent research in asset pricing has seen the emergence of agent-based models which focus on computing an equilibrium in models with heterogeneous agents. Agent-based models do not make the same restrictive assumptions as the classical models do, however. Due to the complex nature of the models we are no longer able to compute the equilibria analytically and as a result, we must resort to simulations.

Despite their flexibility, agent-based models have not generally been well accepted by the

theoretical financial economics community. The main reasons for this, are that agents are often too limited in their analytical abilities. Instead of solving an infinite period, forward looking optimization problem, as theory suggests they should, agents repeatedly solve a one period problem. Additional assumptions must also be placed on the utility and forecasts of the agents in order to allow for a closed form for their demand functions.

In this work we create a model that incorporates both the classical and the agent-based approaches. We populate an artificial economy with utility maximizing, infinitely lived, forward looking agents and allow them to trade with one another. Agents have a price forecasting rule that they will use to compute their expectations of future prices. Over time, as more information becomes available, the agents will update their forecasting rule using an adaptive learning scheme. The model is efficient and flexible and provides a platform that is suitable to explore a wide number of asset pricing questions. The main questions addressed here are whether or not the agents are able to learn the equilibrium under various forms of heterogeneity, and if so, what is the equilibrium distribution of holdings (or wealth)?

We find that in all cases considered that the agents are able to learn the equilibrium asset prices. Additionally, in most cases the agents' wealth distribution converges to a steady-state and trading halts. There are cases in which the agents fail to converge to a steady-state wealth distribution and trading will persist forever. However, it is unclear if these exceptional cases have some underlying theoretical explanation, or if they are the product of numerical issues.

The manuscript is organized as follows. The remainder of this chapter will discuss the common assumptions made in asset pricing problems. In particular, we will examine the assumptions of homogeneous agents and rational expectations. We will then provide an example showing the problems associated with relaxing the homogeneous agent assumption. Finally, we will briefly examine the Santa Fe Artificial Stock Market model and explain why it has had little success in satisfying the theoretical community.

In Chapter 2 we carefully construct the model that will be used throughout the manuscript. We prove that a solution exists to the (single) agent's problem and that the solution is unique. In addition, several useful properties of the problem and its solution will be proved for use later. In Chapter 3 we present the numerical methods needed to solve the problem for a single agent. Specifically, we will use value function iteration that is known to be reliable but slow. Using many of the properties proved in Chapter 2 we will be able

to significantly improve upon a basic grid search algorithm. Numerical results are validated using a special case that admits an analytical solution. Chapter 4 presents the numerical methods we require at the market level. Specifically, we discuss market clearing and learning, as well as a market algorithm. Again, numerical results are validated using known solutions. Chapter 5 presents various simulation results. Finally, Chapter 6 has concluding comments and future research directions.

1.1 Common Assumptions

The assumption of homogeneous agents simplifies the asset pricing problem immensely since it implies that agents will never trade. This is clear since a necessary condition for a trade to occur is that agents differ in some way. There are many ways in which agents could differ in order to induce a trade, including, but not limited to, differing wealths, different discounting of future payoffs and different views about future price behavior. With the assumption of a homogeneous agent, however, none of these differences exist. The following simple example illustrates the situation in a homogeneous economy. Suppose that one agent has a particularly optimistic view of the future price of the stock, which makes him wish to purchase shares. Since everyone is identical, all other agents have the same optimistic view and thus, they all wish to buy as well. The result is an excess demand and a higher price. This will continue until everyone is happy holding the shares they currently have. Note that the price will change in this example, even though there is no trading.

It is often the case that we will assume a single (or "representative") agent. This case is the same as the case of many identical agents just discussed. The reason to assume a single agent instead of many is that since all agents are identical, they will behave as if they were a single agent. Assuming a representative agent allows us to not have to keep track of agents. It is, however, quite confusing to think of an economy in which there is only a single agent. One is tempted to believe that the agent does not trade simply because he is alone in the world. While it is certainly true that there must be more than one agent in order for a trade to occur, this is not the intent of the representative agent assumption. To avoid the confusion, one should keep in mind that a representative agent is equivalent to many identical agents.

We now turn to the assumption of *rational expectations*. In each period, the agents are forced to form expectations about future prices. Given these expectations and a dividend,

a market clearing price will be determined. Agents are said to have rational expectations if, given a dividend, the agent could correctly determine the current market clearing price. To put it another way, suppose that agents have some function (depending on the current states of the world) that they use to forecast future prices. There is also an underlying "market clearing function" that determines the current period price, given the current states. The rational expectations assumption says that these two functions are the same. In other words, the agents are able to correctly determine the aggregate states.

In Chapter 4, we will show a special case when agents, with sufficient information about the market, will be able to compute the true market clearing price function. Specifically, we will consider the case when all agents are identical, and they know this fact. Identical agents who do not know they are identical will solve a much different problem than if they knew they were identical. The key difference is that if agents know they are identical, then they also know that they will never trade in any period. Thus, since they will never have the opportunity to hold more (or fewer) shares than they start with, they have no need to make a decision about how many shares to hold for subsequent periods. In other words, they do not need to know their demand function. On the other hand, agents who do not have the knowledge that they are identical must be prepared to trade at all times. Of course, it will turn out that they never trade, but since they so not know this ex ante they must compute their demand function in each period. The difference between agents who are identical and know it and those who don't is subtle, but it will turn out to be of major importance.

The rational expectations assumption is very strong. It will often be the case that, as modelers, we do not know the true underlying market clearing price function. Therefore, in making the rational expectations assumption, we have created an economy that is populated with agents who are assumed to be smarter than we are. In a sense, these agents are better than simply being rational, they are "super-rational". While the term super-rational has been used occasionally in the literature, we will use the convention that these agents are rational or *fully rational*. For a collection of work examining the rational expectations assumption see Frydman and Phelps (1983).

It should be noted that agents who are not rational, in the sense used here, should not be thought of as irrational. It is not the case that non-rational agents are acting sub-optimally in some way, but rather that they do not possess the information needed to be (fully) rational. Agents for whom we do not assume rational expectations will be referred to as *boundedly*

rational (Simon, 1957).

We conclude this section with a quote from Sargent (1993) that supports our desire to construct a non-equilibrium model or asset prices.

> "Rational expectations is an equilibrium concept that at best describes how such a system might eventually behave if the system will ever settle down to a situation in which all of the agents have solved their 'scientific problems'." (Sargent (1993), page 23)

1.2 A Heterogeneous Agent Model

Suppose now that we wish to move away from the very restrictive rational expectations framework. For now, let us assume that agents differ only in how they compute expectations. (This is still a stronger assumption than we would like, but it will suffice to make our point.) Suppose there are N agents and each agent forecasts the future price in their own way. Denote agent i's period t expectation of the dividend and price at time $t+1$ by $E_t^i[d_{t+1}]$ and $E_t^i[p_{t+1}]$, respectively. Without going into technical details, let us consider an example given in Arthur et al. (1997) where the market clearing price in period t is determined by[1]

$$p_t = \beta \sum_{j=1}^{N} w_{j,t} \left(E_t^j[d_{t+1}] + E_t^j[p_{t+1}] \right), \tag{1.1}$$

where β is a discount factor and $w_{j,t}$ is the weight given to agent j's forecast in period t. Under the assumption of homogeneous agents we are able to drop the j subscript and replace all weights by $1/N$ in (1.1). In the case of heterogeneous agents, however, we may not do so.

Now, suppose that agents know (1.1). Then taking expectations, agent i will forecast the next period's price as

$$E_t^i[p_{t+1}] = \beta E_t^i \left[\sum_{j=1}^{N} \left\{ w_{j,t+1} \left(E_t^j[d_{t+2}] + E_t^j[p_{t+2}] \right) \right\} \right]. \tag{1.2}$$

This requires that agent i, in forming his expectation of price, take into account his expectations of all other agents' expectations of dividend and price, two periods ahead.

[1] We will formally introduce our model in Chapter 2. This result is proposed here purely to illustrate the difficulties that arise when we incorporate heterogeneous agents. The interested reader is referred to Arthur et al. (1997) for more details on the present example.

We could, in a similar way, replace $E_t^j[p_{t+2}]$ with terms involving variables at time $t+3$. But doing so would require that agents take into account all other agents expectations of all other agents expectations. In addition, agent i's expectation is influenced by what others expect him to do, which depends upon what he expects others to do, and so on. The result is an infinite recursion that will lead us nowhere.

Note that in the above argument we have in no way limited the agents with respect to their knowledge or computational ability. The problem, posed in this way, does not have a well defined solution. As Arthur et al. state, "Infinitely intelligent agents cannot form expectations in a determinate way", (Arthur et al., 1997, page 6).

The desire to study markets populated with heterogeneous agents coupled with the lack of analytical tractability led to the emergence of Agent-Based Computational Economics (ACE) models, beginning in the early 1990s. As the name suggests, ACE models attempt to compute an equilibrium numerically rather than analytically. Typically, large numbers of agents populate an artificial, computer-based market. Agents interact with each other, via trading, using rules that are updated over time. For a survey of work done in ACE modeling, see LeBaron (2006).

1.3 The Santa Fe Institute's Artificial Stock Market

One of the original, and best known ACE models is the Santa Fe Institute's Artificial Stock Market, or SFI-ASM.[2] The Santa Fe Institute is a collection of economists, physicists and computer and life scientists who work jointly on (among other things) numerically simulating complex economic models. Below we will briefly describe the Santa Fe model since it provided the initial motivation for the model presented in this manuscript. For further details on the construction of the SFI-ASM see Arthur et al. (1997), LeBaron (2002) and Ehrentreich (2004).

In the SFI-ASM there are two securities, a risky stock and a risk free bond. The bond pays a constant interest rate, r, and is in infinite supply, while the stock pays stochastic dividends from a distribution unknown to the agents. There is a total supply of N shares and the market is populated with N heterogeneous, myopic agents. The term myopic means that instead of solving an infinite horizon problem, as in Lucas (1978), the agents solve only

[2] See Chiarella et al. (1998) for a subsequent approach.

a one period problem. In addition, agents use a constant absolute risk aversion (CARA) utility function of the form $u(c) = -\exp(-\lambda c)$, where $\lambda > 0$ is the level of absolute risk aversion. Assume that agent i's prediction at time t of the next period's price plus dividend is normally distributed with (conditional) mean and variance, $E_t^i[p_{t+1}+d_{t+1}]$ and $\sigma_{t,i}^2$. Arthur et al. (1997) note that under assumptions of Gaussian distributions for forecasts and CARA utility, it can be shown that agent i will demand x_t^i shares of the stock at time t, where

$$x_t^i = \frac{E_t^i(p_{t+1} + d_{t+1}) - p_t(1+r)}{\lambda \sigma_{i,t}^2}. \tag{1.3}$$

The market price of the stock, p_t, is determined by equating total demand with total supply.

The key feature of the SFI-ASM is how agents forecast the future price and dividend. At each time, we assume that the time series of current and past prices can be summarized by a set of J market descriptors. A descriptor could be, for example, "price has risen the past three periods", or, "the last price was larger than the 10 day moving average". These descriptors are available to all agents.

Agents have in their possession, M market predictors. A predictor is a condition/forecast pair that dictates if a certain combination of the J market descriptors are true or false, then a certain forecast of the price plus dividend is to be used. A forecast of the future price plus dividend is a linear combination of the current price plus dividend. For example, suppose there are four market descriptors (i.e. $J = 4$). These are

1. The price is greater than the 5 period moving average
2. The price is greater than the 10 period moving average
3. The current price-dividend ratio is greater than 10
4. The current price-dividend ratio is greater than 15

The agents' predictors are strings of zeros, ones and number signs, of length 4, followed by a vector of two real numbers. A zero indicates that the descriptor should be false, a one indicates that the descriptor should be true, and a number sign indicates that we do not care about that descriptor. The real numbers are the forecasting parameters. For example, the predictor $(1\#\#\#)/(0.9, 1)$ would mean if price is higher than the 5 period moving average, regardless of all other conditions, use the forecast $E_t^i[p_{t+1} + d_{t+1}] = 0.9(p_t + d_t) + 1$.

Clearly more than one predictor can be active at one time. To forecast the price plus dividend in this case, the agent will use a linear combination of the H best predictors that are active for that period. In order to decide which H predictors to use, the agent must keep a history of how his predictors have done. In each time period, after the price is determined, the agent will check *all* of his predictors (active or not) to see which ones would have worked best, had they been used. See Appendix A in Arthur et al. (1997) for more details on how the agent assigns a score to his predictors.

The preceding discussion explains one way in which the agents in the SFI-ASM learn. Since the agents are constantly updating the accuracy of their predictors, they are constantly learning. But learning will also take place on a larger time scale. Every so often (perhaps 250 periods), the agents will discard some of their worst predictors in favor of new ones. After all, why should the agent continue to waste time checking rules that he knows are rarely, if ever, useful? To generate new predictors the agents use a genetic algorithm. Rules are created either by mutation, when one rule is changed in some way, or by crossover, when two existing rules are combined in some way to make a new one.

The results of running the model, under various parameter configurations, are that agents who do not learn "too quickly" will converge to the rational expectations solution. Since agents learn every period which of their current rules work the best, the speed of learning is controlled by how often the genetic algorithm is invoked. See Arthur et al. (1997) and LeBaron et al. (1999) for a further discussion of the results obtained from this artificial stock market.

As pointed out in LeBaron (2002), there are several weaknesses associated with the approach taken by the SFI-ASM. First, the agents are using a linear forecasting rule to predict future prices and dividends. It is not clear how much this affects the outcomes. A second problem is the assumption of CARA utility. This was a convenient choice in order to get a closed form for the agent's demand for shares of the risky asset, but note that nowhere in (1.3) does the agent's wealth enter. Therefore, agents with a large wealth have no more impact on prices than agents with a smaller wealth. We would certainly like to have wealthier agents weighted heavier in the market. A move to constant relative risk aversion (CRRA) utility would allow wealthier traders to have more impact. In addition, recall that agents were assumed to be myopic. Traditional economic theory has agents solve infinite horizon problems. On this issue of intertemporal preferences, LeBaron says, "In general this would

involve some potentially complicated issues in terms of the interactions between portfolio and consumption choices", (LeBaron (2002), page 6).

As we stated earlier, the SFI-ASM was the initial motivation for the model we will study throughout this manuscript. Traditional economic theory has generally been presented with the frame of mind that we are not interested to actually simulate markets, but to rather make assumptions to simplify the problem to the point where we are able to obtain analytical solutions. The SFI-ASM has, in a sense, taken a completely opposite approach to solve the same problem by focusing solely on computing an equilibrium. In doing so, the SFI-ASM essentially ignores decades of economic theory (by using CARA utility, for example) to create a model in which agents are truly heterogeneous and may interact with one another in an artificial market environment. Our goal is to create a model that is an improvement on both of these extremes.

Like the SFI-ASM, we have created an artificial stock market populated with heterogeneous agents. The market consists of a single risky asset, paying stochastic per period dividends. There is no bond in our model, however, agents will receive a constant per period endowment. It can be shown that our model corresponds to one with a single stock and a bond that pays a constant interest rate. In contrast to the SFI-ASM (which assumes myopic agents), our agents will be forward looking, CRRA utility agents. We will assume that agents know the distribution of dividends, but they do not know the market clearing price function. Agents are assumed to know only public, aggregate variables (i.e. dividends and market clearing prices) and they know nothing about the other agents.

Our agents will not use a predictor method to learn about the market. Instead, agents will be given a forecasting rule, which depends on the current states of the world, that they use to forecast future prices. These forecasts will, in turn, be incorporated into the market clearing price. Agents will observe the prices that occur in the market and update their forecasting rule over time using a recursive updating scheme.[3] The key question we wish to address is whether or not the agents can learn the equilibrium market clearing prices. The question of convergence of heterogeneous agents has been addressed previously. The Marimon-Sargent hypothesis (Marimon et al., 1990) is that agents are able to learn the market clearing price function over time.

Note that, in general, the market clearing price function will depend on all aggregate

[3]For a very broad survey of available learning mechanisms, see Tesfatsion (2003).

variables. These variables include the dividend, the total supply of shares, the distribution of wealth and the distribution of all other variables (e.g. discount factors, endowments). Since agents will not observe most of these variables (in general), we cannot expect them to be able to use them in the learning process. Instead, we hypothesize that agents will be able to correctly forecast prices using a function of dividends alone. This hypothesis will be discussed further in Chapter 5.

CHAPTER 2

THE LUCAS ASSET PRICING MODEL

This chapter provides a careful setup and rigorous treatment of the version of the Lucas asset pricing model (Lucas, 1978) that will be used in this manuscript. Unlike the traditional approach used to solve asset pricing models (see, for example, Judd (1998), Section 17.1), we will not make equilibrium assumptions early. Every effort will be taken to present the results in the most general setting possible. Generality is crucial to our study of out-of-equilibrium price behavior and convergence to the Rational Expectations Equilibrium (REE). The majority of this chapter follows Stokey and Lucas (1989) and Harris (1987).

2.1 The Model

We begin with a description of the model. Suppose there are N agents and a single risky asset (stock) paying stochastic dividends according to a distribution the agents are assumed to know. The agents are endowed at time 0 with initial stock holdings s_0^i. At time 0, agent i solves the following problem:

$$\max_{\{c_t^i, s_{t+1}^i\}_{t=0}^{\infty}} E_0 \left\{ \sum_{t=0}^{\infty} \beta_i^t u^i(c_t^i) \right\} \tag{2.1}$$

subject to

$$c_t^i + p_t s_{t+1}^i \leq s_t^i(p_t + d_t) + e_t^i \quad \forall t$$

$$c_t^i \geq 0, \forall t, \quad s_0^i \text{ given},$$

where β_i is agent i's discount factor, c_t^i and s_t^i are agent i's period t consumption and share holdings, respectively, d_t is the stochastic dividend for period t, p_t is the price of the stock and e_t^i is the period t endowment of agent i.

Our first goal in this chapter will be to show that (2.1) is equivalent to a two period problem. Before we are able to do this some discussion of timing will be necessary.

In the model presented in (2.1), time is discrete. Thus, we should not think of a time t as an *instant*, but rather as a *period*. It is convenient to think of a time period being one day, although it could be any increment of time we wish. The most natural timing in a model such as this is as follows. At the beginning of period 0, the agent awakes with s_0^i shares of stock and his endowment, e_0^i. Immediately after waking up he learns what the dividend for that day will be. Some time later the market clearing price is declared.[1] The agent's task is to solve (2.1). A solution to the problem is a sequence of consumptions and asset holdings, $\{c_t, s_{t+1}\}$ for $t = 0, 1, 2, \ldots$, that maximizes the infinite sum of discounted expected utility. Corresponding to such a sequence is a pair of functions, $g_s, g_c : \mathbb{R} \times \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$, such that $s_{t+1} = g_s(s_t, d_t, p_t)$ and $c_t = g_c(s_t, d_t, p_t)$, for all t. Solving for the sequences can equivalently be thought of as solving for these functions of the state variables.

It should be noted that the agent can solve for his demand functions, g_c and g_s, before the game actually commences. Once he learns his initial state he will be able to form a demand. However, he will use the same demand functions to form that demand regardless of what the initial state happens to be. Also, in the above discussion we have specified the demand functions to be stationary (i.e. with no explicit dependence on time). This is justified since time only enters (2.1) through the geometric discounting. Whether the agent solved for his demand functions today or a year from now, he would solve the same infinite problem.

2.2 The Formal Setup

We now step back and more carefully specify the setup of the model. We assume the following list of assumptions holds throughout the manuscript, unless otherwise noted.

A1. There are N (possibly) heterogeneous agents.

A2. Agents have Constant Relative Risk Aversion (CRRA) utility, $u^i(c) = \dfrac{c^{1-\gamma_i} - 1}{1 - \gamma_i}$, where $\gamma_i \geq 0$ is the level of risk aversion for agent i.[2] Clearly u^i is twice differentiable on $(0, \infty)$, for any γ_i, with $(u^i)' > 0$ and $(u^i)'' < 0$ for any $\gamma_i > 0$.

[1] The mechanics of the market will be discussed at length in Chapter 4. At that point it will become clear why we distinguish between the time the dividend is called and the time the price is determined.

[2] Note that letting $\gamma_i \to 1$ gives $u^i(c) = \ln c$.

A3. There are N shares of the risky asset available and short sales are not allowed.[3]

A4. Dividends are drawn independently and identically (iid) from a distribution with compact support. Agents know the true distribution of dividends.

A5. The agents' endowments are constant. That is, for each i, $e^i_t = e^i$ for all t.

A6. Agents have a discount factor $0 < \beta_i < 1$.

A7. Agents have a price forecasting function, $\tilde{p}^i$. This function is assumed to be a polynomial in d_t and should be strictly increasing.

A8. Agents must consume at least $c_\varepsilon > 0$ each period. That is, $c^i_t \geq c_\varepsilon$ for all t and all i. This constant should be thought of as the minimum the agents require to sustain life. Consuming less than this will result in death, the least desirable outcome.

A9. Prices are strictly positive and finite for all t.

Note that we have not assumed a distribution for the price process as is common in the finance literature. Instead we have given the agents a way to forecast future prices as functions of the random dividend. The price at time zero is known to the agents and they use their forecasting functions to form their best guess for all future prices. A forecasting function together with the known distribution of dividends defines a hypothesized distribution on the values of (d_t, p_t) for $t \geq 1$. We will denote the joint cumulative distribution for agent i as $q^i(d, p)$. Using the same argument as we did to justify the stationarity of the policy functions, and the fact that dividends are iid, it makes sense to have a time independent distribution function here.

Note that, in general, agents will have different forecasting functions, and thus they will disagree about the future prices of the stock. This will induce them to trade with one another. Before we can discuss trading we need to carefully describe how each agent will solve (2.1). That is the focus of the remainder of this chapter. We will suppress the superscript i whenever it is not confusing but it should be understood that the following applies to each agent. Assume for the remainder of the chapter that the initial state, (s_0, d_0, p_0), is known.

[3]Formally we can allow short sales, but to prevent Ponzi schemes, where the agent can accumulate infinite wealth by repeatedly selling the asset, we would need to limit the agent's ability sell the stock short. All of the analysis in this chapter holds if we impose $s_t \geq -M$ for some $M \geq 0$, so without loss of generality, we assume the lower bound on holdings is 0.

Proposition 2.1. *The budget constraint $c_t + p_t s_{t+1} \leq s_t(p_t + d_t) + e$ holds with equality for all t.*

Proof. Since the utility function is strictly increasing the agent would always prefer to consume more than not. Therefore he will never choose any action except to either invest or consume his entire wealth each period. Doing so would result in a lower utility and thus would not be optimal. □

Knowing that the budget constraint will hold with equality means that instead of solving for both $\{c_t\}$ and $\{s_{t+1}\}$, we need only solve for one. If we know s_{t+1} then we can use the budget constraint to find c_t, for all t.

Proposition 2.2. *From the agent's point of view, utility will be bounded for all t. In addition, marginal utility (or u') is bounded for all t also.*

Proof. Since u is continuous on $(0, \infty)$, it will suffice to show that there exists $\underline{c}$ and $\overline{c}$ such that $c_t \in [\underline{c}, \overline{c}]$, for all t. By assumption we have $c_t \geq c_\varepsilon$. Thus it remains to find an upper bound. Note that since short sales are not allowed we have $s_t \in [0, N]$, for all t. Let $\underline{d}$ and $\overline{d}$ denote the lower and upper limits for the dividend, respectively.

At time 0, the price and dividend are given. Thus we have $c_0 = s_0(p_0 + d_0) + e - p_0 s_1 \leq N(p_0 + d_0) + e$. For $t \geq 1$ the agent uses his forecasting rule to predict the prices. Since $\tilde{p}$ is a polynomial in d and d is bounded, the agent's predicted future prices are bounded also. Thus we have, for $t \geq 1$,

$$\begin{aligned} c_t &= s_t(p_t + d_t) + e - p_t s_{t+1} \\ &= s_t(\tilde{p}(d_t) + d_t) + e - p_t s_{t+1} \\ &\leq N(\tilde{p}(\overline{d}) + \overline{d}) + e. \end{aligned}$$

Therefore, we take $\overline{c} = \max\{N(p_0 + d_0) + e, N(\tilde{p}(\overline{d}) + \overline{d}) + e\}$ and u is bounded.

To see the second part, note that from our assumptions about u we have $u'(c) > 0$ for all c. Since $u''(c) < 0$ we must have that the maximum u' value occurs at the smallest c value, c_ε. From the CRRA form of the utility function we see that $u'(c) = c^{-\gamma} \leq (c_\varepsilon)^{-\gamma}$, which is clearly finite for any $\gamma > 0$. Thus marginal utility is bounded also. □

Let $(S, \mathcal{S})$ and $(Z, \mathcal{Z})$ be measurable spaces and let $(X, \mathcal{X}) = (S \times Z, \mathcal{S} \times \mathcal{Z})$ denote the product space. The set S is the *endogenous* state space (i.e. the values taken on by the

agent's stock holdings) and Z is the *exogenous* state space. A typical element in Z will be a dividend-price pair, (d, p). We will use X and $S \times Z$ interchangeably, using whichever one is most convenient given the current context. Note that the agent may have some control over the price since it is to be determined by the aggregate demand in the market. However, agents do not know this fact and they do not use it in any way. Thus, from the agent's point of view there are two exogenous variables, dividend and price. The agent knows the process the dividend follows and has hypothesized a function, $\tilde{p}$, to predict future prices, given dividends. Eventually, we will allow the agents to update their price forecasting rule to reflect outcomes in the market. At this point, however, the agent solves his problem for a fixed rule, and treats the rule as if it were correct.

In each period the agent must decide how many shares of stock to hold, given the current state of the world. His choice is subject to constraints, namely the budget constraint and market constraints such as no short sales and finite number of shares. Formally, the agent's choice set is described by a *correspondence* (or *set-valued mapping*) $\Gamma : S \times Z \to S$. Specifically, $\Gamma(s_t, z_t)$ is the set of all possible values for s_{t+1} if the current state is (s_t, z_t). We will be able to specify precisely what Γ looks like. First, we need to establish a lower bound on holdings in order to ensure the minimum consumption level is always feasible.

Proposition 2.3. *If $s_t \geq \dfrac{c_\varepsilon - e}{\underline{d}}$, then for all t there exists a choice for s_{t+1} that will lead to $c_t \geq c_\varepsilon$. Specifically, $c_t \geq c_\varepsilon$ for any $s_{t+1} \leq s_t$.*

Proof. Fix t and suppose $s_t \geq \dfrac{c_\varepsilon - e}{\underline{d}}$. Assume the agent chooses $s_{t+1} \leq s_t$. From the budget constraint we have

$$\begin{aligned} c_t &= p_t(s_t - s_{t+1}) + s_t d_t + e \\ &\geq s_t d_t + e \\ &\geq s_t \underline{d} + e \\ &\geq \frac{c_\varepsilon - e}{\underline{d}} \underline{d} + e = c_\varepsilon. \end{aligned}$$

□

The important point in Proposition 2.3 is that the agent may always choose to not trade (i.e. choose $s_{t+1} = s_t$) and consume the minimum amount, provided he began the period with holdings of at least $\frac{c_\varepsilon - e}{\underline{d}}$. This will be true for all t provided $s_0 \geq \frac{c_\varepsilon - e}{\underline{d}}$ since we have assumed the agent will never choose to consume less than c_ε.

We may now describe what our feasible correspondence looks like. Since the agent only has one choice variable, his feasible set will consist of just a subset of the real line. Proposition 2.3 provides a natural lower bound on the choice set. This bound, however, will often be negative if $e > 0$. Thus, we take the lower bound on s_{t+1} to be $\max\{0, \frac{c_\varepsilon - e}{\underline{d}}\}$, independent of the current states. To find an upper bound, we return to the budget constraint. Solving for s_{t+1} we get

$$s_{t+1} = \frac{s_t(p_t + d_t) + e - c_t}{p_t} \leq \frac{s_t(p_t + d_t) + e - c_\varepsilon}{p_t} \quad \forall t.$$

Thus we take $\Gamma(s_t, d_t, p_t) = \left[\underline{\Gamma}, \overline{\Gamma}_t\right]$, where $\underline{\Gamma} = \max\{0, \frac{c_\varepsilon - e}{\underline{d}}\}$ and $\overline{\Gamma}_t = \frac{s_t(p_t+d_t)+e-c_\varepsilon}{p_t}$.

Proposition 2.4. *The feasible correspondence defined above is nonempty for all t.*

Proof. We are required to show that $\overline{\Gamma}_t \geq \underline{\Gamma}$, for all t. There are two cases. First, suppose $\underline{\Gamma} = 0$. Then we must have $0 < c_\varepsilon \leq e$. By definition

$$\overline{\Gamma}_t = \frac{s_t(p_t + d_t) + e - c_\varepsilon}{p_t} \geq \frac{s_t(p_t + d_t)}{p_t} \geq 0 = \underline{\Gamma}.$$

Now, suppose $\underline{\Gamma} = \frac{c_\varepsilon - e}{\underline{d}}$. Then we have $\underline{\Gamma} \cdot \underline{d} = c_\varepsilon - e$. From the budget constraint we get

$$\begin{aligned}
\overline{\Gamma}_t &= \frac{s_t(p_t + d_t) + e - c_\varepsilon}{p_t} \\
&= \frac{s_t(p_t + d_t)}{p_t} - \frac{c_\varepsilon - e}{p_t} \\
&\geq \underline{\Gamma}\frac{(p_t + d_t)}{p_t} - \frac{\underline{\Gamma} \cdot \underline{d}}{p_t} \\
&= \underline{\Gamma}\left[1 + \frac{1}{p_t}(d_t - \underline{d})\right] \\
&\geq \underline{\Gamma}.
\end{aligned}$$

□

We will now state two properties of the feasible correspondence that will be useful later on. First, it is clear that for $s, s' \in S$ if $s \leq s'$, then $\Gamma(s, z) \subseteq \Gamma(s', z)$, for all $z \in Z$. Also, if $y \in \Gamma(s, z)$ and $y' \in \Gamma(s', z)$, then for any $\theta \in [0, 1]$ we have $\theta y + (1-\theta)y' \in \Gamma(\theta s + (1-\theta)s', z)$, for any z. This second property will be referred to as the *convexity* of Γ.

Next, we wish to define the agent's one period return function, whose domain is the graph of Γ. Define the set A as follows:

$$A = \{(s, y, z) \in S \times S \times Z |\ y \in \Gamma(s, z)\}.$$

Now, let $F : A \to \mathbb{R}$ be the one period return function. That is, $F(s, y, z)$ represents the current-period return if the state is (s, z) and the choice is y. In our case we will have $F(s, y, z) = u[s(p + d) + e - py]$, where $z = (d, p)$. Note that F is bounded since u is.

A (feasible) *policy* is a mapping $\pi : X \to S$ such that $\pi(x) \in \Gamma(x)$ for all $x \in X$. Thus, given a current state of the world, x, the agent will demand $\pi(x)$ shares of the asset. We will refer to the function π as a policy and the value $\pi(x)$ as an action. Let Π denote the set of all policies. Note that Π is nonempty since Γ is. Given any policy π and the distribution $q(z)$, we can define a conditional distribution $q_\pi(x_1, x_2, \dots | x_0)$ on the future states of the world. This value will represent the probability of observing the sequence of states $(x_1, x_2, \dots)$ given that the initial state was x_0 and the agent followed policy π.

For any initial state x_0 we can define the function $U(\cdot, x_0) : \Pi \to \mathbb{R}$ as

$$U(\pi, x_0) = E_\pi \left\{ \sum_{t=0}^{\infty} \beta^t F(s_t, \pi(s_t, z_t), z_t) \middle| x_0 \right\}, \tag{2.2}$$

where E_π is the expectation taken with respect to q_π. Clearly since F is bounded and $0 < \beta < 1$, U is bounded. $U(\pi, x_0)$ represents the agent's infinite, discounted, expected utility resulting from starting at state x_0 and using policy π forever. The agent's problem is to find π such that $U(\pi, x_0) \geq U(\pi', x_0)$ for all $\pi' \in \Pi$. When it is not confusing, we will denote $U(\pi)$ to mean $U(\pi, x_0)$, for any x_0. It should be kept in mind that everything is conditional on the initial state, which is known.

We next wish to define an operator that we will show is closely related to (2.2). To do so, let $M(X)$ denote the set of bounded, continuous functions mapping the state space X into $\mathbb{R}$. It can be shown that $M(X)$ is a Banach space under the sup norm. Unless otherwise stated, the sup norm will be used throughout this chapter. For any policy π, define the operator $T_\pi : M(X) \to M(X)$ by

$$(T_\pi v)(x) = (T_\pi v)(s, z) = F(s, \pi(s, z), z) + \beta \int v(\pi(s, z), z') dq(z'), \quad \text{for any } v \in M(X) \tag{2.3}$$

where z' represents exogenous state next period. Clearly $T_\pi v$ maps X into $\mathbb{R}$ and T_π is bounded since F and v are. T_π can be thought of as the value the agent receives by following action $\pi(s, z)$ today and terminating tomorrow with receipt of $v(x')$. If $v(x') = U(\pi', x')$ for some π', then we can interpret (2.3) as the value of following action $\pi(s, z)$ today and then following policy π' starting tomorrow. Note that if $v(x') = U(\pi, x')$ then T_π represents the

value of choosing action $\pi(s,z)$ today and then using policy π starting tomorrow. Clearly this is the same as just using π starting today which gives a value of $U(\pi,x)$. This argument shows that $U(\pi,x)$ is a fixed point of T_π, that is, $(T_\pi)U(\pi,x) = U(\pi,x)$, any $\pi \in \Pi$.

Lemma 2.5. *The operator T_π is monotone.*

Proof. This follows directly from the definition of T_π and the fact that the integral preserves monotonicity. □

Lemma 2.6 (Blackwell's Theorem)**.** *Let Q be a monotone operator mapping $M(X)$ into itself. If $Q(v+c) \leq Qv + \beta c$ for any $c \in \mathbb{R}$ and some $0 < \beta < 1$, then Q is a contraction mapping with modulus β.*

Proof. Let v_1 and v_2 be two functions in $M(X)$. Clearly, for any x, $v_1(x) - v_2(x) \leq \|v_2 - v_1\|$, so $v_1(x) \leq v_2(x) + \|v_2 - v_1\|$. Using the assumptions about Q we have

$$(Qv_1)(x) \leq Q(v_2(x) + \|v_2 - v_1\|) \leq (Qv_2)(x) + \beta\|v_2 - v_1\|$$

which implies

$$(Qv_2)(x) - (Qv_1)(x) \geq -\beta\|v_2 - v_1\|, \quad \text{for all } x.$$

Since it is also true that $v_2(x) - v_1(x) \leq \|v_2 - v_1\|$ we can repeat the above argument to obtain $(Qv_2)(x) - (Qv_1)(x) \leq \beta\|v_2 - v_1\|$. Combining these two inequalities gives $|Qv_2(x) - Qv_1(x)| \leq \beta\|v_2 - v_1\|$, for all x. This implies that $\|Qv_2 - Qv_1\| \leq \beta\|v_2 - v_1\|$, as required. □

Together the previous two lemmas show that T_π is a contraction mapping of modulus β, for any policy π.

Next, let us define a new operator $T : M(X) \to M(X)$ as

$$(Tv)(x) = (Tv)(s,z) = \sup_{y \in \Gamma(x)} \left\{ F(s,y,z) + \beta \int v(y,z')dq(z') \right\}. \tag{2.4}$$

Clearly Tv is bounded. Tv can be interpreted as the value realized from choosing an optimal action today and terminating tomorrow with receipt of $v(x')$. If $v(x') = U(\pi', x')$ for some policy π' then we can interpret (2.4) as the value of choosing an optimal action today, then following policy π' forever, starting tomorrow. Suppose, for the moment, that there exists a function $v^* \in M(X)$ such that $v^*(x) = (Tv^*)(x)$ for each $x \in X$. Also, suppose that

$v^*(x) = U(\pi^*, x)$ for some $\pi^* \in \Pi$. Then the right hand side of (2.4) represents the value of choosing an optimal action today, given that tomorrow the agent will follow policy π^*. On the other hand, the left hand side represents the value of following π^* beginning today. Thus π^* is the optimal policy and it gives the agent value $v^*(x)$ for any initial state x.

We next address the issue of the existence and uniqueness of the function v^* and finally show how it relates to the solution to the agent's original, infinite sequence problem.

Lemma 2.7. *The mapping T is a contraction of modulus β.*

Proof. Fix $x \in X$ and choose any two functions $v_1, v_2 \in M(X)$. Define $k = (Tv_2)(x) - (Tv_1)(x)$. Without loss of generality, assume $k \geq 0$. By the definition of T, for any positive integer n, there exists π_n such that $(Tv_2)(x) - (T_{\pi_n} v_2)(x) \leq k/n$ which gives $(Tv_2)(x) - k/n \leq (T_{\pi_n} v_2)(x)$. Also, by the definition of k we have $(Tv_2)(x) - (Tv_1)(x) = k \geq k/n$ which gives $(Tv_2)(x) - k/n \geq (Tv_1)(x)$. Combining we get $(Tv_1)(x) \leq (Tv_2)(x) - k/n \leq (T_{\pi_n} v_2)(x)$. Subtracting $(Tv_1)(x)$ everywhere gives $0 \leq (Tv_2)(x) - (Tv_1)(x) - k/n \leq (T_{\pi_n} v_2)(x) - (Tv_1)(x)$. Using the fact that $Tv \geq T_\pi v$ for any π and any v, we get

$$(T_{\pi_n} v_2)(x) - (Tv_1)(x) \leq (T_{\pi_n} v_2)(x) - (T_{\pi_n} v_1)(x) \leq \|T_{\pi_n} v_2 - T_{\pi_n} v_1\| \leq \beta \|v_2 - v_1\|,$$

where we have used the fact that T_π is a contraction. Since the above holds for any n, we take the limit as $n \to \infty$ to get $|(Tv_2)(x) - (Tv_1)(x)| \leq \beta \|v_2 - v_1\|$, for any choice of v_1, v_2, and x. Thus we have $\|Tv_2 - Tv_1\| \leq \beta \|v_2 - v_1\|$ as required. □

We now state a very useful theorem from real analysis (Banach (1922), Theorem 6).

Theorem 2.8 (Banach Fixed Point Theorem)**.** *Let f be a contraction mapping from a closed subset F of a Banach space E into F. Then there exists a unique $z \in F$ such that $f(z) = z$.*

Using Theorem 2.8 and Lemma 2.7 it can easily be shown that there exists a unique function $v^* \in M(X)$ satisfying $v^* = Tv^*$. In addition, if v is any function in $M(X)$ then $v^* = \lim_{n \to \infty} T^n v$, where $T^n v = T(T^{n-1} v)$ with $T^0 v = v$. Thus, we are not only guaranteed a unique solution, we also have a method to construct the solution. It remains to show that the solution, v^*, is also a solution to the agent's original maximization problem, which we will denote as $U^*(x)$. That is,

$$U^*(x) = \sup_{\pi \in \Pi} U(\pi, x). \tag{2.5}$$

Before stating the central theorem to this section we need to establish two preliminary results. We will denote $U(\pi)$ to mean $U(\pi, \cdot)$.

Lemma 2.9. *For any $\pi \in \Pi$ and any $v \in M(X)$, $\|U(\pi) - v\| \leq \dfrac{\|T_\pi v - v\|}{1-\beta}$.*

Proof. It was previously argued that $U(\pi)$ is a fixed point of T_π, for any π. Since T_π is a contraction it must be the case that $T_\pi^n v \to U(\pi)$ for any $v \in M(X)$. By uniqueness of the fixed point this implies that $\lim_{n\to\infty} T_\pi^n v \to U(\pi)$, for any $\pi \in \Pi$ and any $v \in M(X)$. Now, using the triangle inequality and the fact that T_π is a contraction gives

$$\|T_\pi^n v - v\| \leq \sum_{i=1}^{n} \|T_\pi^i v - T_\pi^{i-1} v\| \leq \sum_{i=1}^{n} \beta^{i-1} \|T_\pi v - v\|.$$

Letting $n \to \infty$ on both sides gives

$$\|U(\pi) - v\| \leq \frac{\|T_\pi v - v\|}{1-\beta}, \text{ for any } \pi \in \Pi, v \in M(X). \tag{2.6}$$

□

Lemma 2.10. *For any $\varepsilon > 0$, there exists a policy π such that $\|U(\pi) - v^*\| \leq \varepsilon$.*

Proof. From the definition of T there exists π such that $\|T_\pi v^* - T v^*\| \leq \varepsilon(1-\beta)$. Thus, using Lemma 2.9 with $v = v^*$, we obtain the required result. □

We are now ready to prove the main result of this section. The following theorem establishes the equivalence of solutions to (2.4) and (2.5).

Theorem 2.11. *If $v^* = Tv^*$, then $v^* = U^*$.*

Proof. We first show $v^* \leq U^*$. Suppose, on the contrary, that there exists some x^1 such that $v^*(x^1) > U^*(x^1)$. Then by the above claim there also exists a policy, π^1, such that, for any $\varepsilon > 0$, $\|U(\pi^1) - v^*\| \leq \varepsilon$. Then clearly we have $|U(\pi^1, x^1) - v^*(x^1)| \leq \varepsilon$. If $U(\pi^1, x^1) \geq v^*(x^1)$, then $U(\pi^1, x^1) > U^*(x^1)$ also. This contradicts the definition of U^*. If $v^*(x^1) > U(\pi^1, x^1)$, then $v^*(x^1) - U(\pi^1, x^1) \leq \varepsilon$. This gives $U^*(x^1) < v^*(x^1) \leq U(\pi^1, x^1) + \varepsilon$. Since this inequality holds for *any* $\varepsilon > 0$, it follows that $U^*(x^1) < U(\pi^1, x^1)$, a contradiction. Therefore it must be the case that $v^* \leq U^*$.

To show the inequality in the other direction is simpler. From the definition of T we have $T_\pi v^* \leq T v^* = v^*$, for any policy π. Since T_π is monotone we have $T_\pi^2 v^* \leq T_\pi v^* \leq T v^* = v^*$. Continuing in this way gives $T_\pi^n v^* \leq v^*$, for any n. Taking the limit as $n \to \infty$ gives $U(\pi) \leq v^*$, for any π. Thus, we must have $U^* = \sup_\pi U(\pi) \leq v^*$. Therefore $U^* = v^*$ as required. □

Corollary 2.12. *Suppose that for each $x = (s, z)$, $\pi^*(x)$ solves*

$$\sup_{y \in \Gamma(s,z)} \left\{ F(s, y, z) + \beta \int v^*(y, z') dq(z') \right\}. \tag{2.7}$$

Then $U^* = U(\pi^*)$. That is, π^* is the optimal policy.

Proof. Since π^* solves (2.7) for all x we have $T_{\pi^*} v^* = T v^* = v^*$. But as we just showed, $v^* = U^*$. This gives $T_{\pi^*} U^* = U^*$ so U^* is a fixed point of T_{π^*}. From the preceding proof we know that $U(\pi)$ is the unique fixed point of T_π, for any π. Therefore $U^* = U(\pi^*)$. □

2.3 Properties of the Value Function

We have now shown that to solve the agent's infinite lifetime utility problem, it is sufficient to solve a corresponding two period problem. Furthermore, we are not only guaranteed a unique solution to the two period problem but we are provided with a very convenient way to construct such a solution. The remainder of this chapter will study the properties of the functional equation and its solutions. Before proceeding, we require a preliminary result that we will state without proof. This is the Theorem of the Maximum (see Harris (1987), Theorem 1.1, page 11).

Theorem 2.13 (Theorem of the Maximum)**.** *Suppose $\phi : X \times Y \to \mathbb{R}$ is continuous on $X \times Y$. If Γ is continuous on X and if $\Gamma(x) \neq \emptyset$ for every $x \in X$, then $M : X \to \mathbb{R}$ defined by*

$$M(x) = \max_{y \in \Gamma(x)} \phi(x, y)$$

is continuous on X, and the set of maximizers

$$\Phi(x) = \operatorname*{argmax}_{y \in \Gamma(x)} \phi(x, y) = \{ y \in Y |\ y \in \Gamma(x), \phi(x, y) = M(x) \}$$

is nonempty and compact-valued. Furthermore, if Φ is a singleton for every $x \in X$, then $\Phi : X \to Y$ is continuous on X.

In our current setup, we have

$$\phi(x, y) = \phi(s, y, z) = F(s, y, z) + \beta \int v(y, z') dq(z'). \tag{2.8}$$

Since v and F are bounded and continuous, and q is a probability measure, it follows that $\phi(x, y)$ is bounded and continuous for all x and y. Also, for each x, $\Gamma(x)$ is a compact set.

Therefore the supremum in (2.4) will actually be attained, for each x. This allows us to replace "sup" with "max". Since prices are nonzero, the upper and lower bounds on the correspondence Γ are continuous functions of the state variable $x = (s, d, p)$. Thus, Γ is a continuous correspondence. We can therefore apply Theorem 2.13 to conclude that $v^*(x)$ is a continuous function. Note that v^* will also be bounded since clearly the maximum of a set of bounded functions is bounded also. Given a solution $v^* = Tv^*$ we can define a correspondence

$$G(s, z) = \{y \in \Gamma(s, z) | \phi(s, y, z) = v^*(s, z)\} \tag{2.9}$$

to be the set of all choices $y \in \Gamma(s, z)$ that lead to the optimum. We will show that G is single-valued and therefore a continuous function on $S \times Z$.

Theorem 2.14. *The function $\phi(\cdot, \cdot, z)$ defined in (2.8) is strictly concave for any z, provided v is at least weakly concave.*

Proof. Fix z and let $s_1, s_2, y_1, y_2 \in S$, $0 < \theta < 1$. Define $c_i = s_i(p + d) + e - py_i$ for $i = 1, 2$. Then using the fact that u is strictly concave we have

$$\begin{aligned} F[\theta(s_1, y_1, z) + (1 - \theta)(s_2, y_2, z)] &= u[\theta c_1 + (1 - \theta)c_2] \\ &> \theta u(c_1) + (1 - \theta)u(c_2) \\ &= \theta F(s_1, y_1, z) + (1 - \theta)F(s_2, y_2, z) \end{aligned}$$

Therefore F is strictly concave in s and y. Since the sum of a strictly concave function and a weakly concave function is strictly concave, we need only show that the integral term is weakly concave if v is. Since this term is independent of s we need only show that it is weakly concave in y. Let $y_1, y_2 \in Y$, suppose $v \in M(X)$ is weakly concave and denote $(Iv)(y) = \int v(y, z')dq(z')$. Then for $0 < \theta < 1$ we have

$$\begin{aligned} (Iv)[\theta y_1 + (1 - \theta)y_2] &= \int v[\theta y_1 + (1 - \theta)y_2, z']dq(z') \\ &\geq \int [\theta v(y_1, z') + (1 - \theta)v(y_2, z')]\, dq(z') \\ &= \theta(Iv)(y_1) + (1 - \theta)(Iv)(y_2). \end{aligned}$$

□

Theorem 2.15. *If v is weakly concave in s, then the mapping $(Tv)(s, z)$ is strictly concave in s, for all z. In addition, $v^*(s, z)$ is also strictly concave in s.*

Proof. Fix $z \in Z$ and let $s_1, s_2 \in S$. Choose $0 < \theta < 1$. Define $s_\theta = \theta s_1 + (1-\theta)s_2$. From previous discussion we know that the maximum in the operator T will actually be attained. Suppose that $y_i \in \Gamma(s_i, z)$ attains this maximum for s_i, $i = 1, 2$, and define $y_\theta = \theta y_1 + (1-\theta)y_2$. By the convexity of Γ we have $y_\theta \in \Gamma(s_\theta, z)$. Thus we have

$$\begin{aligned}
(Tv)(s_\theta, z) &= \max_{y \in \Gamma(s_\theta, z)} \phi(s_\theta, y, z) \\
&\geq \phi(s_\theta, y_\theta, z) \\
&> \theta\phi(s_1, y_1, z) + (1-\theta)\phi(s_2, y_2, z) \qquad \text{[by Theorem 2.14]} \\
&= \theta(Tv)(s_1, z) + (1-\theta)(Tv)(s_2, z) \qquad \text{[by definition of } y_i\text{]}.
\end{aligned}$$

To prove the second part, note that if we begin with a weakly concave function $v_0 \in M(X)$, then $T^n v_0$ is strictly concave, for any n. Taking the limit we see that $v^* = \lim_n T^n v_0$ is also strictly concave.

□

Corollary 2.16. *With $\phi(s, y, z)$ defined as above, the maximum $M(s, z) = \max_{y \in \Gamma(s,z)} \phi(s, y, z)$ is attained at a unique y value.*

Proof. Fix s and z and suppose that there are two values $y_1, y_2 \in \Gamma(s, z)$ that attain the maximum. Then we have $v^*(s, z) = F(s, y_i, z) + \beta \int v^*(y_i, z')dq(z')$, for $i = 1, 2$. Let $0 < \theta < 1$. Since Γ is convex, $y_\theta = \theta y_1 + (1-\theta)y_2 \in \Gamma(s, z)$. Thus, using the strict concavity of F and v^*, we have

$$\begin{aligned}
v^*(s, z) &\geq F(s, y_\theta, z) + \beta \int v(y_\theta, z')dq(z') \\
&> \theta F(s, y_1, z) + (1-\theta)F(s, y_2, z) + \beta \int \left[\theta v^*(y_1, z') + (1-\theta)v^*(y_2, z')\right] dq(z') \\
&= \theta \left[F(s, y_1, z) + \beta \int v^*(y_1, z')dq(z')\right] + (1-\theta)\left[F(s, y_2, z) + \beta \int v^*(y_2, z')dq(z')\right] \\
&= \theta v^*(s, z) + (1-\theta)v^*(s, z) \\
&= v^*(s, z), \quad \text{a contradiction.}
\end{aligned}$$

Therefore we conclude that the maximum is attained at a unique value $y^* \in \Gamma(s, z)$. □

This Corollary shows that the correspondence that we defined in (2.9) is in fact a single valued function, and thus by Theorem 2.13, continuous. We will denote this function by $g : S \times Z \to S$ and refer to it as the agent's *policy function*. The following theorem establishes the monotonicity of the operator T.

Theorem 2.17. *For any $v \in M(X)$, the function $(Tv)(s,z)$ is strictly increasing in s, for all z.*

Proof. Let $s_1, s_2 \in S$ and suppose $s_1 < s_2$. Then we have

$$\begin{aligned}(Tv)(s_1,z) &= \max_{y\in\Gamma(s_1,z)}\left\{F(s_1,z,y)+\beta\int v(y,z')dq(z')\right\}\\ &\leq \max_{y\in\Gamma(s_2,z)}\left\{F(s_1,z,y)+\beta\int v(y,z')dq(z')\right\}\\ &< \max_{y\in\Gamma(s_2,z)}\left\{F(s_2,z,y)+\beta\int v(y,z')dq(z')\right\}\\ &= (Tv)(s_2,z)\end{aligned}$$

where the second line follows from the fact that $s_1 < s_2 \Rightarrow \Gamma(s_1,z) \subseteq \Gamma(s_2,z)$ and the third from the fact that F is strictly increasing in s (since $u' > 0$). □

It remains to show that the function v^*, which solves the agent's infinite period maximization problem, is differentiable. Once we have established differentiability we can derive the first order conditions for the problem. We begin with a statement of a theorem first proved by Benveniste and Scheinkman. For a proof see (Beneveniste and Scheinkman, 1979).

Theorem 2.18 (Benveniste and Scheinkman)**.** *Let V be a real valued, concave function on a convex set $D \subseteq \mathbb{R}^n$. If W is a concave, differentiable function in a neighborhood N of x_0 in D with the property that $W(x_0) = V(x_0)$ and $W(x) \leq V(x)$ for all x in N, then V is differentiable at x_0.*

In the following theorem we use the notation *int* X to mean the interior of the set X.

Theorem 2.19. *Let v^* be the unique solution to (2.4) and let g be the policy function. If $x_0 \in$ int X and $g(x_0) \in$ int $\Gamma(x_0)$, then v^* is continuously differentiable in s at s_0 and*

$$\frac{\partial}{\partial s}v^*(s_0,z_0) = \frac{\partial}{\partial s}F[s_0,g(s_0,z_0),z_0].$$

Proof. Fix s_0 and z_0. Recall that it was previously argued that Γ is a continuous correspondence. Using continuity and $g(s_0,z_0) \in$ *int* $\Gamma(s_0,z_0)$ it follows that there exists a neighborhood, N, of s_0 such that $g(s,z_0) \in \Gamma(s,z_0)$, for all $s \in N$. Define a new function w on D as

$$w(s,z_0) = F[s,g(s_0,z_0),z_0] + \beta\int v^*(g(s_0,z_0),z')dq(z'). \tag{2.10}$$

Clearly, since F is concave and differentiable in s, so is w. Also, since g is the optimal policy, we can see that $w(s_0, z_0) = v^*(s_0, z_0)$. Now, for any $s \in N$,

$$\begin{aligned} v^*(s, z_0) &= \max_{y \in \Gamma(s, z_0)} \left\{ F[s, y, z_0] + \beta \int v^*(y, z') dq(z') \right\} \\ &\geq F[s, g(s_0, z_0), z_0] + \beta \int v^*(g(s_0, z_0), z') dq(z') \qquad [\text{since } g(s_0, z_0) \in \Gamma(s, z_0)] \\ &= w(s, z_0). \end{aligned}$$

Thus, by Theorem 2.18, the function v^* is differentiable in s. Taking derivatives of (2.10) gives the second part. □

2.4 First Order Conditions

We are finally at a point where we can derive the first order conditions for the agent's problem. For the remainder of this chapter we will drop the $*$ notation and denote $v^* = v$. We will also return to expectation notation with the understanding that the expectation is taken over all perceived future states of the world. We will use $\tilde{E}$ to denote the agent's expectation taken while using the price function $\tilde{p}$. Before proceeding we will need to establish a preliminary result from measure theory.

Lemma 2.20. *Let X be an open subset of $\mathbb{R}$, and Ω a measure space. Suppose a function $f : X \times \Omega \to \mathbb{R}$ satisfies the following conditions:*

1. *$f(x, \omega)$ is a measurable function of ω, for each $x \in X$.*
2. *For almost all $\omega \in \Omega$, the derivative $\partial f(x, \omega)/\partial x_i$ exists for all $x \in X$.*
3. *There is an integrable function $\Theta : \Omega \to \mathbb{R}$ such that $|\partial f(x, \omega)/\partial x| \leq \Theta(\omega)$ for all $x \in X$.*

Then

$$\frac{\partial}{\partial x} \int_\Omega f(x, \omega) d\mu = \int_\Omega \frac{\partial}{\partial x} f(x, \omega) d\mu.$$

Proof. Define the function $F : X \to \mathbb{R}$ by $F(x) = \int_\Omega f(x, \omega) d\mu$. This function is well defined given assumption 1. Let $\{x_n\}$ be a sequence in X converging to x. We aim to show

$\lim_{n\to\infty} \frac{F(x_n)-F(x)}{x_n - x} = \int_\Omega \frac{\partial}{\partial x} f(x,\omega) d\mu$. Using the definition of F and elementary properties of integrals, we have

$$\frac{F(x_n)-F(x)}{x_n - x} = \int_\Omega \frac{f(x_n,\omega)-F(x,\omega)}{x_n - x} d\mu.$$

By the Mean Value Theorem we have, for all n and ω,

$$\frac{f(x_n,\omega)-f(x,\omega)}{x_n - x} = \frac{\partial}{\partial x} f(x^*,\omega)$$

for some $x^*(\omega)$ between x_n and x. By assumption 3, the right hand side of this expression is bounded, for any ω. Therefore we can apply the Dominated Convergence Theorem to obtain

$$\begin{aligned}
\lim_{n\to\infty} \frac{F(x_n)-F(x)}{x_n - x} &= \lim_{n\to\infty} \int_\Omega \frac{f(x_n)-f(x)}{x_n - x} d\mu \\
&= \int_\Omega \lim_{n\to\infty} \frac{f(x_n)-f(x)}{x_n - x} d\mu \qquad \text{[by DCT]} \\
&= \int_\Omega \frac{\partial}{\partial x} f(x,\omega) d\mu,
\end{aligned}$$

as required. □

If we return for a moment to the ϕ notation for the agent's problem, it is

$$v(s,z) = \max_y \underbrace{F(s,y,z) + \beta \tilde{E} v(y,z')}_{\phi(s,y,z)}. \tag{2.11}$$

Suppose $y^* = s'$ is the optimal value. Then it must be the case that the partial derivative of ϕ with respect to y, evaluated at s', is 0. Computing the derivative gives

$$\frac{\partial}{\partial y}\phi(s,y,z)\Big|_{y=s'} = F_2(s,s',z) + \beta \tilde{E} v_1(s',z') = 0, \tag{2.12}$$

where the subscript i denotes the partial derivative with respect to the ith variable. Note that we have applied Lemma 2.4 to take the derivative inside of the expectation.[4] Recall from the budget constraint (which binds with equality) that $c = s(p+d) + e - ps'$ and so then $c' = s'(p'+d') + e - p's''$. From the definition of F we have $F_2(s,s',z) = u'[s(p+d)+e-ps'](-p) = u'(c)(-p)$.[5] Also, from Theorem 2.19 we have $v_1(s,z) = F_1(s,s',z) = u'(c)(p+d)$.

[4] In order to compute $\partial v(s',z')/\partial s$ we compute $\partial v(s,z)/\partial s$ and evaluate it at (s',z'). Since u' is bounded and $\partial v(s,z)/\partial s = u'(c)(p+d)$, we can apply Lemma 2.4 with $\Theta(z') = \Theta(d',p') = A(p'+d')$, where $|u'(c)| \leq A$. This function is integrable since p' will be computed using the polynomial $\tilde{p}(d')$, for bounded d'.

[5] Prime notation has multiple meanings here. A prime on a variable denotes that variable's value next period. Two primes denotes the value two periods ahead. A prime on a function of one variable denotes the function's derivative.

Updating all variables one period we get $v_1(s', z') = F_1(s', s'', z') = u'(c')(p' + d')$. Plugging these into (2.12) gives

$$u'(c)p = \beta\tilde{E}[u'(c')(p' + d')]. \tag{2.13}$$

This is the well-known *Euler Equation*. The Euler Equation can be written more compactly if we define additional variables. Let $m' = \beta u'(c')/u'(c)$ and $R' = (p' + d')/p$. Then (2.13) can be written as

$$\tilde{E}[m' \cdot R'] = 1. \tag{2.14}$$

The variable m' is the stochastic *discount factor* or *pricing kernel*. This kernel can be used to price any asset. The value R' is the one period return on the asset. Given these definitions, (2.14) has a natural interpretation. In each period the agent may choose to do one of two things with his available wealth; consume or invest. If he consumes one unit of the good he receives a return of one with certainty. If he invests his return is uncertain. It must be true, however, that the expected, discounted return on investing must be equal to one also. Consider the case when the expected, discounted return on investing is greater than the return on consuming. Then the agent would choose to invest more. Similarly, if the situation were reversed he would consume more. Thus, it must be true that the two returns are equal. This is nothing more than the familiar condition that marginal revenue must equal marginal cost.

Given the first order conditions we are able to state a very useful theorem. This result will be used later to assist in speeding up computation time.

Theorem 2.21. *For each z, the function $g(\cdot, z)$ is strictly increasing.*

Proof. Fix z and let $s_1 < s_2$. Suppose, on the contrary, that $g(s_1, z) \geq g(s_2, z)$. Then since v is strictly concave and integration preserves monotonicity we have

$$\tilde{E}v_1[g(s_1, z), z'] < \tilde{E}v_1[g(s_2, z), z'], \quad \text{for all } z'. \tag{2.15}$$

Let $c_i = s_i(p + d) + e - pg(s_i, z)$, $i = 1, 2$. Then $c_1 < c_2$ and since u is strictly concave we have $u'(c_1) > u'(c_2)$. From (2.12) we have $F_2[s_i, g(s_i, z), z] + \beta\tilde{E}v[g(s_i), z'] = 0$ for $i = 1, 2$. This gives

$$\beta\tilde{E}v[g(s_1, z), z'] = pu'(c_1) > pu'(c_2) = \beta\tilde{E}v[g(s_2, z), z'],$$

contradicting (2.15). □

Given the fact that the policy function is monotone in s, Araujo (1991) shows that it is necessarily differentiable. In addition, Araujo shows the value function in this case is twice continuously differentiable.

2.5 Summary

In this chapter we have specified, formally, the class of problems for which we are interested in computing a numerical solution. Specifically we have defined the agent's objective and shown its equivalence to a corresponding two-period problem. We have established many desirable properties of this corresponding problem, including the existence and uniqueness of a solution, as well as a method to construct the solution by starting with a very general function. We have also established several facts about the mappings and solutions that will prove very useful in the following chapters. Specifically we will be able to use the concavity of the objective function and the monotonicity of the policy function to reduce our search space when looking for the maximum. As we will discuss in the following chapter, the first order conditions are not immediately useful for directly computing the policy function. They will, however, provide a way to check that our numerical method is, in fact, finding the optimal solution.

CHAPTER 3

COMPUTATIONAL METHODS I: THE SINGLE AGENT'S PROBLEM

In this chapter we discuss the numerical methods required to solve the single agent's problem discussed in the Chapter 2. We use a grid search algorithm, which has the advantage of guaranteed success, but is known to be slow. We will discuss several ways to speed up the algorithm and increase its accuracy. We compare our numerical approximations to a known, analytical solution for certain parameter specifications. In addition, we explore some properties of the demand functions in the case where we know the analytical solution and cases for which no such formula exists.

As in Chapter 2 we will suppress the i notation on the agents when it is not confusing. It should be realized, however, that the algorithm described below will need to be carried out for each agent.

3.1 Numerical Methods

There are two common approaches to computing the agent's policy function. The most basic approach, is *value function iteration*. In this case we begin with an initial guess at the value function and repeatedly iterate the mapping T defined in (2.11) until convergence is obtained. At each iteration the maximum is computed using a search over a discrete grid and is thus sometimes referred to as the *grid search method*. The grid search method will be discussed at length in Section 3.2.

The second type of approach to finding the agent's policy function is to use the fact that the Euler equation

$$u'(c)p = \beta\tilde{E}[u'(c')(p' + d')] \tag{3.1}$$

must hold at the optimum. Taylor and Uhlig (1990) provides a survey of many of the methods that can be used to solve for the agent's policy function via the Euler equation.

One particularly elegant method for solving functional equations is the so called *projection method* (see Judd (1998), Chapter 11). Unlike many approximation methods that give a discrete approximation to an unknown function, the projection method finds an approximating function that is continuous in each of its variables. Since, in general, we believe the true solutions are continuous, it is desirable to have this property in the approximation also.

In this section we will briefly describe the projection method and try to provide some insight into why it was not suitable for our problem.

Let g denote the policy function. Then we have $s' = g(s,d,p)$ and $s'' = g(s',d',p') = g(g(s,d,p),d',p')$. Using the budget constraint, we have

$$c = s(p+d) + e - pg(s,d,p), \tag{3.2}$$

$$c' = g(s,d,p)(p'+d') + e - p'g(g(s,d,p),d',p'). \tag{3.3}$$

Substituting (3.2) and (3.3) into the Euler equation and using the fact that $u'(c) = c^{-\gamma}$, we have

$$\frac{p}{[s(p+d)+e-pg(s,d,p)]^{\gamma}} = \beta E\left\{\frac{p'+d'}{[g(s,d,p)(p'+d')+e-p'g(g(s,d,p),d',p')]^{\gamma}}\right\}. \tag{3.4}$$

The projection method proceeds as follows. Suppose that we assume g is of the form

$$g(s,d,p) = \sum_{i=1}^{r}\sum_{j=1}^{s}\sum_{k=1}^{t} a_{ijk}\varphi_i(s)\psi_j(d)\xi_k(p), \tag{3.5}$$

where a_{ijk} are constants and φ_i, ψ_j, ξ_k are basis functions. It is usually convenient to choose the basis functions to be orthogonal polynomials. Typical choices are Legendre or Hermite polynomials (see Judd (1998), Chapter 6, for a complete discussion of orthogonal polynomials).

Next, we substitute our specific form for g, (3.5), into (3.4). It remains to solve for the constants, a_{ijk}, such that the approximating function satisfies (3.4) sufficiently well. There are many ways to define what one means by "sufficiently well". The easiest way is to force the equation to be equal at exactly rst points. This is called the *collocation method*. Another popular choice is the *Galerkin method*, which will impose rst orthogonality conditions to solve for the constants. This will be more complicated since it will, in general, involve

approximating multidimensional integrals. Judd provides additional methods for solving for the constants.

Let us suppose we use uniform collocation. That is, we will specify a uniform grid of points and force (3.4) to hold exactly at each point. Since g appears in a very nonlinear way in (3.4), the collocation method will require that we solve rst simultaneous, nonlinear equations. Even for modest values of r, s and t, this will prove to be a very difficult task. Even if we assume a solution exists, nonlinear solvers often get stuck when searching for solutions to such systems. As a result we found the projection approach to be unreliable and thus, not feasible for our problem.

We briefly attempted a handful of other popular methods to solve (3.4), but in each case we were able to see almost immediately that they would not be suitable. The two main difficulties we face are that our problem is three dimensional and, in general, very nonlinear. The combination of the two is enough to rule out most solution methods.

3.2 The Grid Search Method

The discussion in the previous section outlines why sophisticated methods are difficult or impossible for solving our model. As a result, we resort to an iterative approach that is known to be reliable, but is usually quite slow. While we would like to have a fast method, it is more important to have a method that we can trust to work properly. This section will describe such a method.

We begin by re-stating the agent's two-period problem: The agent wishes to find the unique function v that satisfies

$$v(s,z) = (Tv)(s,z) = \max_{s'} \left\{ F(s,s',z) + \beta \tilde{E} v(s',z') \right\}. \tag{3.6}$$

We continue to use $F(s,s',z) = u[s(p+d) + e - ps']$ for ease of notation. Given an initial guess at the function, v^0, we can iterate (3.6) repeatedly to obtain the unique fixed point. The remainder of this section will discuss how to carry out this repeated iteration.

3.2.1 Choice of v^0

It was shown in Chapter 3 that since (3.6) is a contraction, with a start of any initial $v^0 \in M(X)$, the iteration will converge to the unique solution. Although this is true, we will

need to be a little more careful with our choice of v^0 for the sake of efficiency. Later in this chapter we will discuss how to use the concavity of the objective of (3.6) to speed up the computation time so we need to be sure that we always have concavity. Recall from Theorem 2.14 in Chapter 2 that we are only guaranteed concavity if v is at least weakly concave. This implies that any constant v^0 will work. The most natural choice is $v^0(s,z) = 0$ for all s and z, but the closer we start to the true solution, the less time it will take to converge.

Suppose, for the moment, that the economy is at a deterministic steady-state. Then $s' = s = 1$, $d = d' = \overline{d}$ and $p = p' = \overline{p}$, where a bar indicates the expected value of the random variable. In this case (3.6) becomes $v(s,z) = u(\overline{d} + e) + \beta v(s,z)$ which gives $v(s,z) = u(\overline{d} + e)/(1-\beta)$. Taking v^0 to be this (constant) function will allow us to begin closer to the true solution and thus speed up our computing time.

3.2.2 Discretization of the State Space

Since we will be using a grid search method to compute our function, we need to define the grid over which the search will take place. Specifically, we need to define upper and lower bounds and a step size for each state variable. If d follows a discrete process then we can simply use the possible values of d as the grid in that direction and the associated probabilities. Otherwise we will need to choose a suitable discretization. For a normal distribution, for example, Tauchen (1986) provides a nice way to discretize and generate a corresponding probability matrix.

For the s grid we have assumed short sales are not allowed and there are N shares of the stock available for purchase, where N is the number of agents in the market. Thus we can take $s \in [s_\varepsilon, N]$, where s_ε satisfies the inequality in Proposition 2.3.

The choice of the p grid will not be as definitive. Since the agents could be trading in an environment that is far from equilibrium they will need to be able to evaluate their policy function in a wide range of possible p values. The only way to be sure what this range will be is to run the model, see what prices result and adjust the grid appropriately. If agents are close to equilibrium, then a range of p values chosen near the equilibrium prices is appropriate. The routines used are designed to throw errors if the grid is too small to contain all relevant prices.

Denote the grids by $\mathbf{S} = \{s_i\}$, $\mathbf{D} = \{d_j\}$ and $\mathbf{P} = \{p_k\}$ and let their sizes be n_s, n_d and n_p, respectively. Note that we have used different indices on purpose here in order to keep

track of which one corresponds to which variable. When it is convenient we will denote the product space of shocks as $\mathbf{Z}$ with a typical element $z_{jk} = (d_j, p_k)$. *Note that the grids are all assumed to be ordered from smallest to largest.*

The step size will be discussed in Section 3.4.

3.2.3 Convergence

We know from Chapter 2 that given v^0 if we compute $\lim_n(T^n v^0)$ then we will converge to the unique solution. Numerically, however, we are unable to iterate an infinite number of times and thus we need to decide at what point we consider the answer "close enough" to the true solution.

There are two notions of convergence. Recall that we are using the sup norm unless otherwise specified.

$C1$: Convergence of the value functions. Let v^n denote the computed value function at iteration n and specify $\varepsilon > 0$. We will say that the value function iterations have converged if $\|v^n - v^{n-1}\| < \varepsilon$.

$C2$: Convergence of the policy functions. Let g^n denote the policy computed at iteration n and specify positive integer M. We will say that the policy functions have converged if $\|g^n - g^{n-1}\| = 0$ for M consecutive iterations.

Some explanation of the above concepts is in order. The agent's problem is essentially the following: For each state of the world, find the next period holdings within $\mathbf{S}$ to maximize the right hand side of (3.6). That is, he will be choosing from a finite number of possible holdings. Thus, the policy function will only take on a finite number of possible values.[1] This means that we can expect that after some iterating the policy functions do not change at all from one iteration to the next. The value function, on the other hand, can take on all real values. Due to rounding and machine precision it is not reasonable to ever expect that $\|v^1 - v^0\| = 0$. Instead we consider the value function to have converged when this norm is sufficiently small. Later in this chapter we will modify our approach to allow the agent to choose any real value for holdings also, and thus we will modify our convergence criteria accordingly.

[1] This will turn out to be a downfall of the grid search approach, but we will present a solution to this problem later.

Since we are ultimately interested in the agent's policy function and the value function is only a tool that we use to obtain the policy, it makes sense that we use the second notion of convergence as our stopping rule. The reason to check if the policy function remains unchanged for M iterations instead of just stopping the first time this is true is to avoid the case of stopping due purely to luck. The policy functions, g^n, may or may not converge to the true function g monotonically. Thus, it is possible that we could observe no change from one iteration to the next even though they have not yet converged. In practice, however, the policy functions do converge monotonically, at least early on, and then only occasionally will show a change of zero followed by a nonzero change but in this case the value functions are sufficiently close (since they do converge monotonically since T is a contraction) and thus we are sure to be near the true solution. Therefore, as long as the value functions are sufficiently close and the policy functions have remained unchanged for a small number of steps we can safely assume we have reached the solution. In short, we use a combination of the convergence criteria $C1$ and $C2$. Unless otherwise specified we will use $M = 2$ and $\varepsilon = 0.01$ throughout this chapter.

3.2.4 The Grid Search

Algorithm 3.1 shows how to solve for the policy function of a single agent. Note that we have not yet specified how the agent will carry out Step 1. At this point it should be assumed that the agent will fix (i, j, k) and for each $y = 1, 2, \ldots, n_s$ compute

$$F(s_i, s_y, z_{jk}) + \beta \tilde{E} v(s_y, z'),$$

then choose the one that gives the largest value. This will prove to be a very inefficient way to solve the model and will be remedied shortly.

3.2.5 The Concavity of the Value Function

It was shown in Theorem 2.14 that the objective of (3.6) is a strictly concave function (of s') as long as v is weakly concave. We will now exploit this concavity to reduce the number of points over which we need to search for the maximum.

A strictly concave function of one variable, defined on a closed interval, will have a unique global maximum on that interval. This maximum can occur at either the left or right endpoint, or in the interior. Algorithm 3.2 provides a binary search method to compute

Algorithm 3.1 Grid Search I

0. Initialization.
   (a) Specify a $\tilde{p}$ function.
   (b) Set up grids as discussed above.
   (c) Specify $\varepsilon > 0$ and a positive integer M. Set $m = 0$.
   (d) Set $v^0(i,j,k) = u(\overline{d} + e)$ and $g^0(i,j,k) = 0$, for all i, j and k.
1. For each (i,j,k), find i^* such that
$$s_{i^*} = \underset{s' \in \{s_i\}}{\operatorname{argmax}} \left\{ F(s_i, s', z_{jk}) + \beta \tilde{E} v^0(s', z') \right\}$$
2. Store i^* as $g^1(i,j,k)$ and $F(s_i, s_{i^*}, z_{jk}) + \beta \tilde{E} v^0(s_{i^*}, z')$ as $v^1(i,j,k)$.
3. If $\|g^1 - g^0\| = 0$, then $m = m + 1$. Otherwise $m = 0$.
4. If $\|v^1 - v^0\| < \varepsilon$ and $m \geq M$, stop. Otherwise set $v^0 = v^1$ and $g^0 = g^1$ and return to Step 1.

the maximum of a strictly concave function f with known values at n points, $x_1, \ldots, x_n$. Algorithm 3.2 is essentially Algorithm 1.2.1 in Heer and Maussner (2005). The algorithm assumes that $n \geq 4$. If not, then we resort to a simple linear search method instead. It can be shown that Algorithm 3.2 will take at most $\log_2 n$ function evaluations to reduce the reduce the choice space to three points.

Algorithm 3.2 Compute the Maximum of a Concave Function

0. If $f(x_1) > f(x_2)$, the max occurs at x_1, stop. If $f(x_n) > f(x_{n-1})$, the max occurs at x_n, stop. Otherwise, proceed.
1. Set $i_{min} = 1$ and $i_{max} = n$.
2. Set $i_l = \left\lfloor \dfrac{i_{max} + i_{min}}{2} \right\rfloor$ and $i_u = i_l + 1$.
3. If $f(x_{i_u}) > f(x_{i_l})$, set $i_{min} = i_l$. Otherwise set $i_{max} = i_u$.
4. If $i_{max} - i_{min} = 2$, stop and choose the maximum of $f(x_{i_{min}})$, $f(x_{i_{min}+1})$ and $f(x_{i_{max}})$. Otherwise return to step 2.

3.2.6 The Monotonicity of the Policy Function

It was shown in Theorem 2.21 that the policy function is a strictly increasing function of s. We can also use monotonicity to reduce the number of points which we need to search over.

To illustrate the idea we use, consider the following simple example. Suppose that we have $n_s = 100$ and fix j and k. Suppose that when $i = 60$ the agent finds $i^* = 30$. This means that for the given (i, j, k) the optimal holdings in that state of the world is s_{30}. Now suppose $i = 61$ and j, k remain fixed. Since we have ordered our grids from smallest to largest we know that $s_{61} > s_{60}$. Using the fact that the policy function is increasing in s we can conclude that $s' \geq s_{30}$. Note that Theorem 2.21 guarantees us *strict* monotonicity, but since we are restricting the agent's choice to be on the grid we cannot guarantee strict. The best we can do is know that the discrete policy function will not be decreasing. Algorithm 3.3 outlines how to use monotonicity.

Algorithm 3.3 Using monotonicity to reduce the space

1. Fix d_j and p_k.
2. For each $i = 1, \ldots, n_s$, if $i = 1$ set $\underline{i} = 1$, otherwise set $\underline{i} = g(i-1, j, k)$. Set $\bar{i} = n_s$.
3. Search for $i^* \in [\underline{i}, \bar{i}]$ to maximize v. Store the maximum value as $v(i, j, k)$ and $i*$ as $g(i, j, k)$.

3.2.7 Howard's Improvement

Even with the above improvements in Algorithm 3.1, the majority of the time to compute a policy function is spent searching for the maximum. The Howard Improvement Algorithm attempts to minimize the number of maximizations that need to be performed while still having the value function converge to its true value (and thus the policy function as well).

The idea works as follows. The agent computes his value function and corresponding policy function once. Then, holding the policy function fixed, he iterates the value function alone. In other words, we treat the policy function as if it were the optimal policy, and compute the corresponding optimal value function. Then, we compute a new optimal policy and repeat. This can be thought of as an inner loop in Algorithm 3.1. For additional discussion on the Howard Improvement Algorithm, see Ljungqvist and Sargent (2000).

Algorithm 3.4 Howard's Improvement

1. Input: A policy g, a value function v^0 and the number of iterations, H.
2. Set a counter, $h = 0$.
3. For each (i, j, k) set $s' = g(i, j, k)$ and compute
$$v^1(i, j, k) = F(s, s', z_{jk}) + \beta \tilde{E} v^0(s', z').$$
4. Set $v^0 = v^1$ and $h = h + 1$.
5. If $h < H$ return to step 3. Otherwise return v^1.

3.2.8 An Improved Grid Search Algorithm

With the above improvements we now state a much better grid search in Algorithm 3.5. Although the new algorithm will be much faster, it still suffers from the fact that the agent's choice is constrained to a grid. The result will be a step like function, a problem that will be addressed in the next section.

Algorithm 3.5 Grid Search II

0. Initialization.
   (a) Specify a $\tilde{p}$ function.
   (b) Set up grids as discussed above.
   (c) Specify $\varepsilon > 0$ and positive integers M and H. Set $m = 0$.
   (d) Set $v^0(i, j, k) = u(\overline{d} + e)$ and $g^0(i, j, k) = 0$, for all i, j and k.
1. For each (i, j, k):
   (a) Set up $[\underline{i}, \overline{i}]$ using Algorithm 3.3.
   (b) Search for i^* using the binary search in Algorithm 3.2.
2. Store i^* as $g^1(i, j, k)$ and $F(s_i, s_{i^*}, z_{jk}) + \beta \tilde{E} v^0(s_{i^*}, z')$ as $v^1(i, j, k)$.
3. Pass g^1, v^1 and H to Algorithm 3.4. Receive back improved v^1.
4. If $\|g^1 - g^0\| = 0$, then $m = m + 1$. Otherwise $m = 0$.
5. If $\|v^1 - v^0\| < \varepsilon$ and $m \geq M$, stop. Otherwise set $v^0 = v^1$ and $g^0 = g^1$ and return to Step 1.

3.3 Continuous Choice Variable

In Algorithm 3.1 presented in the previous section, the agent may only choose s' from the grid of possible s values. This presents two main problems. First, it is likely that, given the opportunity, the agent would in fact prefer to choose a value of s' between two grid points. The result would be a higher level of the objective function. In a sense the agent does a better job of computing his value function (and therefore his policy function too) if we allow him to choose off the grid.

The second problem that arises is the unnatural step-like structure of the policy function if the choices are discrete. Intuitively, one would expect that an increase in price should certainly lead to a decrease in demand. The expected decrease in demand, however, may not occur in the discrete case. When the gird is discrete, there could be many prices for which the agent will demand the same amount of the asset (see Figure 3.1 in Section 3.4). Failure to have a strictly decreasing demand function will cause problems when we attempt to equate supply and demand to find the market clearing price. It should be the case that the market maker can adjust the price up or down in order to see the aggregate demand equal the aggregate supply.

To see why the step-like demand function is undesirable, consider the following extreme example. Suppose there are two agents and two shares of the stock. At all prices higher than 10 the agents demand 1.5 shares each and at all prices equal to or lower than 10 they demand 0.5 shares. There will not exist any price for which total demand equals the total supply of 2. Now, consider a second example. At all prices higher than 12 the agents demand 1.5 shares, for prices between 8 and 12 they demand 1.0 and for all prices below 8 demand is 0.5. In this case the market will be able to clear, but at which price? There does not exist a unique market clearing price since any price between 8 and 12 will lead to a total demand of 2. These situations would not arise in practice since our p grid would be much larger than 2 or 3, but they can (and will) happen at some level, eventually.

To eliminate the problems with the discrete grids we need to allow the agents to choose any value of s' they wish (within the upper and lower bounds). We do so by introducing splines. First, we perform a discrete grid search using Algorithm 3.5. Now, for a given d and p we have the value function defined on a grid of s values. By fitting a spline to this data we can maximize the resulting *continuous* function and choose any s' we wish, on the grid

or not. This idea raises several issues, which we will discuss now.

The addition of splines raises several issues. First, a new spline needs to be constructed for each d and p pair. Computing this many splines is sure to add significant computing time to an already sluggish algorithm. Although it is true that the addition of splines will increase computing time, it is more than worth it given the problems that arise from not having a continuous choice variable. Splines add the advantage of being able to decrease the number of points we use in the s direction. Since we will interpolate between them with a polynomial, we can safely allow the distance between them to be somewhat larger without losing much accuracy, for we know the function will be well behaved between points and thus will likely resemble a low order polynomial anyway.

Next, we need to decide what type of spline is appropriate. The easiest choice is linear, but a bad one since much of what we know about the solution to the problem relies on the strict concavity of the objective. Linear functions are not strictly concave. Also, using linear splines will cause the value function to fail to be differentiable.

Another option is the popular cubic spline. The cubic spline is relatively easy to implement and is quite smooth. Unfortunately cubic splines can fail to preserve concavity.

For problems such as ours, when the data to which we fit the curve to is known to have properties such as monotonicity or concavity, the best choice of spline is a quadratic shape preserving spline. Of course we will not obtain a fit that is as good as if we had used the cubic spline, but we make up for this by the guarantee of shape preservation. The standard reference for these splines is Schumaker (1983). The main results from that paper will be presented here without proof. We will first describe how to solve for the quadratic spline on a single interval, then present the general algorithm.

The Schumaker spline problem is as follows. We wish to find a quadratic function f on the interval $[t_1, t_2]$ such that[2]

$$f(t_i) = z_i \text{ and } f'(t_i) = s_i, \text{ for } i = 1, 2, \tag{3.7}$$

where z_i and s_i are given. The following result is straightforward to show and treats the case when a simple quadratic polynomial will interpolate the data.

[2]The use of a prime $'$ has two meanings in dynamical programming problems. A prime on a variable is used to indicate the next period's value, while a prime on a function of a single variable is used to indicate the derivative.

Lemma 3.1. *If* $\frac{s_1+s_2}{2} = \frac{z_2-z_1}{t_2-t_1}$ *then* $f(t) = z_1 + s_1(t-t_1) + \frac{(s_2-s_1)(t-t_1)^2}{2(t_2-t_1)}$ *satisfies* (3.7).

In general the previous lemma will not apply. Instead we need to use a third point in (t_1, t_2) to construct f. The following result appears as Lemma 2.3 in Schumaker (1983) and shows how to carry out this construction.

Lemma 3.2. *For any* $\xi \in (t_1, t_2)$ *there is a unique quadratic spline that satisfies* (3.7) *with a knot at* ξ. *This spline is given by*

$$f(t) = \begin{cases} A_1 + B_1(t-t_1) + C_1(t-t_1)^2, & t \in [t_1, \xi] \\ A_2 + B_2(t-\xi) + C_2(t-\xi)^2, & t \in [\xi, t_2] \end{cases} \tag{3.8}$$

where $A_1 = z_1$, $B_1 = s_1$, $C_1 = \frac{\overline{s}-s_1}{2\alpha}$, $A_2 = A_1 + \alpha B_1 + \alpha^2 C_1$, $B_2 = \overline{s}$, $C_2 = \frac{s_2-\overline{s}}{2\beta}$, $\overline{s} = \frac{2(z_2-z_1)-(\alpha s_1+\beta s_2)}{t_2-t_1}$, $\alpha = \xi - t_1$ *and* $\beta = t_2 - \xi$.

It remains to decide how to choose the knot, ξ. Schumaker shows that choose ξ in the following to preserve the concavity of the data: Compute the average slope and denote it as $\Delta = (z_2-z_1)/(t_2-t_1)$. If $|s_2-\Delta| < |s_1-\Delta|$ then choose ξ to satisfy

$$t_1 < \xi \le \overline{\xi} \equiv t_1 + \frac{2(t_2-t_1)(s_2-\Delta)}{s_2-s_1}, \tag{3.9}$$

and if $|s_2-\Delta| > |s_1-\Delta|$ then choose ξ to satisfy

$$t_2 + \frac{2(t_2-t_1)(s_1-\Delta)}{s_2-s_1} \equiv \underline{\xi} \le \xi < t_2. \tag{3.10}$$

We now turn to the general problem. Suppose we have data $\{(t_i, z_i)\}_{i=1}^n$. In general we will not have slope data so we will need to estimate the slopes. Schumaker suggests the following formulas:

$$L_i = \left[(t_{i+1}-t_i)^2 + (z_{i+1}-z_i)^2\right]^{1/2} \quad \text{and} \quad \Delta_i = \frac{z_{i+1}-z_i}{t_{i+1}-t_i}, \quad i = 1, 2, \ldots n-1,$$

so that

$$s_i = \begin{cases} \frac{L_{i-1}\Delta_{i-1}+L_i\Delta_i}{L_{i-1}+L_i} & \text{if } \Delta_{i-1}\Delta_i > 0 \\ 0 & \text{otherwise} \end{cases}, \quad i = 2, \ldots, n-1, \tag{3.11}$$

and

$$s_1 = \frac{3\Delta_1 - s_2}{2}, \quad s_n = \frac{3\Delta_{n-1} - s_{n-1}}{2}. \tag{3.12}$$

Algorithm 3.6 Schumaker Spline

0. Input: (t_i, z_i), $i = 1, 2, \ldots, n$.
1. Compute the estimated s_i, $i = 1, 2, \ldots, n$ using (3.11) and (3.12).
2. For each $i = 1, \ldots, n$, if $|s_{i+1} - \Delta_i| < |s_i - \Delta_i|$ then compute $\overline{\xi}_i$ using (3.9) and set $\xi_i = (1/2)(t_i + \overline{\xi}_i)$. Otherwise compute $\underline{\xi}_i$ using (3.10) and set $\xi_i = (1/2)(t_{i+1} + \underline{\xi}_i)$.
3. Use (3.8) to evaluate the function at any $t \in [t_1, t_n]$.

We summarize the above steps with Algorithm 3.6.

Now that we have a way to allow the agent to choose s' in a continuous manner, we need to redefine what it means to have the policy functions converge. Previously, since all the choices were discrete we could say the functions converged if they hadn't changed for some number of periods. Since the policy functions can be any real value now we redefine convergence of the policy functions as:

C2′: Convergence of the policy functions (continuous). Let g^n denote the policy computed at iteration n and specify positive integer M and $\delta > 0$. We will say that the policy functions have converged if $\|g^n - g^{n-1}\| < \delta$ for M consecutive iterations.

Finally, we will need an algorithm to find the maximum of a function of a continuous variable. Since we are using shape preserving splines we know that our function will have a unique maximum, just as before. The following algorithm appears in Heer and Maussner (2005) as Algorithm 8.6.1 on page 468.

Algorithm 3.7 Golden Section Search

0. Input: A single peaked function $f(x)$, bounds $\underline{x}$ and $\overline{x}$ and a tolerance $\varepsilon > 0$.
1. Set $A = \underline{x}$ and $D = \overline{x}$. Compute $\rho = \dfrac{\sqrt{5} - 1}{2}$, $B = \rho A + (1-\rho)D$ and $C = (1-\rho)A + \rho D$.
2. If $f(B) > f(C)$, set $D = C$, $C = B$ and $B = \rho C + (1 - \rho)A$. Otherwise set $A = B$, $B = C$ and $C = \rho B + (1 - \rho)D$.
3. If $|D - A| < \varepsilon$, stop and return B. Otherwise return to the previous step.

3.4 Numerical Results

In this section we will validate the numerical results obtained in a special case using a known analytical formula. Specifically, in the case when $\gamma = 1$ and $e = 0$ the agent's policy function is given by

$$g(s, d, p) = \beta \left(1 + \frac{d}{p}\right) s. \tag{3.13}$$

One can easily check the validity of (3.13) substituting it into the Euler Equation, (2.13), and verifying the equality holds. Since we know that there is a unique solution to the problem, this must be it. As we will see repeatedly, this case is quite special. No such simple formula exists for any other choice of γ and e. One surprising property of (3.13) is that it is actually independent of the agent's price forecasting rule, $\tilde{p}$. One would expect that the agent's demand for the stock would depend a great deal on what he believes the price will be in the future.

We will now specify the default values for our model parameters. As specified above we will use $\gamma = 1$ (log utility) and $e = 0$. We take $\beta = 0.9$. This may be slightly lower than data suggests it should, but since the contraction mapping is of modulus β, the smaller its value the quicker the convergence. Dividends will be assumed to follow a 2 state, iid process. Specifically, we assume that $d \in \{0.75, 1.25\}$ with equal probability of either state. Adding more possible dividends only adds additional points to the state space and thus more computing time. Nothing is lost by assuming a 2 state process. For the other grids we use 201 points[3] in the s and p directions, with $s \in [0.01, 1.99]$[4] and $p \in [1, 30]$.

We first run the code without exploiting the structure of the model at all. That is, for every state, (s, d, p), we check every possible choice of s' and choose the largest value. We also do not use the Howard Improvement Algorithm or splines yet. Table 3.1 shows the time required to compute the exact same policy function when we use various tricks. The times are reported in minutes and seconds. The amount of time saved by employing these methods is remarkable. Since the agent will need to carryout these computations repeatedly it is crucial that it be as fast as possible.

[3] It is convenient to use 201 instead of 200 so that the step size works out to a simpler number. For example, if we have $s \in [0, 2]$ then using $n_s = 201$ gives $\Delta s = (2 - 0)/(201 - 1) = 0.01$. This simply makes for a resulting grid that is easier to write down.

[4] We use this interval instead of $[0, 2]$ so that we do not violate the boundedness restrictions on c and s discussed in Chapter 2.

Table 3.1: Computing Times without Howard's Improvement or Splines

Case	Monotonicity	Concavity	Time
1	No	No	68:47
2	Yes	No	9:17
3	No	Yes	9:14
4	Yes	Yes	5:38

Next, we wish to investigate how much the Howard Improvement Algorithm (HIA) helps. We still solve the problem in a strictly discrete environment, without the use of splines. As discussed above, the HIA works by holding the policy function fixed and iterating on the value function some number of times. The idea is that this will allow us to converge to the true value function faster, because it removes the most costly step - the search for the maximum. There is a trade off here though. Each iteration in the HIA gets us closer to the true solution, but it also adds computation time. There is only so much benefit that can be obtained from the HIA, and after a certain point to continue will just waste time with little to no improvement.

Table 3.2 shows the computing times for various choices of the number of Howard iterations used. Note that there are two notions of iteration here. There is the larger scale iteration where, given an entire function v^0, we compute v^1. Then there is an inner iteration that performs the HIA. More inner iterations (Howard iterations) will lead to a larger time for the outer iteration, but should reduce the total number of outer iterations needed. It can be seen that for a while the addition of Howard iterations does help, but then the benefit of adding more drops off. Case 4 is the best in terms of total computing time. This may not always be the best case for other parameter values but the difference shouldn't be too drastic (as indicated by Table 3.2). The standard choice is to use between 5 and 10 Howard iterations, so we will choose to use 8 from here on.

Now that we have numerically demonstrated the usefulness of the various tricks we can use to speed up the computation, we will now examine some alternative grid sizes. Clearly we wish to use the smallest number of grid points for which we can expect a reasonable approximation. The d grid will remain fixed at 2 values for this entire chapter. We run the routine (using all previous tricks) with equally sized s and p grids. We use sizes of 51, 101, 201, 301 and 401 points. We measure closeness by computing the maximum relative error

Table 3.2: Computing times with HIA and no splines

Case	Howard Iterations	Time	Iterations	Time/Iteration
1	0	5:38	115	2.94 s
2	3	2:08	34	3.76 s
3	5	1:54	25	4.56 s
4	8	1:48	21	5.14 s
5	10	2:00	21	5.71 s
6	15	2:07	18	7.06 s
7	20	2:40	17	9.41 s

compared the true solution for a fixed s and d and for all p. Specifically we choose $s = 1$ and $d = 0.75$.[5] The reason that we do not compute the norm over all s and d is that for values of s near the endpoints the true solution could easily fall outside of the s grid. In this case the norm will not make sense since the approximate solution will be constrained to the grid. Another reason for computing the norm over all p is that it is very natural to think of the demand function as a function of p alone.

The results of computing the demand function for various grid sizes are shown in Table 3.3. We find that the errors decrease as the number of grid points is increased. This is to be expected since in each case the agent has more choices to make for s' and therefore the resulting value function will be at least as large as it was with fewer points. The convergence is shown to be first order in Table 3.3. That is, if we double the number of points the error decreases by a factor of 2.

Table 3.3: Maximum absolute relative errors for the discrete case

n_s	n_p	Error
51	51	0.0210
101	101	0.0108
201	201	0.0053
301	301	0.0036
401	401	0.0027

It is important to realize the implications of Table 3.3. It suggests that there is a limit to how much accuracy we can reasonably expect to obtain from running our numerical procedure. While it does appear that increasing the grid size will continue to improve the

[5]Note that $s = 1$ corresponds to the agent's holdings for the representative agent case.

approximation the improvement is very slow and very costly. For example, in the cases given in Table 3.3 it takes just over five minutes to compute the 301-by-301 case and ten and a half minutes for the 401-by-401 case. The increase in computing time far outweighs the improvement in the accuracy. It is sometimes tempting to believe that we can obtain arbitrary accuracy in a method like this by making the grid sizes larger and larger and reducing the tolerance. In reality though, an error of zero is practically impossible. For a strictly discrete problem we will typically begin with a grid size of 201-by-201 and increase it if we feel it is necessary. It would be necessary to increase the grid size if the range of s values or p values was exceptionally large. For example, we generally take $s \in [s_\varepsilon, N]$, where N is the number of agents (and shares). If we had many agents then this interval would get bigger and therefore a larger number of points should be chosen. For most of this paper we will use splines in the s direction and either 201 or 301 points in the p direction. The number of s points needed will be discussed below.

It remains to show what happens when we add splines. As we discussed previously, the addition of splines will certainly add computing time to the algorithm. This addition, however, will be partially offset by the need for fewer points in the s direction. We begin by running the 201-by-201 and 301-by-301 cases, but with splines included. We now use stopping rule $C2'$ with $\delta = 0.01$ and $M = 2$. This will be shown to increase the computation time and decrease the maximum error. We then demonstrate that we can reduce the number of points in the s direction substantially without incurring additional error. Note that we use the notation $a(b)$ to mean $a \times 10^b$.

Table 3.4: Maximum absolute relative errors for the spline case

n_p	n_s	Error	Time
301	301	3.92(-6)	9:14
301	201	8.41(-6)	4:43
301	101	3.02(-5)	1:47
301	51	1.16(-4)	0:47
201	201	3.91(-6)	2:59
201	101	3.01(-5)	1:10
201	51	1.17(-4)	0:30

Table 3.4 shows that we can substantially decrease our computation time by decreasing

the number of grid points in the s direction. Doing so results in a larger value for the error, but the increase is fairly minor compared to the savings in time. For example, with 301 points in the p grid, going from $n_s = 201$ to $n_s = 101$ saves nearly three minutes, but the error remains small. Since the agent's demand for shares will generally be in the neighborhood of 1 or 2, an increase in the error from 10^{-6} to 10^{-5} is not going to make much difference. Note that changing the p grid and holding fixed the s grid does not affect the error. This is due to the fact that the agent fixes p and d and makes his choice for s'. We then compute the error over all p. The importance of the p grid will become clear in Chapter 4 when we discuss market clearing.

In Table 3.5 we compare the errors for several grid sizes and various δ values. We find that the error can be decreased by either refining the grid, or by lowering the tolerance on the stopping rule. Note that values of δ that are larger than 0.001 will result in the same relative errors since the spline approximation requires very few iterations to have a very good approximation. For example, in the 201-by-201 case after only two iterations the norm on the policy functions is already 10^{-4}, and thus there will be no difference between $\delta = 0.001$ and $\delta = 0.01$. In the 301-by-301 case the policy functions are found to have a norm of less than 10^{-4} immediately so no change results from decreasing δ from 10^{-3} to 10^{-4}. Since the improvement is relatively minor as we decrease δ, we will continue to use $\delta = 0.001$ for the remainder of the chapter.

Table 3.5: Maximum absolute relative errors for various δ values

n_p	n_s	$\delta = 10^{-3}$	$\delta = 10^{-4}$	$\delta = 10^{-5}$
301	301	3.61(-6)	3.61(-6)	2.91(-6)
201	201	8.17(-6)	7.73(-6)	7.73(-6)
201	101	3.03(-5)	3.03(-5)	3.02(-5)
201	51	1.18(-4)	1.17(-4)	1.17(-4)

Figure 3.1 shows a discrete approximation to the known, analytic solution. A very coarse grid was used so that the step like nature of the approximation is clear. The approximation was generated with $n_s = n_p = 41$. The spline approximation, even with very few grid points, yields a plot that is too close to the true solution to distinguish the two.

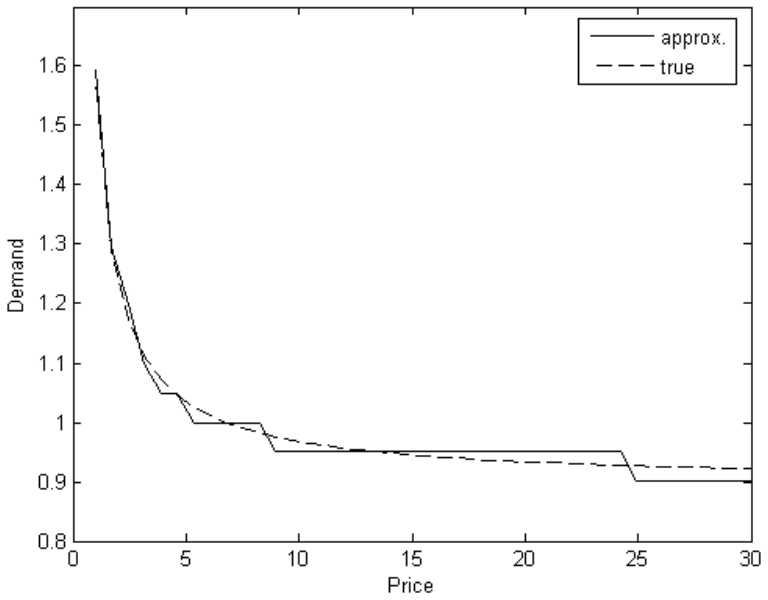


Figure 3.1: A 41x41 discrete approximation of the demand function with log utility and no endowment.

3.5 Properties of the Demand Function

In this section we explore some of the properties of the demand function in the case when we know an analytical solution and the case when we don't. The most unusual property that is present in Figure 3.1 is the existence of an asymptote for large p. The asymptote can be computed in the analytic case by

$$\lim_{p\to\infty} g(s,d,p) = \lim_{p\to\infty} \beta s \left(1+\frac{d}{p}\right) = \beta s.$$

Since we fixed $s = 1$ in Figure 3.1, we see the asymptote is $\beta = 0.9$. An asymptote at β means in that there is no price for which the agent will choose s' less than β - a very surprising result. Selling the stock at a high price would result in the agent making a very large profit. The problem, however, is that the agent has no way to save (in a risk-free way) the wealth he has now realized. The only choice the agent has is to consume or invest. Since he has no endowment, his entire future wealth is determined by how much stock he chooses to hold today. The additional utility he would receive by consuming more today (no matter how large) will not outweigh the penalty he faces for the rest of his (infinite) life. The addition of an endowment will lower the asymptote and, eventually, it will not exist (due to the no

short sales restriction). The reason is that with an endowment in each period the agent is guaranteed to always be able to consume a positive amount, no matter how many shares he holds. Thus, if the price is right he can choose to sell all his holdings and know that he will not face death tomorrow. Figure 3.2 shows the policy function for log utility and many different endowments. As before we have fixed $s = 1$ and $d = 0.75$. We can hypothesize, and demonstrate numerically, that the asymptotes in the above figure are given by $\beta - (1 - \beta)e$. It can be seen from this formula that the asymptote decreases with endowment.

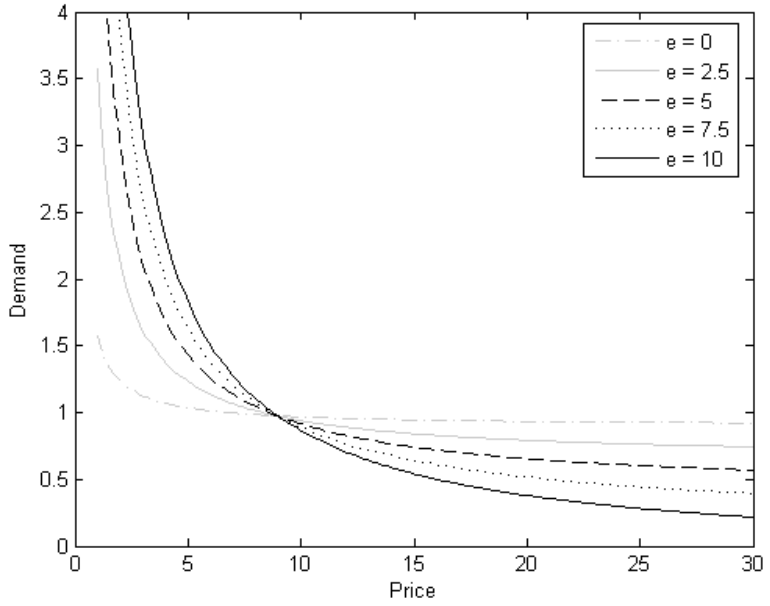


Figure 3.2: Plots of demand with various endowments and log utility.

The asymptotic behavior changes completely for the case when $\gamma \neq 1$. For $\gamma < 1$ the asymptote decreases to 0, even in the case when $e = 0$. The closer γ is to 0 the lower the price the agent requires to sell his holdings. This is due to the fact that for $\gamma < 1$ we have $u(0) = 0$, but for $\gamma \geq 1$ we have $u(0) = -\infty$. Thus, when $\gamma < 1$ the agent may, in some cases, choose holdings that will lead to $c = 0$ in the future. In the extreme case, when $\gamma = 0$, the agent is actually risk neutral and is completely indifferent between receiving an infinite stream of (risky) payments or receiving their expected, discounted value today. Thus, the risk neutral agent will choose to sell his entire holdings for any price that exceeds the infinite, discounted, expected payment. As γ increases toward one the graph of s' will become more

and more flat. It will, however, eventually go to zero for very large prices. Figure 3.3 shows the demand function for several values of $\gamma < 1$.

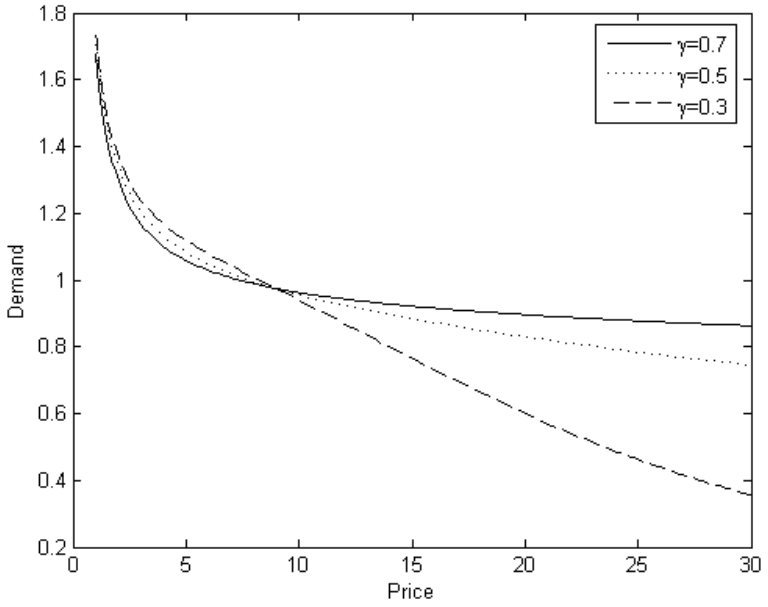


Figure 3.3: Some plots of the demand function for various values of $\gamma < 1$.

For the case when $\gamma > 1$, the situation is even more unexpected. The asymptote turns out to be 1 in this case. In the case where $\gamma \leq 1$, the demand function can be seen to be a decreasing function of p. This property fails in the $\gamma > 1$ case. Indeed, the function will decrease below one for a while, but then it will eventually turn back and approach one for large p. The reason is that when $\gamma > 1$ the agent's marginal utility of consumption (i.e. $u'(c)$) decreases faster than when $\gamma \leq 1$. Thus, as the agent consumes more, each additional unit is less satisfying, and this is happening at a fast rate. Therefore, for large p, the agent will sell shares for a while for the purpose of consuming more of the good. He will reach a point, however, at which he nearly refuses to consume more. At this point he would begin to consume less and hold on to more shares. It is helpful to keep in mind that the agent's wealth is not determined until the market clearing price is revealed. Thus, to understand how this demand function works we can imagine the agent hearing prices called out, each one greater than the last. For a while he will choose to sell his shares and consume more. After a point the price will be so high that he can actually maintain the previous level of consumption without having to sell as much. Figure 3.4 shows the graph of the policy function for several

$\gamma > 1$. Figure 3.5 uses $\gamma = 3$ and $\beta = 0.8$ to display the backward curve of the demand function.

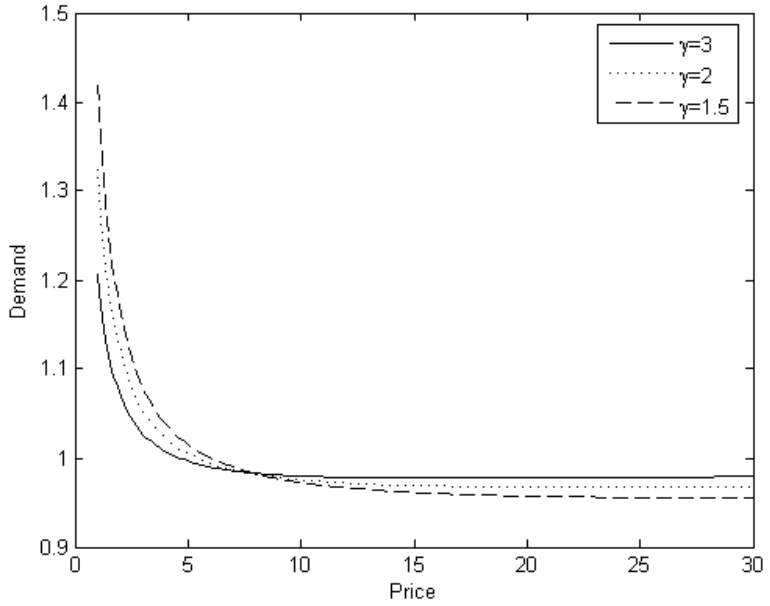


Figure 3.4: Some plots of the demand function for various values of $\gamma > 1$.

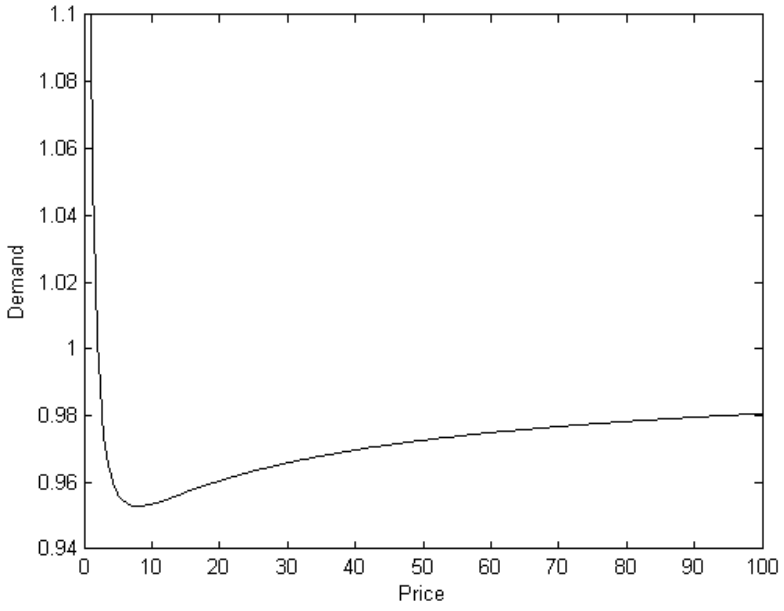


Figure 3.5: Plot of the demand function for $\gamma = 3$ and $\beta = 0.8$ to show the backward bending shape.

3.6 Summary

In this chapter we have outlined how a single agent will solve their optimization problem via value function iteration. We are able to exploit certain properties of the value function and the resulting demand function to speed up the computations. It was shown that using the Howard Improvement Algorithm can lead to a faster convergence as well. The addition of quadratic, shape-preserving splines allows us to move away from a discrete choice space so that the agent is able to choose holdings anywhere in the feasible range of values, not just at grid points. This makes for smoother policy functions, a property that will prove useful when the market clearing price is calculated. We were able to show, in Section 3.4, by comparison to a known analytical solution, that the algorithm does work correctly. Finally, some properties of the demand function were discussed for various cases.

CHAPTER 4

COMPUTATIONAL METHODS II: LEARNING, MARKET CLEARING AND CONVERGENCE

In this chapter we will discuss the computational methods required at the aggregate, or market level. Our ultimate goal is to examine conditions under which agents are able (or not able) to learn the Rational Expectations Equilibrium (REE) pricing function. To do so we begin with a definition of an REE as well as how to compute it in a special case. We are able to show, analytically, that agents will converge asymptotically to the REE pricing function when all agents have log utility and no endowment. The agents with the largest discount factor are shown to survive while the others die off. This case is special since it does not require the agents to learn. We will specify a learning rule to be used throughout this chapter and the next and choose several cases for which we know the true REE pricing function to show that the agents are able to learn it.

4.1 Rational Expectations

In asset pricing models, agents are ultimately interested to know the market clearing price function since then the only remaining uncertainty is in the aggregate variables (which the agents may be able to forecast). Since prices are aggregate variables (i.e. the same for everyone), the market clearing price function must only depend on aggregate states. Let us denote the set of *all* aggregate states at time t by Θ_t. Included in Θ is the dividend and price, the wealth distribution of the agents ($\mathcal{S}_t$), the distribution of the dividends, the total supply of the stock (N) and the distributions of all parameters over all agents. Similarly, let us denote by θ_t^i the individual states for agent i. In this model, the individual states are the agent's stock holdings and his endowment (which is constant, but not known to other agents). Note that, in general, agents will not know Θ since it is composed of public and

private variables. We now wish to define a very important equilibrium concept.

Definition 4.1. *A Rational Expectations Equilibrium (REE) is a set of demand functions* $\{c^i(\theta_t^i, \Theta_t), s^i(\theta_t^i, \Theta_t)\}_{i=1}^N$ *and a pricing function* $p(\Theta_t)$ *such that the following conditions hold:*

1. *These functions solve each agent's optimization problem (i.e. the first order conditions hold).*

2. *All markets clear:*

   (i) $\sum_{i=1}^{N} s_t^i = N, \quad \forall t$

   (ii) $\sum_{i=1}^{N}(c_t^i + p_t s_{t+1}^i) = \sum_{i=1}^{N}(s_t^i(p_t + d_t) + e^i), \quad \forall t,$ *where* $p_t = p(\Theta_t)$.

Let pause briefly to discuss what Definition 4.1 does and does not say. An REE, as defined above, is a set of demand functions and a pricing function, such that if the agents were given those functions, their optimization problems would be perfectly solved. In particular, all agents would have maximum utility and they would correctly predict the market clearing price in each period. That is, $p_t = p(\Theta_t)$, for all t.

It is important to realize that Definition 4.1 does not provide a way for us to compute the demand functions and pricing functions that are given. It simply says that *if* such functions were given to the agents, then they would perfectly solve the agents problem. In general, computing the REE pricing and demand functions is very difficult.

It is worth noting that we are free to ignore many of the variables included in Θ. The agents are assumed to know the distribution dividends and thus it need not be considered as a variable. Also, the total supply of the stock and the distribution of parameters for the agents (e.g. discount factors) are fixed for all periods. These constants will surely affect the REE functions, but they are parameters, not variables. Thus, all that remain as true variables are the dividend and the wealth distribution of the agents. Since the wealth distribution is not, in general, known to the agents, they will not be able to include it as a variable in their own price forecasting rule. We will discuss this complication at the conclusion of this section.

Now that we have defined an REE, we turn our attention to how agents might compute it in a special case. The REE pricing function is, in general, difficult to compute since to do so requires the agents to know more than they usually would. Specifically, the agents are

required to know the aggregate wealth distribution. Unlike the dividend, which is called out to the entire market, the wealth distribution is composed of many private variables. The wealth of investors is not public knowledge and therefore it is not reasonable to expect our agents to know $\mathcal{S}$. The exception would be a very special case when the wealth distribution remains constant over time. In this case the REE pricing function will depend on d alone. We now consider this special case.

Let us begin with the simplest possible market. Suppose that the market is composed of N *identical* agents and assume there are N shares of the stock. The assumption that agents are identical applies to all aspects of the agents. Specifically, agents have the same discount factor, level of risk aversion, endowment, initial holdings and price forecasting rule. In this homogeneous case, agents will never trade, since, if one agent wished to buy, then all agents will wish to buy also. In order for a trade to occur, there must be an agent to buy and another to sell. Since there are N agents and N available shares, and agents are identical, each agent will hold one share in all periods. Thus, we have constructed a market in which $\mathcal{S}_t = (1, 1, \ldots, 1)$ for all t. Since agents are identical, we are able to suppress the superscript i in discussion below.

Given that agents never trade, the budget constraint tells us they will consume the dividend plus their constant endowment each period. Therefore, we are able to replace c_t by $d_t + e$ in the Euler equation, for all t, to obtain (from (2.13)):

$$\frac{p(d)}{(d+e)^\gamma} = \beta E\left[\frac{p(d') + d'}{(d'+e)^\gamma}\right], \tag{4.1}$$

where $p(\cdot)$ is the REE pricing function. Suppose $p(d) = A(d+e)^\gamma$ for some constant A and all d. Section 4 of Lucas (1978) shows that the REE pricing function is unique in this model and therefore, if we can find a constant A that satisfies (4.1), it must be the correct one. Define $\Gamma(e, \gamma) = E\left[\frac{d}{(d+e)^\gamma}\right]$. Note that since we have assumed d is *iid*, Γ is a constant. Now, plugging our specific $p(d)$ into (4.1) we have

$$\frac{A(d+e)^\gamma}{(d+e)^\gamma} = \beta E\left[\frac{A(d'+e)^\gamma + d'}{(d'+e)^\gamma}\right]. \tag{4.2}$$

This can easily be solved to obtain

$$p(d) = \frac{\beta}{1-\beta}\Gamma(e, \gamma)(d+e)^\gamma \tag{4.3}$$

The above setup is indeed quite special. Agents have been assumed to be identical in every way, clearly a very strong assumption. As modelers we know that since all agents are identical then there will never be trading. Therefore, we are free to substitute for c_t and simplify the Euler equation. If the agents were to know that they were all identical, then they too could perform this substitution and immediately solve for the REE pricing function. It is not reasonable, however, to assume that agents know anything about other agents. Without this knowledge they cannot solve the model as we have here.

Note that the pricing function that we obtain in (4.3) only requires that the dividends be *iid*. There is nothing at this point that requires the dividends be drawn from a continuous or discrete distribution. Therefore, (4.3) holds for *any* dividend. In the cases we will study in this manuscript we will always take dividends to be drawn from a discrete distribution, but we do so out of convenience, not necessity.

We now return to the assumption that agents are heterogeneous, or at least that they believe they could be. The discussion in this section shows that boundedly rational agents (i.e. agents who have less than full information) are unable to compute the REE pricing function. It is the case, however, that in each period a market clearing price will be determined as a function of the current aggregate states. Therefore, it is reasonable to think that agents may be able to *learn* the REE pricing function by observing the market clearing prices, at least in some cases. Since agents will be unable to learn the aggregate wealth distribution (because, unlike prices, it is never revealed), they will only be able to successfully learn the REE pricing function in the cases when it is independent of the wealth distribution or when the wealth distribution reaches a steady state after some time. Investigating conditions under which the agent is able to learn the REE pricing function will be the main focus of the following chapter. In order to discuss agents learning the REE, we will need to define carefully how agents will learn. This will be the focus of Section 4.3. Before discussing learning, however, we can present a special case in which convergence to the REE occurs without learning.[1]

[1] Li (2007) considers a similar model, in continuous time.

4.2 A Special Case of Convergence

Recall from Chapter 3 that when $\gamma = 1$ and $e = 0$, the agent's policy function does not depend on the price forecasting rule, $\tilde{p}$. As we saw previously, this is a very special case. In this section we will be able to analytically prove that heterogeneous agents converge asymptotically to the REE. The result in this section also appears in Beaumont et al. (2007).

We define the model is as follows. Suppose we have N agents with $\gamma_i = 1$ and $e^i = 0$, for all i, and N total shares of the stock. Let s^i and β_i be the holdings and discount factor, respectively, for agent i. Recall that agent i has the following policy function

$$g^i(s^i, d, p) = \beta_i s^i \left(1 + \frac{d}{p}\right), \quad i = 1, \ldots, N. \tag{4.4}$$

Equating supply and demand, the market clearing price, p_m, is determined by

$$\sum_{i=1}^{N} \beta_i s^i \left(1 + \frac{d}{p_m}\right) = N, \tag{4.5}$$

which gives

$$p_m = \frac{\sum_j \beta_j s^j}{N - \sum_j \beta_j s^j} d. \tag{4.6}$$

Using (4.6) in the demand function gives the updated share holdings:

$$(s^i)' = g^i(s^i, d, p_m) = \beta_i s^i \frac{N}{\sum_j \beta_j s^j}. \tag{4.7}$$

Note that since $0 < \beta_i < 1$, for all i and $\sum s^i = N$ we have $0 < \sum_j \beta_j s^j < N$ so that p_m is always well defined and positive. Together, (4.6) and (4.7) define a dynamical system for prices and share holdings. The following theorem establishes the convergence of this system.

Theorem 4.1. *Suppose we have N agents with $\gamma_i = 1$ and $e^i = 0$, for all i. Let k be the number of agents who have the largest discount factor and order the agents by decreasing β so that*

$$1 > \beta = \beta_1 = \cdots = \beta_k > \beta_{k+1} \geq \cdots \geq \beta_N > 0.$$

Then the dynamical system given by (4.6) and (4.7) converges to

$$p^*(d) = \frac{\beta}{1 - \beta} d, \tag{4.8}$$

and

$$(s^i)^* = \begin{cases} \frac{N s_0^i}{s_0^1 + \cdots + s_0^k} & i \leq k \\ 0 & i > k \end{cases} \tag{4.9}$$

where s_0^i denotes agent i's initial holdings.

Note that setting $\gamma = 1$ and $e = 0$ in (4.3) gives (4.8) . This means that prices will converge to the REE prices. The asymptotic behavior of s^i given in the theorem has a very natural interpretation. The agent's discount factor, β, represents his level of impatience. A higher β means a more patient agent. For example, consider two agents with $\beta_1 = 0.9$ and $\beta_2 = 0.8$. To the first agent receiving one dollar tomorrow with certainty is worth 0.90 today, while it is worth 0.80 to the second agent today. All agents would prefer to consume today rather than wait and consume tomorrow. In this example, however, Agent 2 prefers consumption today more than Agent 1. In other words, Agent 1 is more willing to wait - he is more patient. It makes sense, then, that if these two agents were to be in the market together that Agent 2 would be willing to sell some of his holdings to Agent 1 in order to consume more today. Agent 1 is happy to accumulate this wealth knowing that it will lead to more consumption in the future. This will happen repeatedly until Agent 2 has less and less of his shares remaining. As time goes to infinity he will have no shares and Agent 1 will hold all of them.

4.2.1 Proof of Theorem 4.1

First, note that we assume in each period that markets clear. This means that the sum of holdings over all agents must always equal the total supply. Now, suppose that we have ordered the β's as in Theorem 4.1 and the share holdings have converged to (4.9). Then using (4.6) we have

$$p_m = \frac{\sum_1^N \beta_i s^i}{N - \sum_1^N \beta_i s^i} d = \frac{\sum_1^k \beta s^i}{N - \sum_1^k \beta s^i} d = \frac{\beta \sum_1^k s^i}{N - \beta \sum_1^k s^i} d = \frac{\beta N}{N - \beta N} d = \frac{\beta}{1 - \beta} d. \tag{4.10}$$

Thus, we need only prove that the share holdings converge to (4.9).

Define the *relative holdings* of agent i to be $x_i = s^i/N$. Then we can rewrite (4.7) as

$$x_i' = \frac{\beta_i x_i}{\sum_j \beta_j x_j}. \tag{4.11}$$

Clearly $x_i \in [0, 1]$ for all i and $\sum x_i = 1$. Thus, the state $(x_1, x_2, \ldots, x_N)$ lies on the $(N-1)$-dimensional simplex

$$S = \{(x_1, \ldots, x_N) \geq 0 : \sum x_i = 1\},$$

in the positive orthant of $\mathbb{R}^N$. Note that since $\sum x_i' = 1$, we can describe the dynamics as iterations of the mapping $T : S \to S$, with the ith coordinate of $T(x)$ to be defined in (4.11). That is,

$$(T(x))_i = \frac{\beta_i x_i}{\sum_j \beta_j x_j}. \tag{4.12}$$

Lemma 4.2. *The ith coordinate of the nth iteration of the mapping T is given by*

$$(T^n(x))_i = \frac{\beta_i^n x_i}{\sum_j \beta_j^n x_j}.$$

Proof. We prove this using induction. When $n = 2$ we have

$$\begin{aligned}
(T^2(x))_i &= (T(T(x)))_i \\
&= \frac{\beta_i (T(x))_i}{\sum_j \beta_j (T(x))_j} \\
&= \frac{\beta_i \left(\beta_i x_i / \sum_s \beta_s x_s\right)}{\sum_j \beta_j \left(\beta_j x_j / \sum_s \beta_s x_s\right)} \\
&= \frac{\beta_i^2 x_i}{\sum_j \beta_j^2 x_j}
\end{aligned}$$

Suppose this holds for $n = k$. Then

$$\begin{aligned}
(T^{k+1}(x))_i &= (T(T^k(x)))_i \\
&= \frac{\beta_i \left(\beta_i^k x_i / \sum_s \beta_s^k x_s\right)}{\sum_j \beta_j \left(\beta_j^k x_j / \sum_s \beta_s^k x_s\right)} \quad \text{by the induction hypothesis} \\
&= \frac{\beta_i^{k+1} x_i}{\sum_j \beta_j^{k+1} x_j}
\end{aligned}$$

Therefore the result holds by the principle of mathematical induction. □

Lemma 4.3. *The ith component of the mapping $T^n(x)$ is increasing in n for $i = 1, \ldots k$.*

Proof. Using the previous lemma we have

$$\begin{aligned}
(T^{n+1}(x))_i &= (T(T^n(x)))_i \\
&= \frac{\beta_i \left(\beta_i^n x_i / \sum_s \beta_s^n x_s\right)}{\sum_j \beta_j \left(\beta_j^n x_j / \sum_s \beta_s^n x_s\right)} \\
&= \frac{\beta_i^n x_i}{\sum_j \beta_j^n x_j \frac{\beta_j}{\beta_i}} \\
&\geq \frac{\beta_i^n x_i}{\sum_j \beta_j^n x_j} \quad \text{since } \frac{\beta_j}{\beta_i} \leq 1, \text{ for all } j \\
&= (T^n(x))_i
\end{aligned}$$

□

Now, let us define $U \subset S$ to be the k-dimensional sub-simplex

$$U = \left\{ (x_1, \ldots, x_k, 0, \ldots, 0) : \sum_{j=1}^{k} x_j = 1 \right\},$$

and $V \subset S$ to be the $(N-k)$-dimensional sub-simplex

$$V = \left\{ (0, \ldots, 0, x_{k+1}, \ldots, x_N) : \sum_{j=k+1}^{N} x_j = 1 \right\}.$$

We claim that any point of U (or V) is a fixed point of the mapping T. To see why, let $u = (u_1, \ldots, u_k, 0, \ldots, 0) \in U$. Then by the definition of U and T, if $i > k$, then $(T(u))_i = 0$. If $i \leq k$ then since $\beta_i = \beta$ for $i \leq k$, and $\sum_1^k u_i = 1$ we have

$$(T(u))_i = \frac{\beta u_i}{\sum_1^k \beta u_j} = u_i.$$

A similar argument shows that T fixes all $v \in V$.

Now, for any $1 \leq i, j \leq$, if $x_j \neq 0$, then we have $T(x_i)/T(x_j) = x_i/x_j$. Thus, T always preserves the relative sizes of the coordinates $x_1, \ldots, x_k$. Therefore if we can show that every forward T-orbit, $\{T^n(x)\}$, converges to U, then the limiting share holdings must be given by (4.9).

Define $P_U : S \to U$ to be the projection fixing the first k coordinates and setting the remaining $N-k$ coordinates to zero. Similarly define $P_V : S \to V$ to be the projection setting the first k coordinates to zero and fixing the remaining $N-k$. Let $S^+ = \{x \in S : P_U(x) \neq 0\}$. This set simply serves to rule out the possibility of all agents with the maximum β having no wealth. Before proceeding we state a useful result from measure theory (see Billingsley (1995), page 80).

Lemma 4.4 (Jensen's Inequality)**.** *Let φ be a convex function and $p_1, \ldots, p_n$ be positive numbers that sum to one. Then for any $x_1, \ldots, x_n$,*

$$\varphi\left(\sum_i p_i x_i\right) \leq \sum_i p_i \varphi(x_i).$$

The inequality is strict if φ is strictly convex and $p_i \in (0,1)$, for all i.

Lemma 4.5. *Define $F : S \to \mathbb{R}$ by $F(x) = \sum_{i=1}^{N} \beta_i x_i$, where the β's are ordered as in the theorem. Then for any $x \in S^+$, $F(T^n(x))$ increases monotonically with limit β, as $n \to \infty$.*

Proof. Since the x_i sum to one, $F(x)$ is nothing more than a weighted average of the β's with β_i receiving a weight of x_i. From the definition of T and F we have, for any x,

$$F(T(x)) = \frac{\sum \beta_i^2 x_i}{F(x)},$$

so that $F(x)F(T(x)) = \sum \beta_i^2 x_i$. Also, $F^2(x) = (\sum \beta_i x_i)^2$. Now, taking $\varphi(x) = x^2$ in Lemma 4.4 we have

$$F(x)F(T(x)) = \sum \beta_i^2 x_i \geq \left(\sum \beta_i x_i\right)^2 = F(x)F(x).$$

Since clearly $F(x) > 0$ we have

$$F(T(x)) \geq F(x), \quad \text{for any } x. \tag{4.13}$$

Note that the inequality is strict if $P_V(x) \neq 0$.

Now, fix $x \in S^+$. If $P_V(x) = 0$ then it must be the case that the first k coordinates of x are 0. Thus, $x \in U$ and $T(x) = x$ since T fixes all points in U. In this case $F(x) = \sum_1^k \beta_i x_i = \beta \sum_1^k x_i = \beta$. Thus, the result clearly holds in this case. Suppose $P_V(x) \neq 0$ and recall that since $x \in S^+$ we also have $P_U(x) \neq 0$ by assumption. Since T maps S into itself it must also be the case that $P_U(T(x))$ and $P_V(T(x))$ are also nonzero. Thus, from (4.13) we have that $F(T^n(x))$ is a strictly monotone sequence bounded by β, and therefore it must converge to its supremum, say β^*.

Suppose that $\beta^* < \beta$. By compactness of S, the sequence $\{T^n(x)\}$ has a convergent subsequence $y_j = T^{n_j}(x) \to x^* \in S$, and by continuity of F we have $F(x^*) = \beta^*$. By Lemma 4.3 we know that $(T^n(x))_i$ is increasing in n so it must be the case that $x^* \in S^+$. Since $F(x^*) < \beta$ and $P_V(x^*) \neq 0$ we have

$$F(T(x^*)) > F(x^*) = \beta^*.$$

However, we also have $F(T(y_j)) \leq \beta^*$, and since $y_j \to x^*$ this contradicts the continuity of F and T. Therefore we must have $\beta^* = \beta$. □

Thus, we have shown that for any $x \in S^+$ the limit of $F(T^n(x))$ is β. Since F is continuous and $F^{-1}(\beta) = U$, every forward T-orbit starting in S^+ must converge to U. This completes the proof of Theorem 4.1.

4.3 Adaptive Learning

In Section 4.1 agents are assumed to be identical and, in addition, they also know they are identical. Given these assumptions the agents are able to make the necessary substitution in their Euler equation and solve for the correct REE pricing function. The key assumption here is that the agents know *a priori* that they will never trade in any period. In Section 4.2 we were able break this convenient symmetry in a special case. We now wish to allow agents to be fully heterogeneous (or at least let them believe they might be). In doing so we are no longer justified to make the substitution $c_t = d_t + e$ (since, in general, it will no longer be true) and thus to solve for the REE pricing function in this case will be difficult or impossible. Note that even if agents are identical but do not know it they still cannot use this substitution since from their point of view they might trade in some period.

In the current setup, our agents are assumed to be *boundedly rational* in the sense that they do not know everything necessary to solve for the REE pricing function. Instead of assuming that they know the true pricing function, we have endowed them with a forecasting function, $\tilde{p}$, that they use to predict future prices. Recall that $\tilde{p}$ is a polynomial in d. For convenience we will denote

$$\tilde{p}(d_t) = \alpha_{0t} + \alpha_{1t} d_t + \cdots + \alpha_{nt} d_t^n = x_t' \alpha_t, \tag{4.14}$$

where $\alpha_t = (\alpha_{0t}, \ldots, \alpha_{nt})'$ and $x_t = (1, d_t, \ldots, d_t^n)'$ are column vectors of length $n+1$.

Using the forecasting rule given by (4.14), the agents will form a demand for the stock at time t which uses their previous parameter estimate α_{t-1}, where α_0 is given. Given all agents' demands, the market clearing price, p_t, will be set such that total demand equals total supply. Most likely the market clearing price will differ from the forecasted prices. Once the new price is revealed, the agents update their α's to reflect the new information they have obtained. Agents will update using *adaptive learning* according to the following scheme:

$$\alpha_t = \alpha_{t-1} + \eta_t R_t^{-1} x_t (p_t - x_t' \alpha_{t-1}) \tag{4.15}$$

$$R_t = R_{t-1} + \eta_t (x_t x_t' - R_{t-1}), \tag{4.16}$$

where R_t is an estimate of the moment matrix of the process x_t and η_t is a "gain sequence". See Marcet and Sargent (1989) and Evans and Honkapohja (2001) for more details on this type of learning.

Typically the scheme in (4.15)-(4.16) would be suitable for an agent that is attempting to learn both the true pricing function as well as the distribution of the dividends. In our case the agents are assumed to know the distribution of dividends, and thus they should be able to compute the true moment matrix R exactly. In fact, one can choose to update the matrix R_t, or keep it fixed throughout, without affecting the limit of the α_t. By fixing R_t to be the identity, for all t, the scheme in (4.15)-(4.16) corresponds to *Stochastic Gradient Learning*, (see Evans and Honkapohja (2001), Section 3.5). We do not explore alternative learning schemes here. Instead, each agent will begin with a given α_0 and with R_0 equal to the identity, and use (4.15)-(4.16) to update.

The gain sequence, η_t, which serves to weight the observations, is non stochastic and must satisfy

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \eta_t^2 < \infty. \tag{4.17}$$

The first condition is required to avoid convergence to a non equilibrium point while the second ensures asymptotic elimination of residual fluctuations in α_t (Evans and Honkapohja, 2001).

A very common and convenient choice for the gain sequence is $\eta_t = 1/t$. This case turns out to correspond to a recursive least squares updating scheme.[2] From a practical standpoint, however, we can do better. Since the agents learn the true values of the parameters they estimate, it makes sense that the prices observed in the market get closer to the REE prices. Thus, we would like to weight the most recent observations as highly as possible since they are the most relevant data points. In other words we would like the sequence η_t to decrease as slowly as possible. Clearly if we take $\eta_t = 1/t^{0.5+\epsilon}$ for any $\epsilon > 0$, then the two conditions stated in (4.17) are satisfied. At the same time, however, the sequence is decreasing at a much slower rate than $1/t$ would. By choosing $\eta_t = 1/t$ we are assigning all observations an equal weight. What this means is that for a large number of observations, each new one has little effect on the overall estimate. Choosing a sequence that decreases slower will assign more weight to recent observations. For the experiments run throughout this chapter and the next we will use $\eta_t = 1/t^{0.6}$.

There is an extensive literature devoted to using known results from stochastic approx-

[2] In addition to choosing $\eta_t = 1/t$ we also need to choose the initial conditions properly in order to reproduce least squares estimates. For a discussion of this see Evans and Honkapohja (2001), page 33.

imation theory (see, for example, Evans and Honkapohja (2001) or Sargent (1993)). The results presented in this literature are very elegant but unfortunately they do not extend well to our case. The models to which these techniques are generally applied often have a simple, explicit equation that determines the market clearing price as a function of the expectation of that price:

$$p_t = F(p^e_{t+1}), \tag{4.18}$$

where p^e_{t+1} is the expected value of the future price. Agents have a *perceived law of motion*, which they use to form their expectation, p^e_{t+1}, that we can easily substitute into the (4.18) to obtain the *actual law of motion*. Once these steps are completed it is relatively straightforward to apply existing stability results to establish conditions under which the system will converge to the REE. The problem that we face is somewhat more complicated. While it is certainly true that the agents expectations (among other parameters) will play a role in determining the market clearing price, we are unable to explicitly write out this dependence. There does exist some underlying function that determines the market clearing price as a function of all market parameters, but to establish stability results in this case we would need to resort to numerical differentiation. Sargent summarizes this difficulty nicely in the following statement:

> "Verifying the convergence of the system is technically difficult because the firms are learning about a 'moving target', a law of motion that is influenced by their own learning behavior." (Sargent (1993), page 123)

4.4 Market Clearing and Updating

Now that we have established how agents will learn, we need to specify exactly how the market works. We begin with a market level algorithm (Algorithm 4.1), and then we will explain each step.

Algorithm 4.1 Market Algorithm

Time 0: Agents are born with s_0^i shares of stock and a pricing function $\tilde{p}^i$. They can immediately compute their initial policy function g^i.[3]

For each time t, the following steps are carried out.

1. The dividend is declared by the market and the agents receive their endowments.
2. Given s_t^i, d_t and e^i, the agents form a demand as a function of p_t.
3. All agent demands are passed to the market maker. Price is set to equate total demand with total supply.
4. Once the price is announced, agents update their share holdings. That is, given p_t, they can compute $s_{t+1} = g^i(s_t^i, d_t, p_t)$.
5. Once holdings are known agents consume the remainder of their wealth.
6. Agents update their pricing functions.
7. Agents can update their policy functions.

4.4.1 Step 1: Declaring a Dividend

It is very straightforward to declare a dividend. Recall from Chapter 3 that we will assume the dividends are draw from a discrete distribution since if they are not then we will discretize the continuous distribution. We have assumed throughout that the dividend process is *iid*, but it need not assign equal probability to each value. Let $d = \{d_1, \ldots, d_n\}$ and $\pi = \{\pi_1, \ldots, \pi_n\}$ denote the dividends and their probabilities, respectively. Algorithm 4.2 describes how to find d_t.

4.4.2 Step 2: Forming a Demand

This step is also very straightforward. The agents all have a complete policy function, g^i. They simply fix s and d and form a new function $h^i : \mathbb{R}^+ \to \mathbb{R}^+$ defined by $h^i(p_t) = g^i(s_t^i, d_t, p_t)$.

[3]Note that although we do not write it explicitly, the policy function g^i is conditional on a pricing function, $\tilde{p}^i$. This g^i is correct given the pricing function being used, but since the pricing function is being learned, so too, in a sense, is the policy function.

Algorithm 4.2 Draw a Dividend

1. Set $j = 0$, $p = 0$ and $q = \pi_1$.
2. Draw a uniform random variable, $u \in [0, 1)$.
3. If $u \in [p, q)$ set $d_t = d_{j+1}$ and stop.
4. Otherwise set $j = j + 1$, $q = q + \pi_j$, $p = q$ and return to step 3.

4.4.3 Step 3: Market Clearing

There are several ways to compute the market clearing price. The most natural way is to have the market maker call out prices and have the agents respond with demands for each price. If the demand were too large then a higher price would be called and similarly if demand were too low then a lower price would be called. This auction would continue until the auctioneer called out a price that resulted in zero excess demand. While this is the most intuitive way to clear the market, it would add an additional loop and would prove to be more costly computationally. Instead, we can clear the market if we have the agents all pass their functions, $h^i(p_t)$, to the market maker. Since the market maker knows the aggregate supply, he can simply sum the individual demands, set them equal to supply and solve. That is, the market maker will find p_m such that

$$H(p_m) = \sum_{i=1}^{N} h^i(p_m) = N. \tag{4.19}$$

In the case when $\gamma \leq 1$, we know from Chapter 3 that the policy functions are strictly decreasing in p. Thus, if all agents have $\gamma \leq 1$, then the sum of the policies is also strictly decreasing. This means there is a unique root and we can use a binary technique to solve for it. Algorithm 4.3 outlines how to do this. The situation is slightly more complicated if at least one agent has $\gamma > 1$. In this case the policy is not strictly decreasing in p. Thus, there could exist more than one p that would solve (4.19). In practice, however, this will rarely happen. To be sure, when we are dealing with the case when $\gamma > 1$ for at least one agent we will check on the fly to make sure that there is only one root to (4.19). If there appears to be more than one root a warning will be displayed to let us know.

It turns out that Algorithm 4.3 does not require the function be strictly decreasing everywhere. All that is required is a unique root and the function be decreasing at the root.

This guarantees that, for any p, if the function evaluates to a positive (negative) then the root must be to the right (left) of p.

Algorithm 4.3 Solving for the Market Clearing Price

Input: A function $f(x)$ with a single root, a starting value x_0 and a tolerance $\varepsilon > 0$.

1. Set $x_l = x_0/2$ and $x_u = 2x_0$.
2. If $f(x_l) < 0$, set $x_l = x_l/2$ and return to step 1. If $f(x_u) > 0$, set $x_u = 2x_u$ and return to step 1.
3. Set $\overline{x} = (x_u + x_l)/2$. If $f(\overline{x}) > 0$ set $x_l = \overline{x}$. Otherwise set $x_u = \overline{x}$.
4. If $x_u - x_l < \varepsilon$, stop and return $(x_u + x_l)/2$. Otherwise return to step 3.

We will pause to clarify how important it is to have a positive endowment. We saw in Chapter 3 that if we chose not to use splines then the resulting demand function would have flat spots in it. While using a spline does technically eliminate this problem, numerically it will still persist. The problem is that even though the function is strictly decreasing, it could decrease very slowly. This means that there could be many prices which will lead to the markets "nearly" clearing. Since (4.19) will be solved to within a given tolerance, having a very flat function can lead to poorly conditioned root finding. Adding a positive endowment will make sure that the slope of $H(p_m)$ is sufficiently steep to ensure that the root finding will converge to unique root.

There is one more detail that needs to be addressed within our discussion of market clearing. Throughout this discussion we have spoken about $h^i(p)$ as if it were a continuous function. In fact, it is not. We have only applied a spline in the s direction, but in the p direction we still have discrete function values. To effectively clear the market, we will need to be able to evaluate H (and therefore each h^i) at any p. We do so by using piecewise linear interpolants. We know from the plots in Chapter 3 that even with a positive endowment we will still see a large amount of curvature in the policy function as a function of p. This could indicate that using linear interpolation is insufficient since linear functions will not preserve this shape. The policy function, however, is generally well behaved in p and thus the function values will not change too much from one point to the next. Thus, as long as we use a sufficiently small step size in p, linear interpolants will perform well enough. It would

be nice to use a higher order interpolation, but we suspect doing so would add significant computation time and thus it was not attempted.

The above discussion raises one final detail. All of the approximations (tolerance on the root, linear interpolations, etc.) have an error associated with them. These errors will occur every time we perform these operations and thus it is expected that they will compound. The result is limited accuracy in the market clearing mechanism. This is the same problem we faced in the computation of the policy functions. It is simply not reasonable to expect that we are able to achieve arbitrary accuracy from these computations. As a result, when we run our test cases (i.e. cases in which we know the true REE pricing function) we will have to be willing to accept that the computed pricing functions will only be correct to certain point. Results and discussion will be provided in Section 4.5.

4.4.4 Step 4: Consumption

The agent's wealth in period t is given by $w_t^i = s_t^i(p_t + d_t) + e^i$. This is a strange quantity in that we usually think of wealth as being known in advance, and then making our decision based upon it. In this model, however, the agent actually does not know his wealth until the market has cleared. Once he knows is wealth, he immediately knows what he will consume. Recall that in Proposition 2.1 in Chapter 2 we showed that the agent's budget constraint will bind in all periods. This implies that $c_t^i = w_t^i - p_t s_{t+1}^i$, for all t.

4.4.5 Step 5: Update Pricing Function

Once the market has cleared in time t, the agent now has a new data point (d_t, p_t). Using this and (4.15)-(4.16), they can update their α.

We note here in passing that if we had chosen a learning rule for which the agent required the complete history of prices and dividends[4], there is one additional trick we could implement to help the agents learn the REE quicker. The earliest data points are weighted the highest in the approximation. Thus if the agent starts with a very poor α then it will take a very long time to converge since he will have to observe a large number of points later to overcome the poor start. To help overcome this problem, we can have the agent observe and update for a while and then choose to ignore all points prior to a certain time. The

[4]In early versions of our model we assumed that agents, in each period, performed a full regression on all available history.

rationale for doing this would be that the agent recognizes that the early observations are noisy and thus he is smart enough to ignore some early points after a while. How long he should wait before ignoring them would depend upon how bad the initial guess was, but the reality is if he resets his history at any point it will certainly help. Since our agents only use the most recent data to update, we have no need for this trick here.

4.4.6 Step 6: Update Policy Function

Once the agent has a new pricing function he can re-compute his policy function. This was the problem studied in Chapter 3. Fortunately, we are able to make this step somewhat faster after the first time step. At time zero the agent computes his policy for the very first time. This means he starts with a constant function, performs a discrete grid search and finally a continuous grid search to obtain his initial policy function and value function. Now, even though we are not ultimately interested in the value function, we do not throw it away. Instead, we store it and use it as the starting point for next period. This will save us alot of work for two reasons. First, since we will already have an approximation we will not have to perform the discrete grid search after time zero. Second, the closer we start to the true solution the fewer iterations we will require to converge. Clearly the last approximation will be closer to the true solution than starting all over again with a constant function and iterating all the way. Typically, the initial runs will require 50-100 iterations but eventually it will only take a few to converge. Thus, the initial time steps are very computationally intensive, but things get better as time goes on. We will discuss the performance of the algorithm further in Section 4.5.

4.5 Numerical Results

to verify that the code is working correctly, several experiments will be run for which we know the true REE pricing function. We will consider three versions of the model in this section. In all cases the agents are identical in all ways except possibly in their price forecasting rule. The following three cases will be considered:

1. All agents begin with the REE pricing function. In this case all agents are identical and they will not trade. The market will clear at the price predicted by the agents' forecasting rule.

2. All agents begin with the same non-REE pricing function. Again, agents will not trade but prices will not initially clear at the REE level. Agents will observe the prices and update their forecasting rules, and eventually converge to the REE.

3. Agents begin with different pricing functions. In this case trading will occur. Thus, agents will no longer be identical since they will have different holdings. It will still be the case, however, that agents will learn the REE pricing function corresponding to the homogeneous agent economy.

4.5.1 Default Model Parameters

Unless otherwise specified, the following parameters will be used throughout this chapter. There will always be two agents. Agents have the following parameters in common: $\beta = 0.9$, $s_0 = 1.0$, $e = 10$ and γ (varied). The parameter γ will be varied across experiment, but for now all agents will have the same γ within an experiment. The s grid will be taken to have 101 points over $[0, 2]$ and we will use splines. The p grid will be taken to be large enough to contain the price forecasts of all agents and will contain 201 points. Several dividend grids will be used, depending upon what the given setup necessitates. For a two state process we will use $\{0.75, 1.25\}$, for a three state process $\{0.75, 1.0, 1.25\}$ and for a four state process we will use $\{0.75, 0.9, 1.1, 1.25\}$. In all cases dividends are each equally likely. Table 4.1 shows the true REE pricing coefficients for some cases that will be used later in the section. When $\gamma \leq 1$, the true values of the coefficients are computed using (4.3) and for $\gamma = 0.5$ we use a Taylor approximation using (4.20)-(4.21), given later in this chapter.

Table 4.1: True REE pricing coefficients for some cases.

γ	e	$\#D$	α_0	α_1	α_2	α_3
1.0	0	2	9	0	0	0
1.0	10	2	8.1395	0.81395	0	0
1.0	10	3	8.1528	0.81528	0	0
2.0	10	3	7.3891	1.4779	0.0739	0
0.5	10	4	8.5680	0.4284	−0.0107	5.3512(−4)

The remainder of this section contains plots depicting the evolution of the agents' pricing coefficients as they update and learn over time. The starting values and the values the agents converged to are all given in Table 4.2 at the end of the section.

4.5.2 The Homogeneous REE Case

Fix $\gamma = 1.0$ and suppose a two state dividend process. The REE pricing function in this case will be $p(d) = 8.1395 + 0.81395d$. Agents are given this function initially. The results are what we would expect: The agents do not trade and their pricing functions do not change. This shows that the code can reproduce the classical homogeneous, perfect information case.

4.5.3 The Homogeneous but non-REE Case

Again, we fix $\gamma = 1.0$. Since we have not assumed the agents have the REE pricing function, trading could be possible. The agents do not trade, however, since they are identical, but there will be learning. The agents still update their pricing function at each time step. Since they are using the same learning rule, their pricing functions are identical in all periods. The result is convergence to the REE as shown in Figure 4.1.

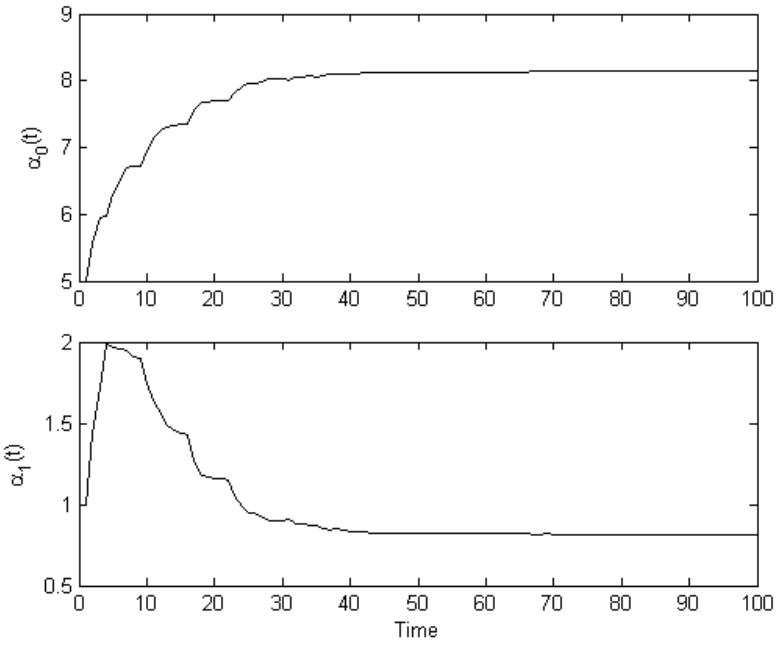


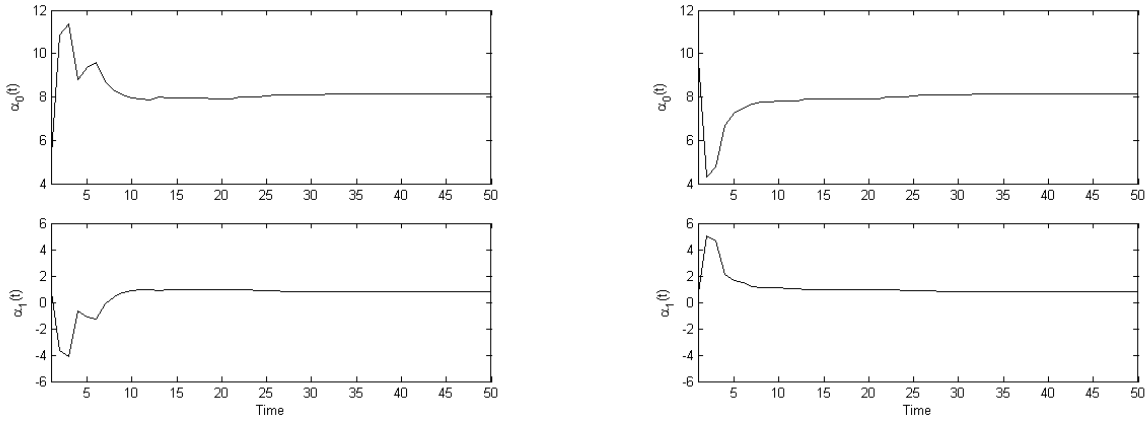
Figure 4.1: The identical, but non-REE case (Case 2). Agents each have $\gamma = 1$ and each started with $\alpha = (5, 1)$. They converged to the REE pricing function, $p(d) = 8.140 + 0.8140d$.

This is an interesting special case since it corresponds to what is typically done in the learning literature (see, for example, Honkapohja and Mitra (2002)). It is very common for models such as this to make the assumption that $c = d$ (or in our case $c = d + e$), substitute this into the Euler equation and use the resulting functional equation as a starting point.

Following this, we would suppose that the agents have the wrong price forecasting rule and show that they converge to the REE. The problem with this approach is the assumption of $c = d$ also, as discussed previously, makes the assumption that agents are identical and they know this fact. In this case it does not make sense for them to not have the REE pricing function. We have to assume that our agents are intelligent. The reason they do not, in general, know the REE pricing function is due to a lack of information, not a lack of ability to compute it. Thus, if agents were able to make the assumption that $c = d$ then they ought to be able to compute the REE pricing function also. The case of identical, but wrong agents is, nonetheless, the most common within the learning literature.

4.5.4 The Heterogeneous Case: Log Utility

We first consider the smallest step away from the homogeneous case. We will assume that there are two agents, both with log utility, but different price forecasting rules (although all other parameters remain identical). The result is what we expected: Agents trade initially because their forecasts about future prices differ and then each agent will converge to the REE pricing function. Figure 4.2 shows the convergence of the coefficients for each agent.

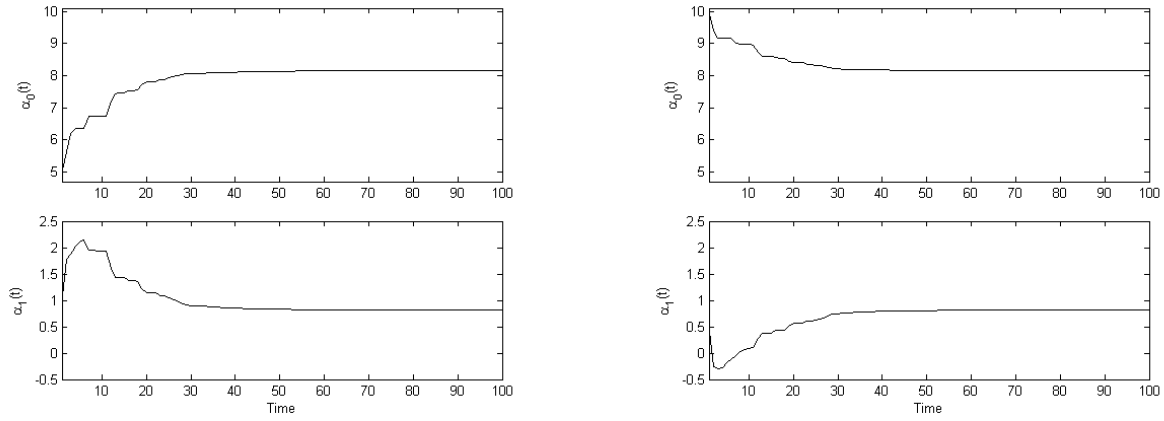


(a) Agent 1 (b) Agent 2

Figure 4.2: Log utility, non identical agents and a two state dividend process (Case 3.1). Agents begin with $\alpha^1 = (5, 1)$ and $\alpha^2 = (10, 0.5)$, and converge to the REE pricing function, $p(d) = 8.140 + 0.8140d$.

In the preceding example we had the agents learn two parameters and there were exactly two possible states of the world (i.e. two dividends). This may seem a little too convenient.

The next case shows that the number of states is not important. The parameters are identical, except that we will assume now that $d \in \{0.75, 1.0, 1.25\}$. Note that the expected value of d is still one as it was before, but the REE pricing function will change slightly. This can be seen from (4.3). If $e = 0$ then the distribution of d is not important but for $e > 0$ it is. The agents are found to converge to the REE in this case also. Figure 4.3 shows the results.



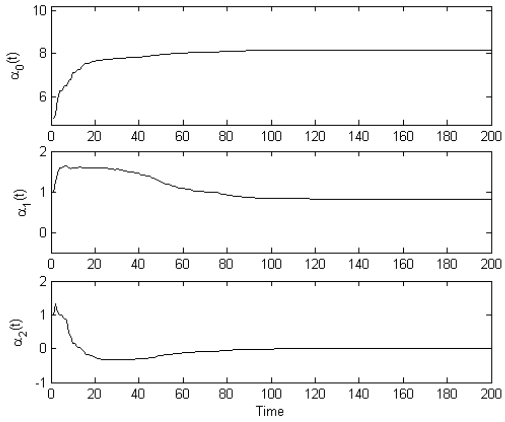
(a) Agent 1 (b) Agent 2

Figure 4.3: Log utility, non identical agents and a three state dividend process (Case 3.2). Agents begin with $\alpha^1 = (5, 1)$ and $\alpha^2 = (10, 0.5)$, and converge to the REE pricing function $p(d) = 8.154 + 0.8154d$.

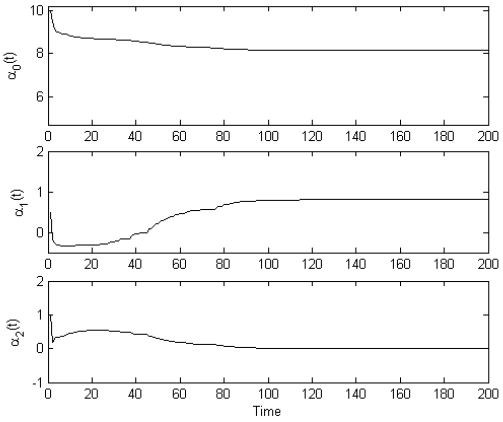
In the previous two cases the agents began with a linear forecasting rule and the true REE function was also linear. This, again, seems like a rather convenient choice. In our next example we will use the same setup as above, except we will have the agents begin with a quadratic forecasting rule. The result is what we would expect - the agents learn the coefficient on the d^2 term is zero. Note that in this case we must use at least a three state dividend process since the agents are attempting to learn three coefficients. If we were to use the two state process in this case the agents would have a free parameter and thus may not find the quadratic coefficient to be zero. Results are shown in Figure 4.4.

4.5.5 The Heterogeneous Case: Non-log Utility

We now take a step away from the often convenient $\gamma = 1$ assumption. Note that (4.3) holds for *any* $\gamma > 0$ and $e \geq 0$. Recall that in this chapter we are only allowing agents to differ in their pricing rule and not in any other way.



(a) Agent 1



(b) Agent 2

Figure 4.4: Log utility, non identical agents and a three state dividend process (Case 3.3). Agents begin with quadratic pricing functions, $\alpha^1 = (5, 1, 1)$ and $\alpha^2 = (10, 0.5, 1)$, and converge to the REE pricing function $p(d) = 8.154 + 0.8154d + 0d^2$.

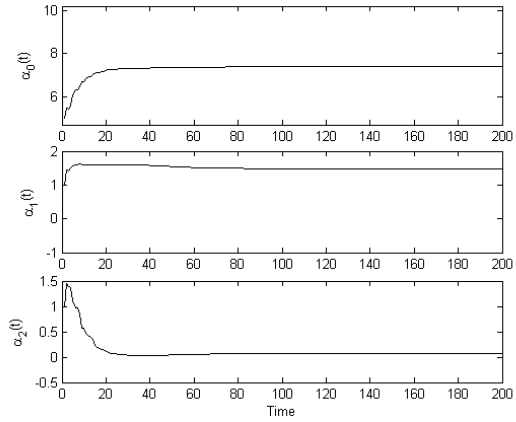
In the first example we will assume $\gamma = 2$ for both agents. Note that we could use any $\gamma > 1$ and obtain similar results, but Agent 2 has the advantage of being easy to expand (4.3) as a polynomial. The setup remains the same as in the log cases, with dividends following a three state process. Agents begin with a quadratic pricing rule and the true solution is also quadratic. We could perform a similar experiment as above and give the agents a higher order polynomial, but they will learn the coefficients of the higher order terms are zero. Results are shown in Figure 4.5.

Finally we wish to investigate the case when $\gamma < 1$. In this case the true REE function is slightly more difficult to deal with since we have a fractional exponent and yet the agents are attempting to find a polynomial that fits the data. Using a Taylor series expansion we can write the true function as an infinite series. Clearly the agent will not be able to learn an infinite number of coefficients. As a result we will have them learn only a truncated version of the true solution. We consider the case when $\gamma = 0.5$. It is straightforward to compute

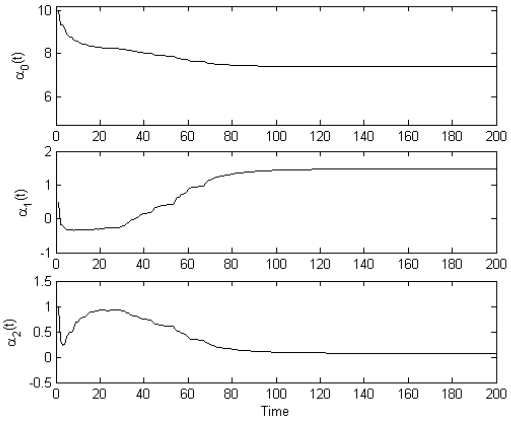
$$(d + 10)^{1/2} = \sum_{n=0}^{\infty} c_n d^n, \tag{4.20}$$

where $c_0 = 10^{1/2}$, $c_1 = -\frac{1}{2}10^{-1/2}$ and

$$c_n = (-1)^{n-1} \frac{[(2n-3)(2n-5)\dots 3 \cdot 1]10^{(2n-1)/2}}{2^n n!}, \quad \text{for } n \geq 2. \tag{4.21}$$



(a) Agent 1



(b) Agent 2

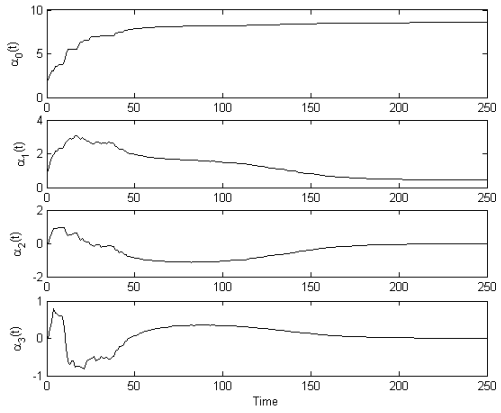
Figure 4.5: $\gamma = 2$, non identical agents and a three state dividend process (Case 3.4). Agents begin with $\alpha^1 = (5, 1, 1)$ and $\alpha^2 = (10, 0.5, 1)$, and converge to the REE pricing function $p(d) = 7.3892 + 1.4782d + 0.0737d^2$.

Note that the radius of convergence of this series is ten so it is valid for all reasonable values of d. Assuming the three state dividend process, we can compute the first few terms of $p(d) = \dfrac{\beta}{1-\beta}\Gamma(e, \gamma)(d + 10)^{1/2}$ giving

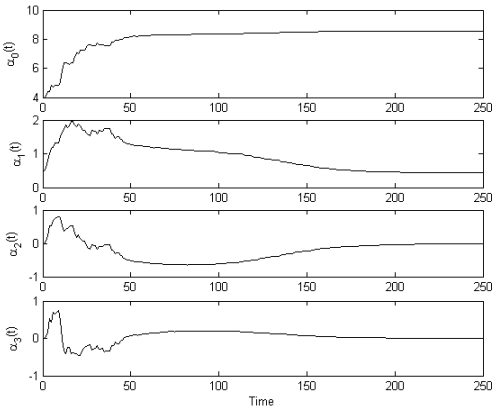
$$p(d) = 8.567984 + 0.428399d - 0.01071d^2 + 0.000535d^3 + \ldots$$

Since these terms are going to zero rather quickly it would be reasonable that we expect the agent to learn only the first few of them. Once they do so, we could assume that the agent is learning the true REE pricing function. The results of this experiment are shown in Figure 4.6.

Theoretically, in the previous experiment we could have the agents learn the true solution to any order we want. Practically, however, this is not possible. Due to the rounding errors and tolerances used throughout the algorithm we are unable to obtain arbitrary accuracy in our computations. Specifically, the market clearing price is generally only accurate to within two or three decimal places. The agents will have a difficult time correctly learning the higher order terms that are close to zero. It is likely that those terms would not converge since they would mostly be the product of compounded approximation errors.



(a) Agent 1



(b) Agent 2

Figure 4.6: $\gamma = 0.5$, non identical agents and a three state dividend process (Case 3.5). Agents begin with $\alpha^1 = (2, 1, 0, 0)$ and $\alpha^2 = (4, 0.5, 0, 0)$, and converge to the approximate REE pricing function $p(d) = 8.568 + 0.430d - 0.012d^2 + 0.000977d^3$.

4.6 Summary

It should be clear from the starting and ending values of the coefficients in Table 4.2, compared them with the true values in Table 4.1, that the algorithm is working correctly and the agents are learning the true values of the parameters. As we can see in the same tables, however, it is often the case that agents are unable to learn the true pricing coefficients beyond a couple of decimal places. As we have mentioned several times in this chapter and previously, refining grids and lowering tolerances will have some positive effect, but the benefits do not outweigh the costs. This discussion raises the issue of what exactly will be meant by *convergence* of the pricing parameters, an issue that has been avoided thus far but will be addressed shortly. In the following chapter we will perform similar experiments except for cases in which we do not have a known analytical solution.

Table 4.2: Results obtained from running various cases.

			Initial α				Final α			
Case	γ	$\#D$	α_0	α_1	α_2	α_3	α_0	α_1	α_2	α_3
1	1	2	8.1395	0.81395	0	0	8.1395	0.81395	0	0
			8.1395	0.81395	0	0	8.1395	0.81395	0	0
2	1	2	5.0	1.0	0	0	8.1395	0.81398	0	0
			5.0	1.0	0	0	8.1395	0.81398	0	0
3.1	1	2	5.0	1.0	0	0	8.1395	0.8140	0	0
			10.0	0.5	0	0	8.1395	0.8140	0	0
3.2	1	3	5.0	1.0	0	0	8.1536	0.8154	0	0
			10.0	0.5	0	0	8.1536	0.8154	0	0
3.3	1	3	5.0	1.0	1.0	0	8.1534	0.8160	$-2.86(-4)$	0
			10.0	0.5	1.0	0	8.1534	0.8160	$-2.80(-4)$	0
3.4	2	3	5.0	1.0	1.0	0	7.3892	1.4782	0.0737	0
			10.0	0.5	1.0	0	7.3893	1.4781	0.0737	0
3.5	0.5	4	2.0	1.0	0	0	8.5676	0.4297	-0.0121	$9.77(-4)$
			4.0	0.5	0	0	8.5688	0.4262	-0.0085	$2.34(-4)$

CHAPTER 5

COMPUTATIONAL RESULTS

Now that we have established that our code works correctly, we turn our attention to computing the REE pricing function when agents are fully heterogeneous. We allow agents to differ in their discount factors, levels of risk aversion, endowments, initial holdings and initial price forecasting rules.

5.1 Choosing a Degree of Approximation

Recall from Chapter 4 that the REE pricing function is computed so that it is not important whether dividends are discrete or continuous. When they are discrete, the agent really does not need to know the entire function, but rather, just what the function evaluates to at each dividend. For example, in the simplest case when $d \in \{0.75, 1.25\}$, with $\gamma = 1$ and $e = 0$, Table 4.1 shows the REE pricing function is $p(d) = 9d$. This function is correct for *any* d. However, since there are only two possible dividends there are also only two possible equilibrium prices, namely, 6.75 and 11.25. As long as the agent knows the two possible prices, it is irrelevant whether or not he knows the entire function, since it is never needed. This turns out to be crucial for cases in which we do not know the true REE pricing function.

In Chapter 4 we often knew what the true REE pricing function looked like. If it was a polynomial then we knew precisely what the order was and thus, we knew how many parameters the agent needed to find. This approach to the problem fails when we have heterogeneous agents since we are unable to compute the true function analytically. However, as discussed above, the agent does not need the entire function, but rather, he needs only a price for each dividend.

The previous discussion seems rather obvious in hindsight, but it is an extremely important observation to make. In order to approximate the true REE pricing function in

the heterogeneous case (as we did in Chapter 4), the following reasoning would be required. First, observe that the function in this case may or not be a polynomial. We saw in Chapter 4, however, that when the true function is not a polynomial the agent may still act as if it is, and simply approximate the first few terms of its Taylor expansion. This is a valid approach except that we really have no way of knowing, *a priori*, what order of polynomial to use. In Chapter 4 we were able to compute the coefficients exactly and thus we knew that using only the first few would lead to a very good approximation. The only way to proceed in general is to try higher orders until we are satisfied that we are approximating the true function sufficiently. This will prove to be very time consuming and should be avoided.

In light of the discussion at the start of this section, we can now think of the problem in a slightly different way. Instead of thinking of the agent as learning an entire (infinite dimensional) function, we can think of his problem as only learning a finite number of prices. Fortunately, this can easily be implemented within our current framework.

Since the agent is assumed to know the distribution of the dividends, he knows exactly how many prices he needs to learn. Suppose there are n possible dividends and therefore n prices to learn. It is well known that there exists a unique polynomial of degree $n-1$ that interpolates n distinct points. Therefore, having the agent learn n prices is equivalent to having him learn a polynomial of degree $n-1$ (which has n coefficients counting the constant). This implies that we can use the same learning scheme as we did in Chapter 4, with the degree of the approximation equal to one less than the number of dividends. In doing so we eliminate the need to guess the degree of the true function.

5.2 The Homogeneous Agent Case

In Chapter 4, we derived the REE pricing function in the case of homogeneous agents. In this chapter we will consider, in general, the case when agents are heterogeneous. Occasionally we will observe the market to converge to a homogeneous market (the case considered in Theorem 4.1, for example).

Throughout this chapter we will assume that dividends follow a two state process so that agents will learn the REE prices using a linear pricing function. In the case that agents remain heterogeneous forever, we have no way to know a priori what the REE prices will be. However, we can compute them when the market eventually converges to a homogeneous economy.

Suppose agents are identical with general γ and e. Suppose there are two dividends, d_1 and d_2, with $d_1 < d_2$. From (4.3) we have

$$p(d) = \frac{\beta}{1-\beta}\Gamma(e,\gamma)(d+e)^\gamma, \tag{5.1}$$

for any d. Now, suppose the agent's forecasting rule is given by $\tilde{p}(d) = \alpha_0 + \alpha_1 d$. Then to find the values for α_0 and α_1 that lead to the REE prices we simply solve the following linear system:

$$\begin{aligned} p(d_1) &= \alpha_0 + \alpha_1 d_1 \\ p(d_2) &= \alpha_0 + \alpha_1 d_2. \end{aligned}$$

Doing so, and using (5.1), gives

$$\alpha_1 = \frac{\beta}{1-\beta}\frac{\Gamma(e,\gamma)}{d_2 - d_1}\left[(d_2+e)^\gamma - (d_1+e)^\gamma\right] \tag{5.2}$$

$$\alpha_0 = \frac{\beta}{1-\beta}\Gamma(e,\gamma)(d_1+e)^\gamma - \alpha_1 d_1. \tag{5.3}$$

These two formulas will be useful throughout this chapter.

5.3 Convergence in the General Case

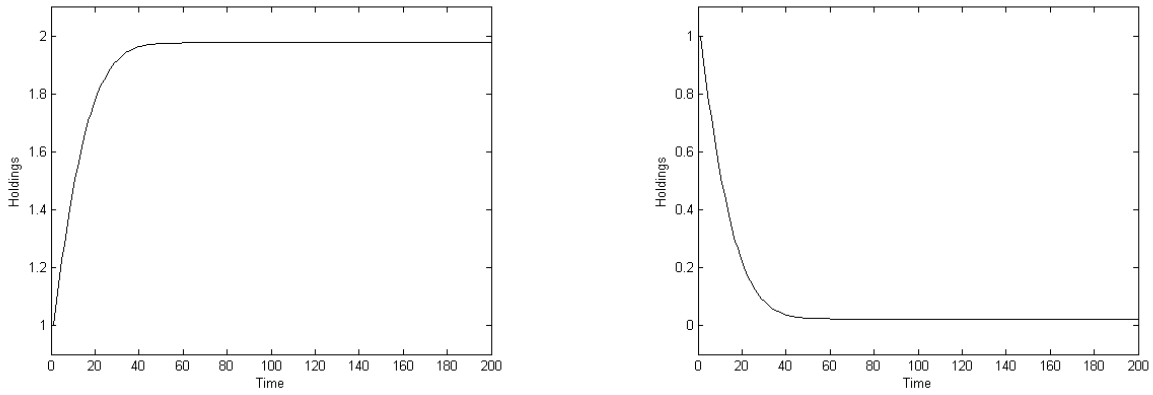
Previously, in Chapter 4, since we only allowed agents to differ in their price forecasting rules, we had the ability to compute the true REE pricing function. It was easy to check if our agents had learned the correct function, since we knew precisely what the correct function was. In the current chapter, however, we wish to study heterogeneity in a more general context. In certain cases, one will be able to compute the equilibrium pricing function, but in general it will no longer be possible. Therefore, we now require a more general definition of what we mean by the convergence of the pricing functions to the equilibrium. Specifically, we will say that the agents have converged to the equilibrium prices if each agent is able to correctly determine the market clearing price in all periods, and for any dividend. That is, for some $T > 0$, $\tilde{p}^i(d_t) = p_t$, for all i and for all $t > T$.

5.4 Varying the Discount Factor

In Chapter 4 we stated and proved a theorem (Theorem 4.1) on the convergence of heterogeneous agents with log utility and no endowment. Specifically, we showed that the

agents with the largest β would survive and hold all of the wealth while the remaining agents would asymptotically vanish. Furthermore, we showed that the market clearing prices converge to the REE prices corresponding to a homogeneous agent economy.

We now verify that we can reproduce the results of Theorem 4.1. Assume there are two log-utility agents, one with $\beta = 0.9$ and the other with $\beta = 0.8$. Each agent begins with one share of stock and no endowment. Figure 5.1 shows that the agent with the larger β will hold all of the shares, while the other agent converges to zero (or close to it). Note that the theorem shows that the holdings of the agents with the smaller discount factors will converge to *zero* asymptotically. If an agent with no endowment ever chooses to hold zero shares he would die in the very next period. Death is assumed to be the worst possible outcome and thus, agents would never choose to hold zero shares. Therefore, for all finite times, all agents will hold a positive amount of shares.

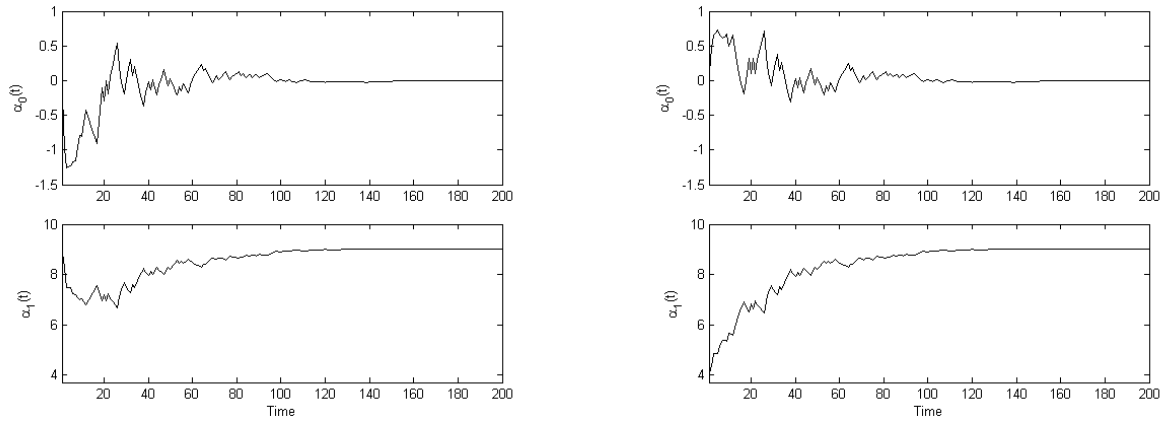


(a) Agent 1: $\beta = 0.9$ (b) Agent 2: $\beta = 0.8$

Figure 5.1: The holdings of agents who are identical except for β. The agents depicted here satisfy the hypotheses of Theorem 4.1 ($\gamma = 1$, $e = 0$) and each initially held one share.

Figure 5.2 shows the convergence of the agent's pricing rules to the REE pricing function corresponding to the case when both agents have $\beta = 0.9$, which is $p(d) = 9d$ in this case. The holdings converge quite quickly (approximately 70 periods), while the the pricing functions take well over 100 periods to converge.

The next question we investigate is whether or not a similar result can be stated for agents with γ different than one (but constant) and/or positive (constant) endowment. We set up the model as follows: There are two agents and two possible dividends. As we have



(a) Agent 1: $\beta = 0.9$ (b) Agent 2: $\beta = 0.8$

Figure 5.2: The pricing coefficients of agents who are identical except for β. The agents depicted here satisfy the hypotheses of Theorem 4.1 ($\gamma = 1$, $e = 0$). Agent 1 began with $\alpha = (0, 9)$, Agent 2 began with $\alpha = (0, 4)$, and both converged to $p(d) = 9d$.

discussed previously, there is little to gain by using more than the minimum number of dividends. For now, agents are assumed to have the same γ and e although we place no restriction on what they may be. Agents will have different discount factors and different initial pricing functions. Specifically one agent will have $\beta = 0.9$, and the other will have $\beta = 0.8$. Although the Theorem allows us to use any initial holdings we wish, we will continue to assume that each agent will begin with a single share. Different initial holdings will be discussed in a later section.

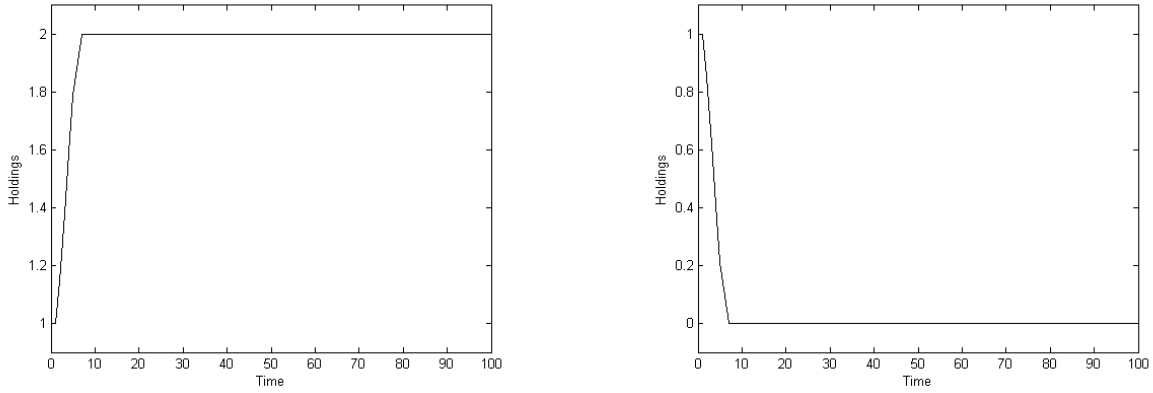
The result is similar to the case when $\gamma = 1$ and $e = 0$ for all agents. That is, the agents will trade to a point where the agent with the larger β holds all of the shares of the stock and the other agent has none. The situation now is slightly different, however, since the agent with the lower β has an endowment. Therefore, even after he sells all of his shares, he is still able to remain in the market, and will continue to hold no shares, forever.

The Case of $\gamma = 1$ and $e > 0$

In the case of log utility and positive endowment, the result is nearly identical to the result when there is no endowment, with the exception of faster convergence of the holdings. In the no endowment case, agents define their entire future wealth by how much they decide to invest in each period. They will be reluctant to sell their shares since each time they do so, it reduces their future wealth. In the end, however, if one agent has a larger β, then his

desire to save will outweigh the other agent's desire to not go broke, and trading will occur. This is repeated until one agent eventually holds an arbitrarily small amount of the stock.

With positive endowment, the agents are much more willing to trade since it will not result in a significant loss of future wealth. Typically in these simulations we use an endowment of five or ten, while the dividends paid by the stock have an expected value of one. Thus, with an endowment, only a small percentage of the agent's total wealth is determined by the dividend paid. (This is true in real markets also.) Hence, the agent with the lower β is much more willing to sell his shares of stock in favor of consumption today. The wealth distribution converges to the corner solution faster than in the no endowment case. Figure 5.3 shows this faster convergence, while Figure 5.4 shows the convergence of the pricing functions.

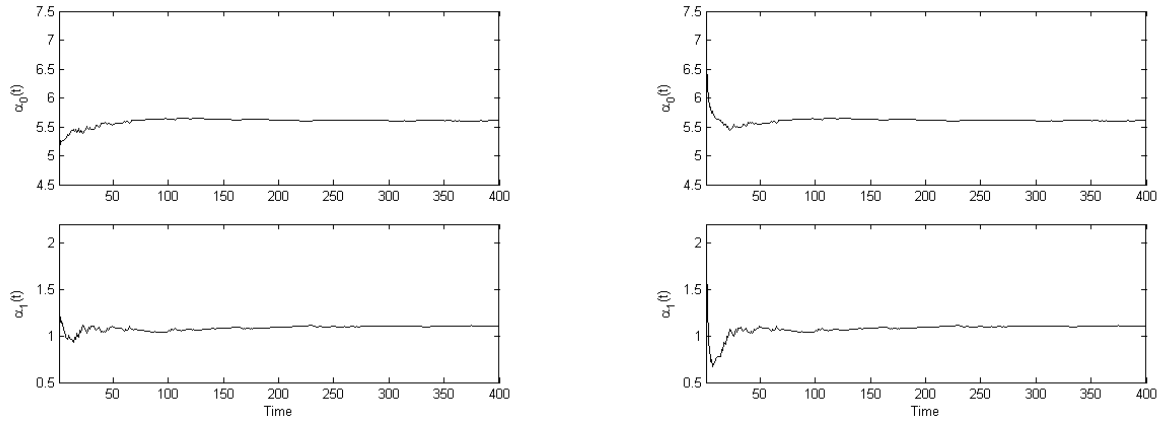


(a) Agent 1: $\beta = 0.9$ (b) Agent 2: $\beta = 0.8$

Figure 5.3: The holdings of agents who are identical except for β. Both agents have $\gamma = 1$ and $e = 10$, and each initially held one share.

It is important to note here that agents, if given the opportunity, would like to continue trading. We have not allowed any short sales in this market and there is a finite supply of the asset. Therefore, the agents have stopped trading, not because it is optimal to do so, but because the market does not permit them to trade further.[1] In this case the agent's first order conditions (i.e. his Euler equation) will fail to hold.

[1]Note that even if we allow short sales of the stock, we would have to impose a restriction on how many shares an agent could sell. In this case the result would still be a corner solution, but just at a different corner.



(a) Agent 1: $\beta = 0.9$

(b) Agent 2: $\beta = 0.8$

Figure 5.4: The pricing coefficients of agents who are identical except for β. These agents both have $\gamma = 1$ and $e = 10$. Agents begin with $\alpha^1 = (5, 1)$ and $\alpha^2 = (7, 2)$, and converged to $p(d) = 5.605 + 1.105$.

Another difference between this case and the analytic result proved in Chapter 4 is that when agents have a positive endowment, they are able to hold no shares, where as in the case when their endowment is zero they will hold zero shares only asymptotically. When an agent has a positive endowment, he is able to hold no shares without facing death in the next period. In fact, an agent with positive endowment would have the choice to buy back shares at any time in the future. Of course if this agent's β is the lower of the two, then he will choose not to buy shares since he dislikes waiting to consume more than the agent with the larger β.

The convergence of the wealth distribution to the corners (i.e. the case when one agent holds all shares and the other hold none) happens very quickly. The larger the difference between the discount factors, the faster the convergence will occur. Typically it takes at most ten or twenty periods for the holdings to converge to the corners. Once the holdings have converged, the agents will continue to learn the equilibrium prices. Since there are only two dividends, the agents need only to learn two prices. They do so by fitting a linear pricing function to the data they observe using the updating scheme presented in (4.15)-(4.16).

The convergence of the pricing functions happens more slowly than the convergence of the holdings. Typically it will take at least one hundred periods for the pricing functions to converge, but it could take several hundred periods. The agents first converge to the same,

wrong pricing function, then they will learn the equilibrium prices together. Since the agents observe the same data and use the same updating scheme, once they agree on the pricing function they will never deviate from one another again.

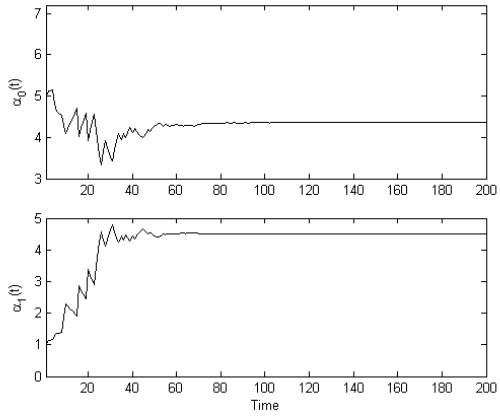
In Theorem 4.1 we saw that the equilibrium prices will converge to the REE prices corresponding to a homogeneous economy populated with agents with the largest discount factor. This makes sense since all other agents vanish in the limit and thus all that remains are the agents with the largest discount factor (i.e. it is a homogeneous agent economy). With a positive endowment, however, no agents will vanish and therefore the equilibrium prices will not converge to a prices corresponding to a homogeneous economy (since it remains heterogeneous).

The Case of $\gamma \neq 1$ and $e = 0$

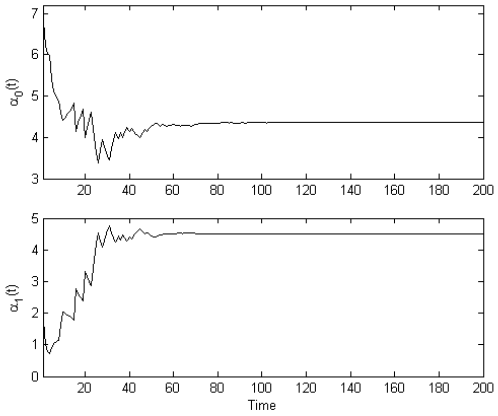
It makes sense that when agents have a positive endowment, the prices will not converge to the homogeneous REE pricing function of to the agent with the largest β since the agents with the smaller β values have not actually left the market. The market is not homogeneous, since there are two kinds of agents - one type with all the shares and an endowment and another type with no shares and an endowment.

When $e = 0$, however, the agents with the smaller β values do go broke and will vanish. Although these agents vanish only in the limit, at some point they will hold so little stock that they no longer have any practical impact on the market. Thus, the market is eventually composed of only one type of agent. It is reasonable, then, to expect that in the case of no endowment, that the market will converge to the homogeneous, REE prices corresponding to the agent with the largest β given by (4.3), regardless of the utility.

The result supports the hypothesis that agents with any utility and no endowment will converge to a homogeneous economy. The agents will trade to the corners and then their pricing functions will converge to the prices corresponding to the homogeneous agent case for the agent with the largest β. Figures 5.5 and 5.6 show the convergence of the pricing functions to the REE values as computed by (5.2)-(5.3), for the $\gamma = 0.5$ and $\gamma = 2.0$ cases, respectively.

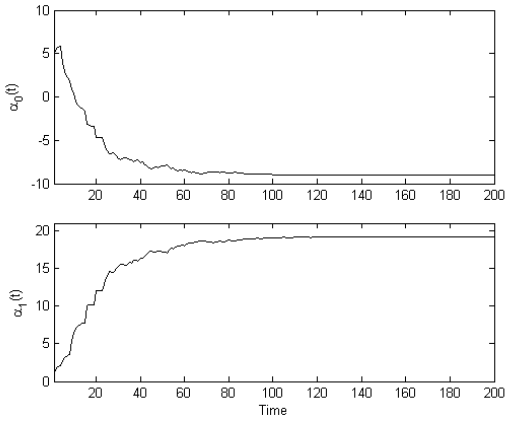


(a) Agent 1: $\beta = 0.9$

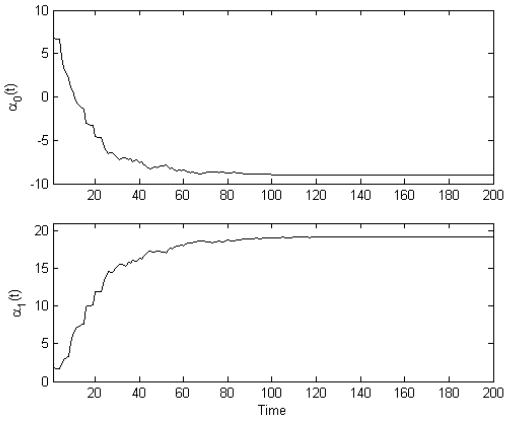


(b) Agent 2: $\beta = 0.8$

Figure 5.5: The evolution of the pricing coefficients for agents with $\gamma = 0.5$ and no endowment. Agents began with $\alpha^1 = (5, 1)$ ans $\alpha^2 = (7, 2)$ and converged to the REE price function $p(d) = 4.3571 + 4.50d$, as given by (5.2)-(5.3).



(a) Agent 1



(b) Agent 2

Figure 5.6: The evolution of the pricing coefficients for agents with $\gamma = 2.0$ and no endowment. Agents begin with $\alpha^1 = (5, 1)$ ans $\alpha^2 = (7, 2)$ and converged to the REE price function $p(d) = -9 + 19.2d$, as given by (5.2)-(5.3).

The Case of $\gamma \neq 1$ and $e > 0$

In the case when the agents are assumed to have non-log utility and a positive endowment, the results are very similar to the case of log utility and positive endowment. The convergence to the corners of the wealth distribution happens very quickly. Since the agents who converge to zero holdings do not ever leave the market, we do not expect the prices to converge to

the homogeneous REE prices.

We consider the case of $\gamma \neq 1$, and varied across agents, in Section 5.6.

The Case of e varied

When agents have different β's, the results are not affected greatly when the agents have different endowments. Depending on how the agents are set up (i.e. how the larger β and e are chosen), the agents may take more or less time to converge. The end result is always convergence to the corners. As when the agents have a positive endowment, we would not expect the market to converge to the homogeneous REE prices.

We consider the case of a varied endowment with constant β in Section 5.7.

5.4.1 Summary of Results on Varying β

In this section the agent with the largest β always holds all of the shares, while the other agents converge to zero holdings. Results similar to those shown in this section hold for cases when γ is held fixed and e and β are varied across agents. The amount of time to convergence may vary slightly, but the end result is always the same. Also, the results in this section are independent of the initial holdings of the agents, and the initial pricing functions they use.

Simply put, in all cases in which β varies across agents the agent with the largest β will hold all shares and all other agents will converge to zero holdings. Also, because the wealth distribution always becomes a constant, the agents eventually learn the equilibrium pricing function as a function of dividends alone.

5.5 Varying the Level of Risk Aversion

We now shift our attention to the risk aversion parameter, or γ. The following definition (Leroy and Werner, 2001) will be useful to explain the results in this section.

Definition 5.1. *Let y represent the long-run, average consumption level and let z be a risk such that $E(z) = 0$. The risk compensation, $\rho(y, z)$, is defined to be the solution to the equation*

$$E[u(y + z)] = u[y - \rho(y, z)].$$

It represents the amount of deterministic consumption the agent is willing to give up in exchange for being relieved of risk. In economics $\rho(y, z)$ is called the certainty equivalent of the risky consumption $y + z$.

For CRRA utility, no endowment and $d \in \{0.75, 1.25\}$ with equal probability, we have $y = 1$, $z = 0.25$ or -0.25 and

$$\rho = 1 - \left[\frac{0.75^{1-\gamma} + 1.25^{1-\gamma}}{2}\right]^{\frac{1}{1-\gamma}}. \tag{5.4}$$

The risk compensation increases in γ, so that as agents become more risk averse, they are willing to give up more deterministic consumption in order to be free of risk. For example, when $\gamma = 0.5$ we have $\rho = 0.0159$ but when $\gamma = 2$ we have $\rho = 0.0625$. Note that in the limiting case of $\gamma = 0$ we have $\rho = 0$, i.e. the agent is willing to give up nothing to be relieved of risk. An agent with $\gamma = 0$ would rather accept the risk, no matter how great, than give up consumption. For this reason we refer to an agent with $\gamma = 0$ as *risk neutral*.

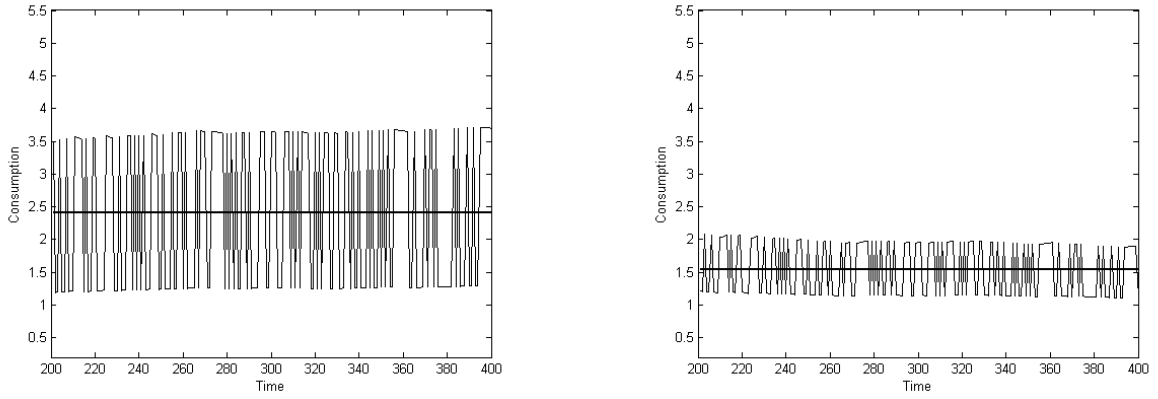
Let us now consider an economy with one risk neutral agent and one risk averse agent. Since the risk averse agent prefers constant consumption and the risk neutral agent is indifferent to risk, the risk neutral agent will allow the risk averse agent to smooth his consumption. That is, the risk neutral agent will, through trading, allow the risk averse agent to consume the same amount in every period. In other words, the risk neutral agent is insuring the risk averse agent against shocks to his consumption. In exchange for this insurance, the risk neutral agent will charge the risk averse agent a premium in the amount of ρ. Since he is willing to accept risk, the risk neutral agent gets to consume more, on average, over his lifetime.

In our model, we have assumed that all agents are risk averse and therefore there will never be opportunities for agents to fully insure against consumption shocks. There will, however, exist opportunities to partially insure. The larger an agent's γ, the more risk averse is that agent. Even though all agents dislike risk, some may dislike it more than others. This can be seen by the fact that the risk compensation is increasing in γ. In this case, the less risk averse agent (i.e. the one with the smaller γ) will act to partially insure the more risk averse agent, since the latter is willing to pay more for the insurance. The result will be the agent with the larger γ will have a consumption stream with a lower variance than the less risk averse agent. Since there is aggregate risk in our model, the agent with the largest γ

will have a variance in his consumption that is smaller than the variance of the aggregate dividend process. The other agent's will be larger. Since the more risk averse agent gets charged an insurance premium to lower the variance in his consumption, he will consume a smaller amount, on average, over his lifetime than the less risk averse agent will.

In Figure 5.7 we show the consumption stream for 200 periods for two agents. Agent 1 has $\gamma = 1.0$, while Agent 2 has $\gamma = 2.0$. The agents are otherwise identical with $\beta = 0.9$ and $\varepsilon = 1$. In this case we chose a dividend with more volatility, $d \in \{0.2, 1.8\}$. The reason for this is to ensure that there is a sufficiently difference between a good state and a bad. We can see in the graphs that Agent 1 has a much larger volatility in his consumption than Agent 2. Also, the average consumption is shown as a horizontal line and we can see that Agent 1 does consume more on average than Agent 2.

The average and variance for Agent 2's consumption are approximately 1.52 and 0.17, respectively, while the average and variance for Agent 1 are 2.4 and 1.44, respectively. The variance in the dividend stream over the same period was 0.642 so what we observe is consistent with the discussion above.

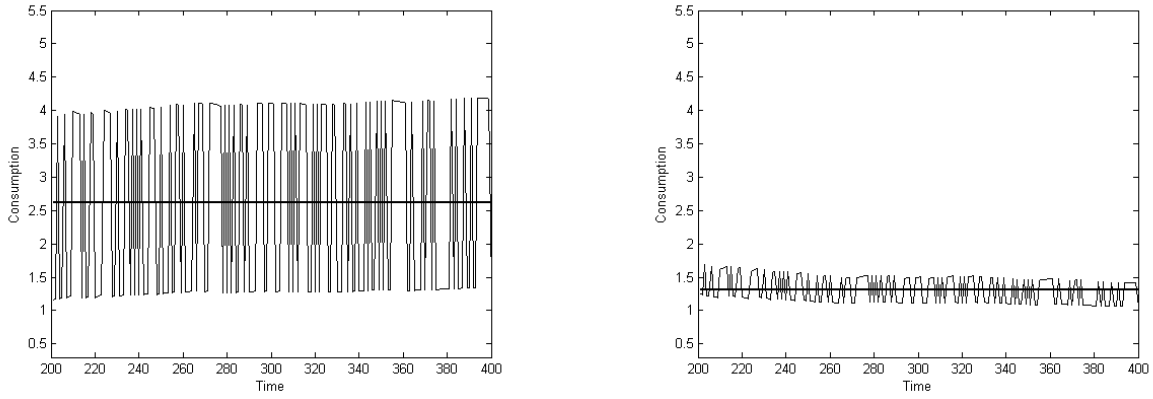


(a) Agent 1: $\gamma = 1.0$ (b) Agent 2: $\gamma = 2.0$

Figure 5.7: The consumption streams of agents with $\gamma_1 = 1.0$ and $\gamma_2 = 2.0$. Agent 1 provides insurance to Agent 2 and thus reduces the variance in Agent 2's consumption.

In Figure 5.8, the same experiment is shown as in Figure 5.7 except that Agent 1 has been given a smaller γ. The result is less variance in the consumption of Agent 2 (0.067) and more in the consumption of Agent 1 (1.932), since Agent 1 is more willing to accept risk in this case than in the previous case. To compensate for this additional risk, Agent 1 now

consumes an average of 2.564 per period while Agent 2 consumes 1.403 per period.



(a) Agent 1: $\gamma = 0.5$ (b) Agent 2: $\gamma = 2.0$

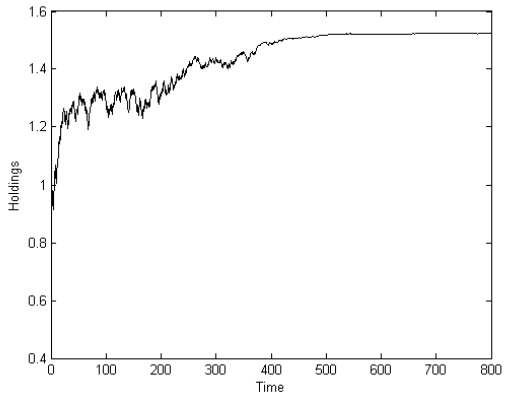
Figure 5.8: The consumption streams of agents with $\gamma_1 = 0.5$ and $\gamma_2 = 2.0$. The result is an lower variance in Agent 2 than was observed in Figure 5.7.

We now turn our attention to the equilibrium market clearing prices and wealth distribution. Unlike when the agents have different discount factors, in this case both agents continue to hold a positive number of shares in all periods. The wealth distribution reaches a steady-state, but away from the corner. Each agent consumes approximately the same amount in each state of the world. In addition, market clearing prices reach an equilibrium and since the wealth distribution becomes constant, the equilibrium prices do not depend on it. Agents are able to learn the REE without knowledge of the wealth distribution.

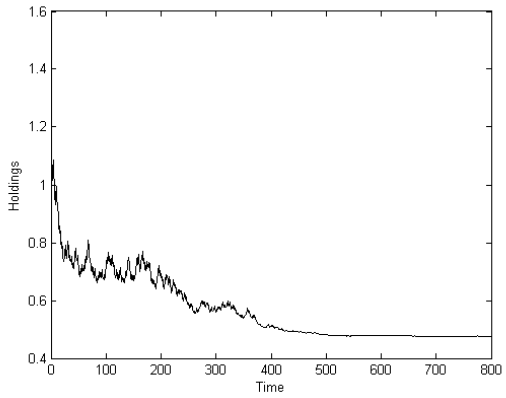
Figure 5.9 shows the convergence of the agents' holdings. In these plots, Agent 1 has $\gamma = 1.0$ and Agent 2 has $\gamma = 2.0$. Note that since there are assumed to be two shares of the stock and markets clear, the two plots in Figure 5.9 are perfectly (negatively) correlated. Figure 5.10 shows the convergence of the pricing functions.

Intertemporal Elasticity of Substitution

For CRRA utility, the inverse of the risk aversion parameter is known as the *Intertemporal Elasticity of Substitution (IES)* or the *Intertemporal Rate of Substitution*. The IES represents an agent's willingness to trade consumption in one period for consumption in another. In a sense, it represents the agent's personal assessment of intertemporal risk. For example, suppose the agent had an opportunity to sell shares of the stock today at a high price and

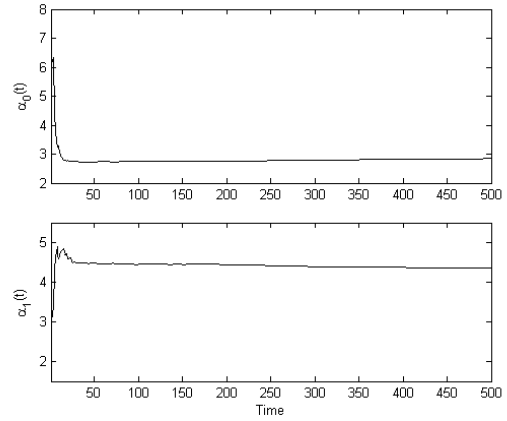


(a) Agent 1: $\gamma = 1.0$

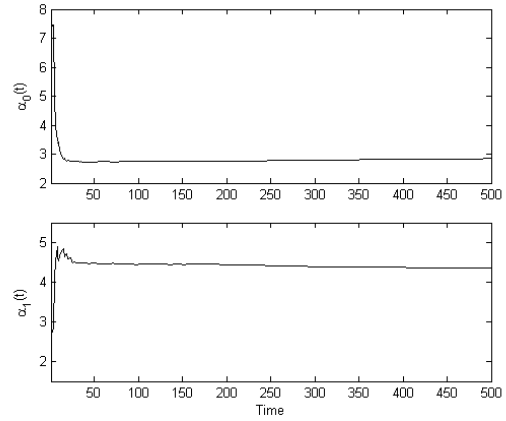


(b) Agent 2: $\gamma = 2.0$

Figure 5.9: The holdings of agents who are have different γ's. Both agents have $\beta = 0.9$, $e = 1$, and each initially held one share. The agents converge to $\mathcal{S} = (1.52.0.48)$.



(a) Agent 1: $\gamma = 1.0$



(b) Agent 2: $\gamma = 2.0$

Figure 5.10: The evolution of the pricing coefficients for two agents. Agent 1 has $\beta = 0.9$, $e = 1$ and $\gamma = 1.0$, while Agent 2 has $\beta = 0.9$, $e = 1$ and $\gamma = 2.0$. The agents started with $\alpha^1 = (5, 1)$ and $\alpha^2 = (7, 2)$, and converged to $p(d) = 2.852 + 4.351d$.

would have a chance to buy them back tomorrow at what he believes will be a lower price. An agent with a high IES (therefore, very willing to trade consumption in one period for the next) would execute a trade. He sells his shares today for additional consumption on the chance that he will be able to buy them back later at a cheaper price. The risk associated with selling for consumption today lies in the chance that the price will actually be higher in the future. If so, the agent may have permanently reduced his future consumption in all

periods. An agent with a low IES would not take this risk and would therefore not trade.

In the simulation shown in Figure 5.9, Agent 1 began trading with an initial pricing function of $p^1(d) = 5 + d$, while Agent 2 began with $p^2(d) = 7 + 2d$. Thus, Agent 2 has a more optimistic view about the future price of the stock. Given that Agent 1 has the lower γ, he has the higher IES, and is therefore more willing to sell his shares and consume today. That is, he is willing to possibly forego consumption in the future in order to consume today. Also, the market clearing price in the first period was 8.12, so from Agent 1's point of view he expects to be able to buy shares in the next period cheaper than he sold them for. In addition, Agent 2 expects the price to be higher in the following period than it is currently so he is willing to buy. This explains why Agent 1 sells his shares initially in Figure 5.9.[2]

5.6 Varying the Discount Factor and the Risk Aversion

In the previous sections, agents differed only in their discount factors or risk aversion parameters and pricing functions (although the latter is irrelevant since the agents will always converge to the same pricing function). We now briefly investigate whether similar results hold when, in addition to variations in β, we also vary γ across agents.

The results are very similar to what we observed in the cases discussed earlier in this chapter: The agent with the largest discount factor will hold all of the shares while the other agents will hold zero. We are unable to obtain different results in any case when agents had differing discount factors. Differences in β dominate all differences in other parameters.

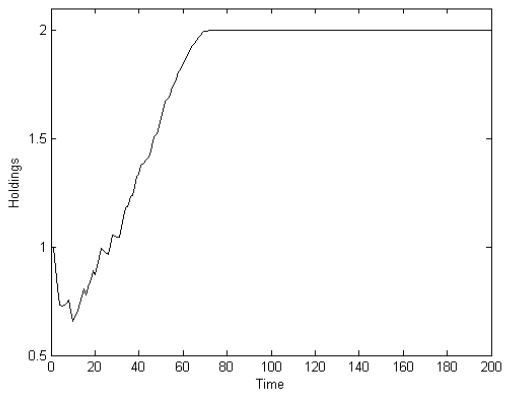
As we discussed in Section 5.5, a small value of γ will cause the agent to be more willing to sell some of his holdings to an agent with a larger value of γ. On the other hand, an agent with a large β values future payments highly and thus would like to hold as many shares as possible. We might expect, then, that an agent with a large β and small γ would be interesting to observe since there will be forces acting in opposite directions.

To investigate this we run a simulation as follows. As before, there are two agents and two dividends. Agent 1 has a $\beta_1 = 0.9$ and $\gamma_1 = 0.05$, and Agent 2 has $\beta_2 = 0.89$ and $\gamma_2 = 3.0$. Thus, the two agents differ only slightly in their discount factor, but differ greatly in their level of risk aversion. Our goal is to see if we can construct a case in which agents

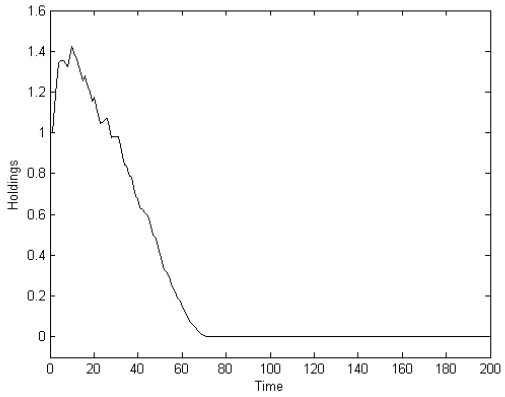
[2]Note that the agents began with one share each and trade immediately. This is not clear in the graphs due to the scale.

have different discount factors, but do not end up at the corners with respect to the wealth distribution.

Figures 5.11 and 5.12 show that the agents eventually trade themselves to the corners and remain there. There is trading initially, due to the large difference in the γ's, but over time the agent with the larger β dominates. Note that the agents were given the same initial pricing function of $p(d) = 5 + d$ to make sure that they traded only due to differences in their risk tolerance and discount factor. If they had different views on future prices then that would also cause them to trade.



(a) Agent 1: $\beta = 0.9$ and $\gamma = 0.05$



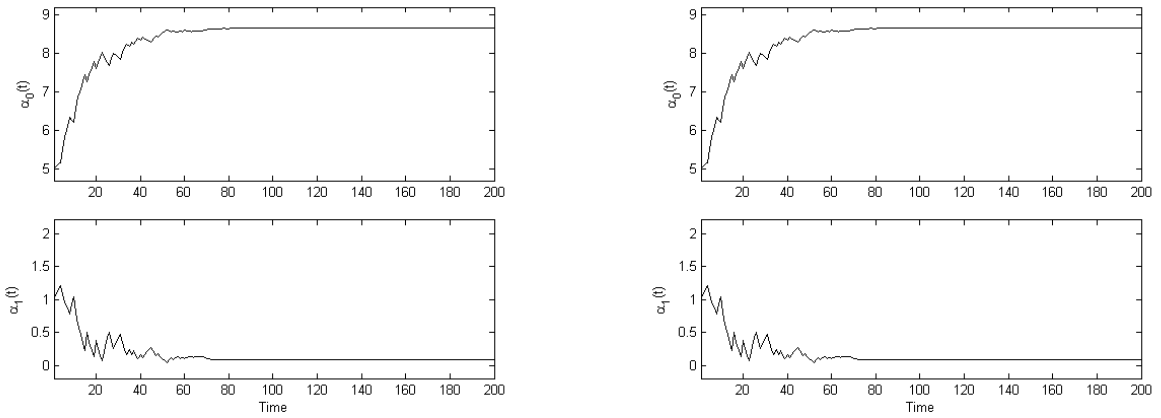
(b) Agent 2: $\beta = 0.89$ and $\gamma = 3.0$

Figure 5.11: The holdings of agents who are have different β and γ. Both agents have $e = 10$, and each initially held one share.

It should be noted that, as we have seen previously, part of the reason that we see trading in many cases is that agents have an endowment. In the case considered in this section, for example, the agent with the lower γ is very willing to sell his shares to the other agent since it will not significantly impact his future consumption. If the agents did not have endowments, the result would be similar, but less extreme. The agent with the lower γ would still trade initially, but he would not trade as much.

5.7 Varying Endowments

We now investigate the effects of different per period endowments across agents. Agents will be assumed to be identical in all ways except for their endowments. We saw in Section



(a) Agent 1: $\beta = 0.9$ and $\gamma = 0.05$

(b) Agent 2: $\beta = 0.89$ and $\gamma = 3.0$

Figure 5.12: The pricing coefficients of agents who have different β and γ. Agents both began with $\alpha = (5, 1)$ and converged to $p(d) = 8.660 + 0.080d$.

5.4 that different endowments have little effect on the outcome of the market if agents have different discount factors. Before proceeding, a brief discussion of risk aversion is needed.

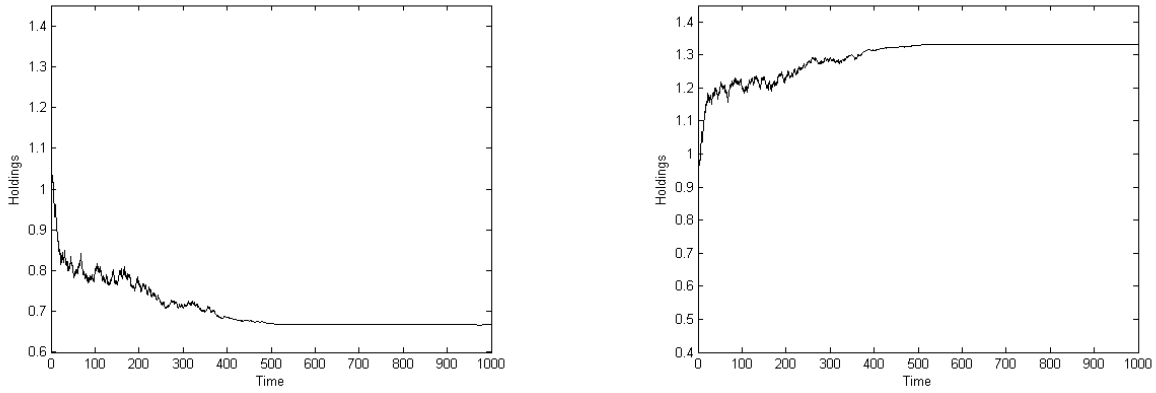
Agents in our model have Constant Relative Risk Aversion (CRRA) utility. CRRA utility means that regardless of wealth, agents will have the same attitude toward risking a given percentage of their wealth. For example, an agent with \$100 will view a \$1 bet the same as an agent with \$1000 would view a bet of \$10. It is also true that our agents have Decreasing Absolute Risk Aversion (DARA) utility. DARA utility means that as agents become wealthier, they become less risk averse, in *absolute* terms. This makes sense intuitively. Certainly we would expect that an agent with \$100 would be more averse to making a \$10 bet than an agent with \$1000 would. For further discussion on risk aversion, see Leroy and Werner (2001).

We now return to our discussion of the effect of varying the agents' endowments. Since the endowment represents income in all periods (with certainty), the agent with the larger endowment is the wealthier agent. Thus, he will also be less risk averse, in absolute terms. Therefore, agents with different endowments (who are identical in all other ways) will interact similarly to agents with the same endowment and different γ's. The agent with the larger endowment would represent the smaller γ and thus, he will attempt to partially insure the agent with the smaller endowment.

Similar to the results obtained in Section 5.5, the agent with the larger endowment will

allow the agent with the smaller endowment to reduce the variance in his consumption stream. In exchange for this reduction in variance, the less risk averse agent will consume more, on average, over his lifetime. The agents eventually reach a steady-state wealth and they learn the equilibrium prices.

The convergence of the agents holdings is shown in Figure 5.13. The convergence of the pricing functions is shown in Figure 5.14. Each agent has $\beta = 0.9$ and $\gamma = 1$ and both began with a single share. It is interesting to note that the agents learn the market clearing prices quite quickly, while it takes significantly longer for them to converge to the steady-state wealth distribution.



(a) Agent 1: $e = 1$

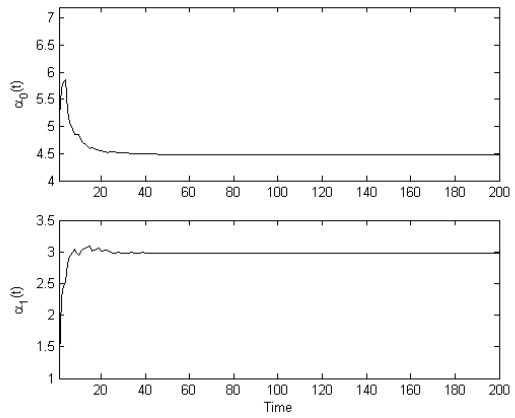
(b) Agent 2: $e = 2$

Figure 5.13: The holdings of agents who are have different endowments. Both agents have $\beta = 0.9$ and $\gamma = 1.0$ and initially held one share. The agents trade immediately away from one.

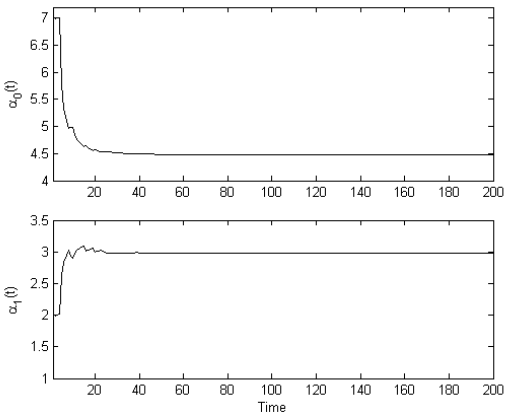
5.8 Varying the Initial Holdings

So far in this chapter, the initial holdings of our agents have been irrelevant. When agents have different discount factors, the agents will converge to the corners regardless of what holdings they begin with. When agents have different levels of risk aversion, or per period endowments, they will converge to an interior steady-state wealth distribution. In this section the agents begin the simulation identical, except for their initial holdings.

When $\gamma = 1$ and $e = 0$, Theorem 4.1 shows that the agents will not trade. Let us consider, then, the case when $\gamma = 1$ and $e > 0$, for both agents, and have the agents begin with 0.1 shares and 1.9 shares. The result is that the agents will converge to $\mathcal{S} = (1, 1)$,



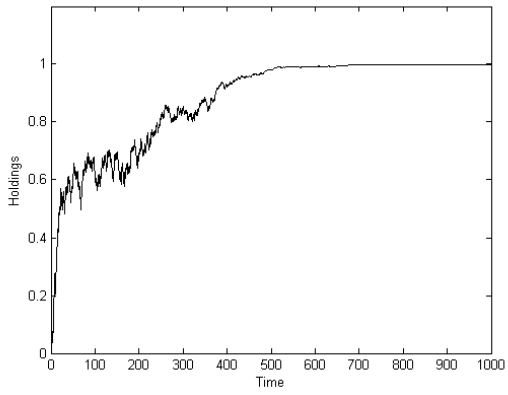
(a) Agent 1: $e = 1$



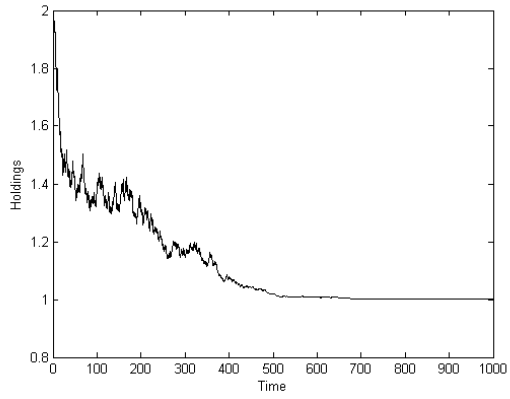
(b) Agent 2: $e = 2$

Figure 5.14: The pricing coefficients of agents who have different endowments. These agents both have $\beta = 0.9$ and $\gamma = 1.0$. Agent 1 began with $\alpha = (5, 1)$ and Agent 2 began with $\alpha = (7, 2)$, and both converged to $p(d) = 4.476 + 2.984d$.

that is, the equilibrium wealth distribution is constant, and each agent holds a single share. Figure 5.15 shows the evolution and convergence of the agents' holdings.



(a) Agent 1: $s_0 = 0.1$



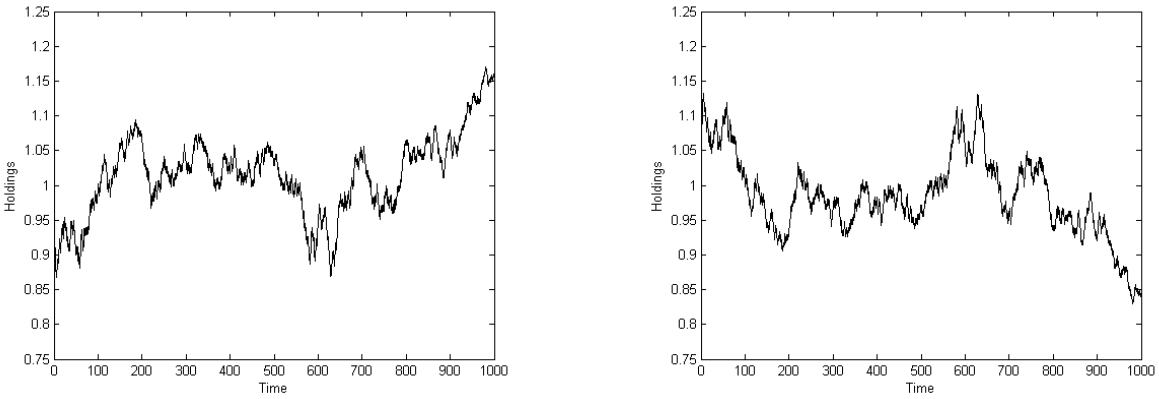
(b) Agent 2: $s_0 = 1.9$

Figure 5.15: The evolution of holdings for agents who begin with different holdings. Both agents have $\beta = 0.9$, $\gamma = 1.0$ and $e = 1$. Agent 1 began with 0.1 shares and Agent 2 began with 1.9 shares. The agents converge to one share each.

5.9 An Open Problem

We saw in Sections 5.5 and 5.7 that agents with different levels of risk aversion or different per period endowments will converge to a steady-state, interior wealth distribution. In the cases shown, the agents are assumed to have relatively small endowments, specifically, 1 or 2. In this section we will consider similar cases but for larger endowments. In the case of different risk aversion parameters, we use $e = 10$ and in the case of different endowments we use $e = 5$ and $e = 10$.

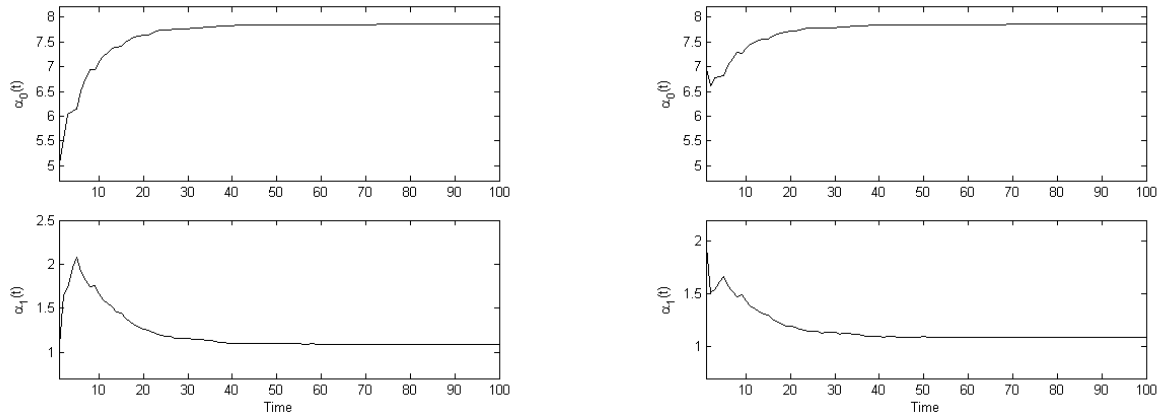
The results are surprising. The agents in these cases will fail to converge to a steady-state wealth distribution and will continue to trade forever. Prices, however, reach an equilibrium and the agents are able to learn them. In Figures 5.16 and 5.17 we show the agents' share holdings and pricing coefficients in the case when $\gamma_1 = 1.0$ and $\gamma_2 = 2.0$, with $e = 10$. The situation where the endowments are varied looks similar.



(a) Agent 1: $\gamma = 1.0$ (b) Agent 2: $\gamma = 2.0$

Figure 5.16: The holdings for agents with different levels of risk aversion and a large endowment. Both agents have $\beta = 0.9$, $e = 10$ and initially held a single share.

It is not immediately clear whether this result is due to some underlying economics or whether it is simply a numerical issue. With a large endowment and little variance in the dividend process, there is not a big difference between a high state and a low state. That is, since the agents are receiving an endowment of 10, they are relatively indifferent between a dividend of 0.75 and 1.25. This could result in the market being unable to correctly determine the price, which means that agents will converge to the wrong pricing function and hence the



(a) Agent 1: $\gamma = 1.0$ (b) Agent 2: $\gamma = 2.0$

Figure 5.17: The pricing coefficients for agents with different levels of risk aversion and a large endowment. Both agents have $\beta = 0.9$, $e = 10$ and initially held a single share. Agent 1 began with $\alpha = (5, 1)$ and Agent 2 began with $\alpha = (7, 2)$, and both converged to $p(d) = 7.855 + 1.085d$.

wrong demand function also. In the cases considered with smaller endowments, the agents have a unique fixed point in their equilibrium demand functions and thus trading halts. In the case of a larger endowment, they may have a different fixed point depending on which dividend is called out. This means that they are constantly being attracted to a different point and thus will continue to trade forever.

5.10 Global Convergence

For each case we have considered in this chapter, we have shown numerically that the agents will learn the equilibrium prices. In each case we gave the agents an initial pricing function that, for all practical purposes, was randomly chosen, since we did not know ahead of time what the true function would look like. The purpose of this section is to consider a case in which we know the true REE pricing function and to systematically check the convergence from various starting values. Our goal is to convince ourselves that the algorithm is *globally convergent*. That is, the algorithm will converge to the same equilibrium regardless of what initial pricing functions the agents are given.

While our goal in this section is to investigate global convergence, we need to be sure that the pricing functions the agents begin with at least makes sense. In particular, the agents should never forecast prices to be zero or negative. Also, the function should be strictly

increasing in d. It would not make sense for agents to believe that a higher dividend would lead to a lower price.

To investigate this, we consider a simple market. Since we are not interested in trading, but only in learning the market clearing prices, there is no need to have heterogeneous agents. Thus, we could assume there are two identical agents, or equivalently, a single agent. It may seem strange to consider market clearing prices when the market has only one agent, but nothing in the mechanics of our market requires there be more than one agent. Having two identical agents instead of one does nothing but double the computational time.

We assume there are the two usual dividends, 0.75 and 1.25, with equal probability. The agent has $\beta = 0.9$, $\gamma = 1$ and $e = 0$. Since there is only one agent, he must start with one share and hold it in every period. The market clearing price will be adjusted in each period so that the agent is happy holding his single share. The REE pricing function in this case is $p(d) = 9d$.

Since the agent is only learning two prices (or equivalently, the parameters of a linear function), we can imagine the evolution of the their forecasting rule as a particle moving in the plane. The REE point in is $(0, 9)$. We start the agent at 4 different starting values, each 6 units from the solution, in different directions. Specifically, the starting values are $(-6, 9)$, $(0, 15)$, $(6, 9)$ and $(0, 3)$. The trajectory of each particle is shown in Figure 5.18.

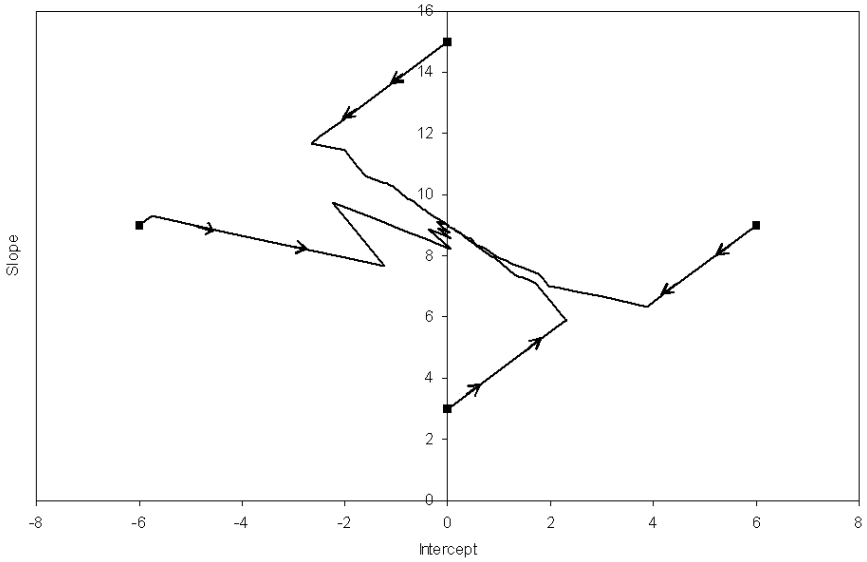


Figure 5.18: The convergence of the pricing coefficients for four different starting values.

We offer no proof of global convergence beyond the numerical demonstration shown in Figure 5.18. A similar exercise could be carried out for any case we desire. Practically speaking, we have never come across a case for which we were not able to achieve convergence. Therefore, it is our conjecture that the REE is a global attractor and any initial pricing function will converge to it under learning.

5.11 The Market Clearing Price as a Contraction

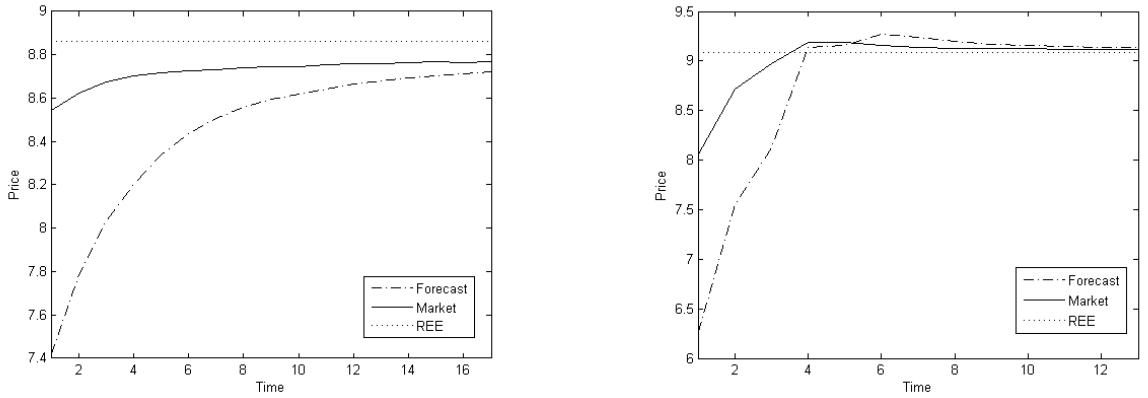
The conjecture that the REE is a global attractor is a strong claim. In order for this to be so, there must exist some underlying contraction that guides the market toward the REE. In this section we will show, via example, that this contraction lies in the market clearing price mechanism.

We consider the following setup for the model. Suppose we have two agents and two dividends. The first agent has $\beta = 0.9$ and $\gamma = 0.05$, while the second agent has $\beta = 0.89$ and $\gamma = 1.0$. Both agents have no endowment. This situation is similar to the case we saw in Section 5.6. The reason we chose this particular case was to ensure that we had a sufficient number of time steps in which agents were away from equilibrium. Since the agents have no endowment, they will not converge too quickly to the corners.

Our goal is to show that the market clearing price acts as a contraction on the forecasted prices. We have assumed that both agents begin with the same pricing function so that their forecasts will be the same. This eliminates the need to weight the respective forecasts of the agents. In Figure 5.19 we show the forecasted price, the market clearing price, and the REE price for the low and high dividends. In the low dividend case, the market clearing price can be seen to always be between the forecast and the REE. In the high dividend case, the picture is not quite as elegant. The forecast crosses the REE line at one point, but the market clearing price does eventually guide it back.

5.12 Summary and Discussion

Via simulation, we computed the equilibrium stock prices when agents are fully heterogeneous. When the agents differ in their discount factors, the agents with the lower discount factors eventually hold no shares of the stock. If there is no endowment, we get a truly homogeneous agent economy in which we know the REE pricing function. Agents are found



(a) Low dividend

(b) High dividend

Figure 5.19: The forecasted price, market clearing price and REE price for each dividend.

to correctly learn this function. When endowments are positive, no agents leave the market and it never converges to a homogeneous agent economy. Nonetheless, the agents are still able to compute the equilibrium prices even though we are not able to compute them analytically.

When agents had the same discount factor, but differed in other ways, the results were more interesting. With different levels of risk aversion or endowments, one agent is permanently less risk averse than the other and is therefore willing to partially insure the other agent. The result is more variance and a higher average consumption for the agent who is less risk averse. The agents are still able to reach a steady-state wealth distribution and learn the equilibrium prices.

Finally, when otherwise identical agents with positive endowments begin with different initial holdings they converge to a constant holdings distribution in which they each hold one share. The convergence is slow and non monotone, because the agents only differ in their wealth slightly and therefore, they trade only small amounts at a time. As the agents approach holdings of one, they become less risk averse relative to each other. This causes them to trade even less. The result is a random walk that slowly converges to one.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have constructed a simple, yet robust artificial stock market model. The model is simple in that it contains a single risky asset and no bond. On the other hand, we make no equilibrium assumptions a priori and our agents are allowed to be heterogeneous in any way we wish. The model we have created in this work is extendable and can be used to investigate the answers to numerous other asset pricing questions, such as the risk premium puzzle, for example.

Agents are found to learn the equilibrium asset prices, regardless of the initial conditions or parameter choices. We conjecture, and offer a numerical justification, that the Rational Expectations Equilibrium is a global attracting point in the space of pricing functions. This is a very strong result since, in general, we are only able to analytically compute the REE in very special cases. Therefore, our model offers an alternative and robust method for computing the REE in a general context.

In addition to the convergence of the agents to the REE pricing function, we are also able to investigate the long-run stock holdings of the agents. Specifically, agents with different discount factors will converge to a constant distribution where the agent(s) with the largest discount factor hold all of the shares and the other agents hold none. In the case when agents differ in their risk aversion or endowment, we find that agents also converge to a steady-state wealth distribution, but it is not the corner solution as in the case of variable discount factors. In these cases it is possible (though likely not required) that agents will converge to an interior wealth distribution. Finally, otherwise identical agents with different initial holdings will converge to a homogeneous agent economy.

We have, by design, made very few assumptions in our model. We have not imposed any restrictions on the heterogeneity of the agents, nor have we made any equilibrium

assumptions. The main assumptions we have made and may wish to generalize are that of a single asset, constant endowment, identically distributed dividends and a shared learning rule across agents.

The easiest assumption to relax is that of identically distributed dividends. As shown in the Appendix, many of the numerical methods we use were built to accommodate the assumption of an AR(1) process. Thus, it would not be difficult to generalize the dividend process in our model. Recall, however, that we were able to use the fact that dividends were identically distributed in order to derive the analytic formula for the REE prices in the case of identical agents. Generalizing to an AR(1) process eliminates this convenient closed form solution.

The assumption of constant endowment was made simply to avoid the agents having to compute an additional expectation over their individual shocks. Allowing for idiosyncratic shocks, in addition to aggregate shocks, would likely result in more interesting price and holdings dynamics. It is possible that we could observe continuous trading in the presence of idiosyncratic shocks. However, adding idiosyncratic shocks does create additional design questions. Specifically, if agents are not assumed to know the distribution of idiosyncratic shocks, then they will be required to learn that in addition to the pricing function. If we assume the agents know the distribution of individual shocks, then this extension should be relatively straightforward. An example of an idiosyncratic shock could be that the agents receive a high or low endowment, with a given probability.

The biggest assumption we made, and therefore the most difficult to relax, is that of a single risky asset. Ideally we would like to have several risky assets and a risk-free bond (with a market determined interest rate). However, each additional asset that we add brings with it (at least) two additional state variables, one for the amount of the asset the agent is holding and one for the price of the asset. It the asset pays dividends, then that is one more variable. The result is a state space that is just too large to handle in any reasonable amount of time.

There is one promising idea that we have considered to allow us to handle more assets. Throughout this manuscript, the agent has used a three dimensional state space to construct all possible wealth levels for himself. For each holding, dividend and price, there is a corresponding wealth. From the agent's point of view it is irrelevant where the wealth came from, as long as he knows how much he can spend on his investments and consumption.

Thus, in an effort to eliminate a state variable we can have the agent consider a space of wealth and price only. In the case of n assets the state space would be wealth and n prices. This would significantly reduce the state space (instead of $3n$ state variables we would have $n + 1$) and possibly allow us to simulate such a model in a reasonable amount of time.

The model we have developed in this work provides an important generalization to asset pricing problems. Constructing the model in the way we have allows us to consider asset pricing problems that previous approaches were unable to accommodate. Even in its simplest form we were able to obtain interesting and relevant results, including properties of the demand function, equilibrium asset price and holdings behavior and an analytic result in a special case. Further generalizing the model, as discussed above, will allow us to more adequately explain many of the current puzzles in asset pricing.

APPENDIX A

DETAILS OF THE COMPUTER CODE

The purpose of this appendix is to give a detailed summary of the steps taken to implement the model we have presented in this manuscript. We present here, not a listing of code, but rather a user manual which could be used in conjunction with the code to understand our implementation.

We chose to implement our model using the Java programming language. There were several reasons for this. Initially, we intended to use a Java package called RePast, that is specifically intended for use with agent-based models. Modelers are able to simulate their model and watch the agents and market variables evolve over time. Although this is a very power and convenient tool, in the end we chose not to use it. Instead, we wrote our own class to run the model and write the output to files. We did this simply because it was easier at the time than learning how to use all of the sophisticated features of RePast.

Another reason for using Java is its ability to implement Object Oriented Programming (OOP). Often in numerical analysis OOP is avoided in favor of more linear programming languages such as Fortran or C. But the fact is, in a model such as ours objects occur so naturally that it would not make sense to implement it in any way but using OOP. For example, an agent is nothing more than an encapsulation of data (β, γ, etc.). It is therefore very convenient to have an Agent object that holds all of this data in the same place.

The following sections assume the reader is somewhat familiar with the Java language and syntax (for those familiar with C++ it is very similar). We note here that the classes described below represent the way we chose to solve our model. In no way are we claiming that this is the only way to solve it, nor are we asserting it is the best way. This project was a learning experience for everyone involved. In the end, however, it works.

AssetPriceProj/assetPricePackage/BCKAgent.java

Purpose

This class stores our agent and all the information about him.

Dependencies

Requires: `BCKMarket, LearningRule, PricingFunction, PricingPolynomial, PricingPowerFunction ThreeDArray, java.util.*`

Class Variables

`agentNumber` - `int` - the agents ID number
`beta` - `double` - the agent's discount factor (default 0.95)
`gamma` - `double` - the agent's risk aversion parameter (default 1.0)
`lambda` - `double` - controls the agent's learning rate in `UpdateParams` (default 1.0)
`endowment` - `double` - the agent's constant per period endowment (default 20.0)
`smin,smax` - `double` - limits on the agent's `s` grid (defaults 0, 2)
`numStocks` - `int` - size of the holdings grid (default 201)
`sGrid` - `double[]` - the agent's holdings grid
`stocksShares` - `double` - the initial holdings of the agent (default 1.0)
`policyFunction` - `ThreeDArray` - agent's policy
`valueFunction` - `ThreeDArray` - agent's value function
`pricingFunction` - `PricingFunction` - the agent's pricing function
`lRule` - `LearningRule` - the agent's rule for updating the pricing function
`market` - `BCKMarket` - a market object to hold all market variables (all agents have the same one)

Constructors

In each of the constructors below the user can specify different parameters. Anything not specified we use the default values and/or accessor methods to change them. The default `LearningRule` is `Regression(2)`.

`BCKAgent(BCKMarket m, int aNumber, intGridSize, PricingFunction pf, LearningRule L)`

`BCKAgent(BCKMarket m, int aNumber)`

`BCKAgent(BCKMarket m, int aNumber, LearningRule L)`

`BCKAgent(BCKMarket m, int aNumber, PricingFunction pf, LearningRule L)`

`BCKAgent(BCKMarket m, int aNumber, ThreeDArray policyFun, PricingFunction pf, LearningRule L)`

`BCKAgent(BCKMarket m, int aNumber, PricingFunction pf, int sGridSize, int smin, int smax)`

`BCKAgent(BCKMarket m, int aNumber, int sGridSize, int smin, int smax)`

Functions

`void setUpSGrid()` - Uses `smin`, `smax` and `sGridSize` to create the holdings grid.

`void setDefaultPolicy()` - Initializes the policy function to be 1 in all states and the value function to be `(1/1-beta)*u(1+endowment)`.

`void updateStockShares(double price)` - Called after the market clearing price is determined. Creates a demand function object and evaluates it at `price`. The result is the agent's new holdings.

`void updatePolicyFunction(BCKLucas lucas)` - Calls the grid search routine in `BCKLucas` and sets the agent's new policy.

`void updatePolicyFunction(BCKLucasSpline lucas)` - Calls the grid search routine in `BCKLucasSpline` and sets the agent's new policy.

`void updatePricingFunction()` - Calls the static method `UpdateParams.pricing` and stores the new `PricingFunction`.

`double u(double c)` - The agent's utility. Assumed to be CRRA.

`double up(double c)` - The agent's marginal utility. Used in checking first order conditions.

`double ptilde(double d)` - Evaluates the agent's forecast of price for dividend `d`.

`PricingFunction calcREEPricingFunction()` - In the case when agents are identical we have a closed form solution for the REE pricing function. This routine computes it and stores it as a `PricingFunction`. If `gamma` is 1 then it is stored as a `PricingPolynomial`, otherwise as a `PricingPowerFunction`.

`PricingFunction getPricingFunction, void setPricingFunction(PricingFunction pf)` - Accessor functions for the pricing function.

`ThreeDArray getPolicyFunction(), void setPolicyFunction(ThreeDArray g)` - Accessor functions for the agent's policy function.

`ThreeDArray getValueFunction(), void setValueFunction(ThreeDArray v)` - Accessor functions for the agent's value function.

`int getAgentNumber()` - Returns the agent's unique ID number.

`double getXXXX(), void setXXXX(double x)` - Accessor functions, where XXXX is `Gamma, Beta, StockShares, Lambda, Endowment`.

`double[] getSGrid(), double[] getDGrid(), double[] getPGrid(), int getSizeStockGrid()` - Accessor functions for the state space grids.

`void setLearningRule()` - Sets the agent's learning rule.

`double getLastHoldings()` - Returns last period's holdings.

AssetPriceProj/assetPricePackage/BCKMarket.java

Purpose

This class contains all variables that are common to all agents. Intuitively we would expect that the market class contains agents, but this is actually backwards. The agents hold a market object and it is the same for all agents.

Dependencies

Requires: `AggDemand, DemandFunction, MidpointSolver, Tauchen, java.util.ArrayList`

Class Variables

`numDivs` - `int` - size of the dividend grid (default 3)
`numPrices` - `int` - the size of the price grid (default 201)
`pmin,pmax` - `double` - upper and lower bounds on the `p` grid (defaults 1.0, 30.0)
`div` - `double[]` - dividend grid
`Pr` - `double[][]` - transition probability matrix
`priceDivHistory` - `ArrayList` - holds the price and dividend history
`historyLength` - `int` - number of periods in the history

The variables below are for an AR(1) dividend process. Currently these are ignored.
`rho` - `double` - the AR(1) parameter (default 0.0)
`sigma` - `double` - the standard deviation of the shocks (default 0.1)
`numSDs` - `double` - the number of standard deviations to include in the discretiztion (default 3.0)

Constructors

All constructors initialize `priceDivHistory` and calls the functions `setUpDivsProbs`, `setUpPGrid` and `initializeHistory`.

`BCKMarket()` - All values are default.

`BCKMarket(double rho, double sigma, double numSDs, int numDivs)` - Sets up the market for an AR(1) dividend process and uses the default price grid.

`BCKMarket(double[] d, double[] pr, int priceGridSize)` - For use in the iid case. The probability matrix is populated so that all dividends are equally likely regardless of current dividends. Uses the default price grid.

`BCKMarket(double[] d, double[] pr, int priceGrisSize, double pmin, double pmax)` -
User specifies upper and lower limits on the price grid.

`BCKMarket(double[] d, int priceGrisSize, double pmin, double pmax)` - Same as previous, but assumes all dividends are equally likely.

`BCKMarket(double[] d, double[] pr, double[] prices)` - Same as above, but user specifies a grid of prices.

`BCKMarket(double[] d, double[][] Pr)` - For non iid dividends with a user specified probability matrix. This allows us to specify any probability matrix we wish.

Functions

`void setUpDivsProbs()` - Sets up dividend vector and probability matrix using the Tauchen class to compute nodes and probabilities.

`void setUpDivsProbs(double[] d)` - Sets up the dividend vector and assumes all dividends are equally likely.

`void setUpDivsProbs(double[] d, double[] prob)` - Sets up the dividend vector and the iid probability matrix.

`void setUpDivsProbs(double[] d, double[][] probs)` - Sets up the dividend vector and sets the probability matrix to probs.

`void setUpPGrid()` - Sets up the price grid using the upper and lower bounds, either given or defaults.

`void intializeHistory()` - Adds a dummy price/dividend pair to the history. This will be ignored later so it does not matter what we use.

`class BCKPriceDiv` - An inner class to hold a price/dividend pair. Has member functions `double getPrice()` and `double getDividend()` to extract the data.

`double drawNewDividend()` - Draws a new dividend given the transition probability matrix. Uses an inverse distributon method to choose a dividend given a uniform random variable.

`double calcMCPrice(BCKAgent[] agents, double dividend)` - Takes a list of agents and the current dividend and creates an `AggDemand` object that is passed to `MidpointSolver`.

`boolean addPriceDiv(double price, double div)` - Adds data to the market history.

`double[] priceDivHistoryVec()` - Returns the price and dividend history as a vector. The ordering is `(p1,d1,p2,d2,...)`.

`double[] getDividendHistory()` - Returns a vector of the dividend history only. Ignores the first point since it was not meaningful.

`double[] getPriceHistory()` - Returns a vector of the price history only. Ignores the first point since it was not meaningful.

`ArrayList getPriceDivHistory` - Returns price/dividend history.

`double getCurrentPrice(), double getCurrentDividend()` - Access the most recent price and dividend.

`double getPreviousPrice(), double getPreviousDividend()` - Access the last period price and dividend.

`int gethistoryLength(), double getRho(), void setRho(double r), double getSigma(), void setSigma(double s), double[] getDiv(), double[][] getPr(), int getNumDivs(), double getNumSDs(), double[] getPGrid(), int getNumPrices()` - Accessor methods.

AssetPriceProj/assetPricePackage/BCKLucas.java

Purpose

This class implements the discrete grid search method for computing the agent's policy function. This is done via value function iteration from which the policy function is a by-product.

Dependencies

Requires: `BCKAgent, IntDouble, Maximize, MaxFun, ThreeDArray, ThreeDIntArray, Vectors`

Class Variables

`veps` - `double` - tolerance on the value function norm (default 0.1)
`howardIter` - `int` - the number of Howard iterations to perform (default 8)
`maxPolicyIterations` - `int` - the number of iterations for which the policy functions remain unchanged to consider them converged (default 1)

Constructors

`BCKLucas()` - Default constructor, uses default values.

`BCKLucas(double vepsilon)` - User specifies `veps`.

`BCKLucas(double vepsilon, int hIter)` - User specifies `veps` and `howardIter`.

Functions

`ThreeDArray gridSearch(BCKAgent agent)` - Performs the main grid search routine. Within this function we instantiate a `MaxFun` object and use `Maximize.binMax` to compute the maximum. We continue to iterate until the policy functions remain unchanged for `maxPolicyIterations` steps and the value functions are changing by less than `veps`. Note that we have a contraction here so the norms on successive value functions should decrese monotonically.

`static ThreeDArray interateV(BCKAgent a, ThreeDArray v, ThreeDArray g, int n1, int n2, int n3, int howardIter)` - Performs the Howard Improvement Algorithm. Holds the policy function fixed and iterates `v` for `howardIter` times. Returns the iterated value function.

`void initialize(BCKAgent agent)` - Has no meaning. Gets overriden in `BCKLucasSpline`.

`void setTol(double e), double getTol()` - Accessor functions.

AssetPriceProj/assetPricePackage/BCKLucasSpline.java

Purpose

Implements the same algorithm as `BCKLucas`, but in a continuous setting using Schumaker splines.

Dependencies

Extends: `BCKLucas`

Requires: `BCKAgent, Maximize, MaxFunContinuous, ThreeDArray, SplineMatrix, Vectors`

Class Variables

geps - the tolerance on the policy function norm (default 0.001)
Inherits veps (default 0.1) and maxPolicyIterations (default 10) from BCKLucas.

Constructors

`BCKLucasSpline()` - Uses all default parameters.

`BCKLucasSpline(double veps)` - User specifies `veps` only. This function is obsolete and should not be used. One of the constructors below is more suitable.

`BCKLucasSpline(double veps, double geps)` - User specifies both tolerances.

`BCKLucasSpline(double veps, double geps, int policyIters)` - User specifies both tolerances and `maxPolicyIterations`.

Functions

`void initialize(BCKAgent agent)` - Runs a discrete grid search to get initial data to fit a spline to. This only gets run once at the beginning and it overrides the meaningless function in BCKLucas.

`ThreeDArray gridSearch(BCKAgent agent)` - Works similar to gridSearch in `BCKLucas`, but now instead of `MaxFun` it creates a `MaxFunContinuous` object and uses `Maximize.goldenSectionSearch` to find the maximum of the continuous function. We continue to iterate until the value function norm is less than `veps` and the policy function norm is less than `geps` for `maxPolicyOterations` times steps.

AssetPriceProj/assetPricePackage/BCKDriver.java

This class is rather outdated now and can be quite confusing to someone who is new to Java. It is not clear where to make changes to some of the agents' parameters since where to make those changes depends on which parameters one wishes to change. `BCKDriver2` is an easier class to use and is recommended.

AssetPriceProj/assetPricePackage/BCKDriver2.java

Purpose

This class actually runs all the steps necessary to simulate our artificial economy. It has all the agent and market parameters specified in the class variables. The class then sets up

the agents and runs the model. This class is the control center for all other classes we have discussed. All output in written to a series of csv files for easy manipulation.

Dependencies

Requires: `BCKAgent, BCKMarket, BCKLucas, BCKLucasSpline, CSVWriter, EulerResidual, PricingFunction, LearningRule, Vectors`

Class Variables

`numAgents` - `int` - how many agents we have
`timeSteps` - `int` - how long to run the model
`betas` - `double[]` - a vector of the agents' beta values
`gammas` - `double[]` - a vector of the agents' gamma values
`endows` - `double[]` - a vector of the agents' endowments
`holdings` - `double[]` - a vector of the agents' initial holdings
`useREE` - `boolean[]` - a vectors of `Boolean` indicating whether or not each agent should use the REE pricing function
`pricingCoeffs` - `double[][]` - an array of `double[]` indicating each agent's initial pricing coefficients. Ignored if `useREE` is `true` for a given agent.
`pGridSize` - `int` - the size of the price grid; same for all agents
`pmin, pmax` - `double` - bounds on the price grid; same for all agents
`sGridSize` - `int` - the size of the holdings grid; same for all agents
`smin, smax` - `double` - bounds on the holdings grid; same for all agents
`div` - `double[]` - vector of dividend values
`probs` - `double[]` - vectors of probabilities; assumes iid
`agentList` - `BCKAgent[]` - a array of the agents
`market` - `BCKMarket` - the shared market object
`lucas` - `BCKLucas` - an object for the gird search

Constructors

`BCKDriver2()` - Takes no arguments and calls `setup()` and `buildAgents()`.

Functions

`void setup()` - Creates the market, the agent list, the lucas object and opens a csv file for the market data.

`void buildAgents()` - Sets up agents one at a time. Each agent is given the parameters specified in the class variables, their pricing function is set, their learning rule (Bayesian) is initialized and their policy function is computed for the first time. csv files are opened for each agent.

`void executeTimeStep()` - Performs a complete time step. A dividend is drawn, the market clearing price is calculuated and the agents update their pricing and policy functions. All of the relevant data is written to the csv files.

`static void displayElapsedTime(long startMillis, long endMillis)` - Takes a start and end time in milliseconds and displays to the screen the time elapsed in hours, minutes and seconds.

`static void main(String[] args)` - Creates a `BCKDriver2` object and calls `executeTimeStep()` for each time period. Also, displays the total time elapsed from the start after each iteration.

AssetPriceProj/functions/AggDemand.java

Purpose

Takes an array of DemandFunction objects and aggregates them in order to compute a market clearing price.

Dependencies

Extends: `GeneralFunction`

Requires: `DemandFunction`

Class Variables

`demandList` - `DemandFunction[]` - An array of demand function objects.
`N` - `int` - The length of `demandList`

Constructors

`AggDemand(DemandFunction[] dl)` - Takes an array of `DemandFunction` objects.

Functions

`double evaluate(double p)` - Takes a `double` as input and returns the aggregate demand over all agents. Overrides the function `evaluate(double x)` in `GeneralFunction`.

AssetPriceProj/functions/Bayesian.java

Purpose

This class implements the adaptive learning rule that the agents use to update their pricing coefficients. Details can be found in Evans and Honkapohja (2001).

Dependencies

Extends: `LearningRule`

Requires: `Matrix`

Class Variables

`alpha` - `Matrix` - The pricing coefficients. Note this is a column vector stored as a matrix.
`x` - `Matrix` - The most recent observation of the dividend moments, $(1, d, \ldots, d^n)'$. Also a column vector stored as a matrix.
`S` - `Matrix` - The moment matrix of the dividend process.

Constructors

`Bayesian(double[] coeffs)` - Takes the initial coefficient vector as an array of doubles. Initializes the moment matrix to be the identity.

`Bayesian(double[] coeffs, double[][] R)` - Takes the initial coefficient vector as an array of doubles and an initial moment matrix.

Functions

`double[] update()` - Updates the agents pricing coefficients. The parent class, `LearningRule`, keeps a complete history of all dividends and prices. This function accesses the most recent dividend and price from the history and uses it to update the forecasting parameters. This function overrides the abstract method `update()` in `LearningRule`.

`Matrix xVec(double d)` - Given the dividend, d, this function computes $(1, d, \ldots, d^n)'$ and returns it as a column vector stored in a matrix.

`double[] getAlpha(), Matrix getMatrix()` - Accessor functions.

AssetPriceProj/functions/ComputePolicy.java

Purpose

This class is used to analyze the policy functions of the agents. No learning or market clearing takes place. The class only computes a policy function for a single agent.

Dependencies

Requires: `BCKMarket, BCKAgent, BCKLucas, BCKLucasSpline, PricingPolynomial, ThreeDArray, Matrix, Vectors`

Class Variables

All of the parameters needed to define the agent are stored as static class variables. The variable names are self explanatory and the code is very well commented in this class so we will not explain the variables here.

Constructors

`ComputePolicy()` - Default constructor. Calls the function `setupAll()` to set the agent's parameters.

`ComputePolicy(int ns, int np)` - Allows the agent to specify the number of grid points in the s and p directions. This is useful for doing error analysis when we want to investigate the effects of changing the grid size. Using a for loop we could compute may policy functions in a row with varying grid sizes. Also calls `setupAll()`.

Functions

`void setupAll()` - Sets all of the parameters for the agent and the market.

`void compute()` - Computes the policy function.

`Matrix fixIJ(int i, int j)` - Fixes the agent's holdings and dividend so the policy function is a function of price alone. Returns a matrix where the first column holds the prices and the second column holds the agents demand for the corresponding price.

`Matrix fixJK(int j, int k)` - Fixes the price and dividend so the policy function is a function of holdings alone. Returns a matrix where the first column has the agent's holdings and the second column holds the agents demand for the corresponding holdings.

`static void main(String[] args)` - The main routine. Creates a `ComputePolicy` object and calls `compute()`. Displays the result to the screen along with the time to compute it.

AssetPriceProj/functions/CSVWriter.java

Purpose

This class allows us to write output to a comma separated value (csv) file. This is very useful for keeping track of data when running simulations.

Dependencies

None

Class Variables

`fw` - `FileWriter` - used to write to a file
`pw` - `PrintWriter` - used to write to a file
`n` - `int` - number of fields in the csv file

Constructors

`CSVWriter(String filename, String[] fields)` - Takes as input a single string and one array of strings. The first argument is the name of the file the final csv will be written to.

The second argument contains the fields to include in the csv file. If any difficulties occur in accessing the files, `java.io` will throw exceptions.

Functions

`void write(double[] values)` - Writes values given in the input to the file, each time on a new line.

`void write(String[] values)` - Same, but for strings.

`void blankLine()` - Inserts a blank line in the csv file.

`void close()` - Closes the file. Files must be closed after use or else they will not open properly for viewing later.

`static void closeAll(CSVWriter[] list)` - A function that takes an array of `CSVWriter` objects and closes all of them. Note this is static function, so you do not invoke it on a particular object, but rather pass all the objects to it in an array.

Exceptions

`java.io` exceptions will be thrown if there are errors opening files. The most common error occurs when the file specified to write to is already open.

AssetPriceProj/functions/CubicSpline.java

Purpose

Implements a cubic spline to interpolate discrete data points. The notation was chosen to match that used in Kincaid and Cheney (2002), pp 350-354.

Dependencies

Requires: `TriDiagonalSolver`, `Vectors`

Class Variables

`t` - `double[]` - vector of independent variable values
`y` - `double[]` - vector of dependent variable values
`n` - `int` - number of points
`h,u,b,v,z` - `double[]` - intermediate values used in computing the spline

Constructors

`CubicSpline(double[] t, double[] y)` - User provides the data.

Functions

`double evaluate(double x)` - Returns the interpolated function value at `x`. If `x` is not within the limits of the original `t` vector, then `Vectors` will throw an exception.

`void computeZ()` - Computes the weights and nodes according to Kincaid and Cheney.

AssetPriceProj/functions/DemandFunction.java

Purpose

We use this class to determine the agent's demand for shares as a function of price only. Each agent will have a `DemandFunction` object, all of which will be passed to an `AggDemand` object. We use bilinear interpolation to evaluate the demand in the `s` and `p` direction.

Dependencies

Extends: `GeneralFunction`

Requires: `BCKAgent, BCKMarket, ThreeDArray, Vectors`

Class Variables

`g` - `ThreeDArray` - the agents full policy function
`s,d,p` - `double[]` - holdings, dividend and price grids
`holdings` - `double` - current asset holdings
`currentDiv` - `double` - current dividend
`M` - `BCKMarket` - a market object to access grids
`j` - `int` - the index of the current dividend
`i,k` - `int[]` - vectors of the indices that bracket current holdings and price
`t,u,g1,g2,g3,g4` - `double` - required for interpolation step

Constructors

`DemandFunction(BCKAgent A)` - Takes an agent and gets the current dividend from the market. Calls `setup(BCKAgent A)`.

`DemandFunction(BCKAgent A, double dividend)` - For use before the current dividend has been added to the market. Calls `setup(BCKAgent A)`.

Functions

`void setup(BCKAgent A)` - Sets the market, gets the grids and policy function from the agent. Also computes values needed for the bilinear interpolation that do not depend on price.

`double evaluate(double p)` - Returns an agent's demand for stock, given a price. Performs bilinear interpolation in the holdings and price directions. Overrides the method in `GeneralFunction`.

AssetPriceProj/functions/EulerResidual.java

Purpose

Computes the right hand side of the Euler equation to check if the agent's first order conditions are satisfied. Gets used in the driver classes.

Dependencies

Extends: `GeneralFunction`

Requires: `BCKAgent, BCKMarket, Vectors`

Class Variables

`a` - `BCKAgent` - an agent that gets passed in
`m` - `BCKMarket` - a market obtained from the agent
`d` - `double[]` - dividend vector
`Pr` - `double[][]` - probability matrix

Constructors

`EulerResidual(BCKAgent a)` - Takes an agent and extracts the market, dividends, and probabilities from it.

Functions

`double evaluate(double s, double sp, double div, double price, PricingFunction ptilde)` - Computes the right hand side of the Euler equation, i.e. `beta*E[...]`. Takes the agents previous holdings, `s`, new holdings, `sp`, the current dividend and price and the agent's current pricing function.

AssetPriceProj/functions/GeneralFunction.java

Purpose

This is a an abstract class that we use as a parent for many other functions. It may not be instantiated on its own. Having this parent class allows us create many different functions that are all the same "type".

Dependencies

None

Constructors

None

Functions

`double evaluate(???)` - There are many functions called evaluate in this class. They all differ in their input parameter and they all return 1. At least one of these functions must be overridden when using `GeneralFunction` as the parent class. The choices for parameters are currently `double x`, `double[] x`, `int x`, `int x, int y`, `int[] x` and `double[] x, double[] y, double[] z`.

AssetPriceProj/functions/IntDouble.java

Purpose

This class is used to hold a pair of numbers, one integer and one double. This is useful for maximization procedures that return a maximum function value and the index of the location of the max. It avoids the need for passing doubles and casting them into integers.

Dependencies

None

Class Variables

`integerPart` - `int` - stores the integer part of the pair
`doublePart` - `double` - stores the double part of the pair

Constructors

`intDouble()` - Default constructor, initializes both parts to be 0.

`intDouble(int i, double d)` - Initializes to have integer part equal to i and double part equal to d

Functions

`int getInt()`, `double getDouble()`, `void setInt(int i)`, `void setDouble(double d)` - Accessor functions.

AssetPriceProj/functions/LearningRule.java

Purpose

A parent class for all learning rules. In this class we store the complete dividend and price history for the market so the agents always have access to it. Extending classes must override the abstract method `update()`.

Dependencies

None

Class Variables

`d,p` - `double[]` - vectors to hold the history of dividends and prices
`T` - `int` - length of history
`N` - `int` - length of polynomial to fit to data
`resetTime` - `int` - how long to wait before resetting data

Constructors

None

Functions

`void setD(double[] newD)` - Sets the dividend history to be the dividends in `newD` from `resetTime` until the present.

`void setP(double[] newP)` - Same, but for prices.

`double getLastD(), double getLastP()` - Returns the most recent dividend and price.

`void resetData()` - Has agents ignore all points prior to now. This is a public method that can be invoked at any time and more than once. This function is not useful for Bayesian learning since in that case we only look at the last data.

AssetPriceProj/functions/MaxFun.java

Purpose

This class provides the function to maximize in the value function iteration step. In other words, this class computes the objective of the right hand side of the value function. The maximization step will be computed using `Maximize.binmax`. Note that this is the discrete version (i.e. without splines). See `MaxFunContinuous` for the spline version.

Dependencies

Extends: `GeneralFunction`

Requires: `BCKAgent, ThreeDArray, Vectors`

Class Variables

`agent` - `BCKAgent` - an agent object
`s,d,p` - `double[]` - grids of state variables
`Pr` - `double[][]` - the transition probability matrix for the dividend process
`i,j,k` - `int` - indices indicating the current state
`e` - `double` - the agent's endowment
`v0` - `ThreeDArray` - the agent's value function

Constructors

`MaxFun(BCKAgent agent, ThreeDArray v0, int i, int j, int k)` - Take an agent and a value function as well as the inidices to indicate the current state of the world. Obtains the remaining class variables through the agent.

Functions

`double evaluate(int m)` - Takes integer `m` which represents the index for `sp`. Returns the value of the objective if the agent were to chose `sp=s[m]`. We use bilinear interpolation in the `s` and `p` direction to allow those values to be off the grids. This function overrides the method in `GeneralFunction`.

AssetPriceProj/functions/MaxFunContinuous.java

Purpose

This class provides the function to maximize in the value function iteration step when we use splines. In other words, this class computes the objective of the right hand side of the value function. The maximization step will be computed using `Maximize.goldenSectionSearch`.

Dependencies

Extends: `GeneralFunction`

Requires: `BCKAgent, ThreeDArray, Vectors`

Class Variables

`agent` - `BCKAgent` - an agent object
`s,d,p` - `double[]` - grids of state variables

`Pr` - `double[][]` - the transition probability matrix for the dividend process
`i,j,k` - `int` - indices indicating the current state
`e` - `double` - the agent's endowment
`m` - `SplineMatrix` - a two dimensional array of Schumaker splines

Constructors

`MaxFun(BCKAgent agent, SplineMatrix m, int i, int j, int k)` - Takes an agent and a
`SplineMatrix` as well as the inidices to indicate the current state of the world. Obtains the remaining class variables through the agent. Note that the agent's value function is not needed here since it is incorporated into the `SplineMatrix` object.

Functions

`double evaluate(double sp)` - Returns the value of the objective if the agent were to chose `sp` as next period holdings. We use bilinear interpolation in the `s` and `p` direction to allow those values to be off the grids. This function overrides the method in `GeneralFunction`.

AssetPriceProj/functions/Maximize.java

Purpose

An abstract class containing many static routines used to compute the maximum of various types of functions, both discrete and continuous.

Dependencies

Requires: `IntDouble, GeneralFunction, MaximizeException`

Class Variables

None

Constructors

None

Functions

`static IntDouble binMax(GeneralFunction f, int[] x)` - Computes the maximum of a strictly concave function of a single variable over a vector of possible values. The method this function employs is a binary search. Note that in this version `x` is assumed to be integer valued. The function returns an `IntDouble`, where the integer part is the value of `x`

corresponding to the location of the max and the double part is the function value at the maximum.

`static double[] binMax(GeneralFunction f, double[] x)` - Identical to the above, except the max is taken over a vector of doubles. The return is two doubles, the optimal x and the optimal function value.

`static double[] goldenSectionSearch(GeneralFunction f double[] x)` - Computes the maximum of a strictly concave, continuous function of one variable. The vector `x` provides the upper and lower bounds on the maximization. The return is the optimal `x` value and the optimal function value.

`static intDouble(double[] x)` - Computes the maximum of a vector of doubles. Works the same as the function `max` in Matlab. The return is the index and the value of the maximum.

Exceptions

`IllegalArgumentException` - thrown if vectors are passed in of length zero
`MaximizeException` - thrown if max procedure cannot find a maximum after 100 iterations

AssetPriceProj/functions/MidpointSolver.java

Purpose

Computes the zero of a function of a single variable. The function must possess a unique root and be decreasing at the root. In this case if we evaluate the function and it is positive (negative), then the root must be to the right (left) of the current point.

Dependencies

Requires: `GeneralFunction, MidpointSolverException`

Class Variables

`initValue` - `double` - initial x value to try
`func` - `GeneralFunction` - the function to maximize
`tolerance` - `double` - at what point do we consider the function to be 0

Constructors

`MidpointSolver(GeneralFunction f)` - Default constructor. Sets `initValue` to 1.0 and `tolerance` to 1.0e-8.

`MidpointSolver(double iv, GeneralFunction f)` - Sets `initValue` to `iv` and sets `tolerance` to 1.0e-8.

`MidpointSolver(double iv, GeneralFunction f, double tol)` - Sets `initValue` to `iv` and sets `tolerance` to `tol`.

Functions

`double solve()` - Solves for x such that f(x) $<$ `tol`. Starting at `initValue`, we construct an interval and expand it until we have the root bracketed. If this is not possible an exception is thrown. Once the interval is found, we proceed using a binary search to find the root. If 100 bisections occur without success an exception is thrown. Returns the root, x.

Exceptions

`MidpointSolverException` - thrown for one of two reasons. The solver may have had trouble finding an interval bracketing the root, or the solver got stuck. Be sure the function has a single root and is decreasing near it.

AssetPriceProj/functions/NormalCDF.java

Purpose

An abstract class to compute the cumulative distribution for a normal random variable. The code was copied from a third party and can be found at http://www1.fpl.fs.fed.us/CDF_Normal.java.

Dependencies

None

Class Variables

None

Constructors

None.

Functions

`static double normp(double z)` - Computes the normal CDF, F(z).

AssetPriceProj/functions/PricingFunction.java

Purpose

An abstract parent class for all price forecasting rules. Subclasses will implement a specific functional form, such as polynomials.

Dependencies

Extends: `GeneralFunction`

Class Variables

`coeff` - `double[]` - holds pricing function parameters (coefficients of a polynomial, for example)

Constructors

`PricingFunction()` - Default constructor, does nothing.

`PricingFunction(double[] coefficients)` - Accepts coefficients and stores them in `coeff`.

Functions

`abstract double evaluate(double x)` - An abstract method that must be overridden by the child class. Defines how to compute a price given a dividend.

`abstract void display()` - Defines how to display the function. Must be overridden by a child class.

`abstract PricingFunction convertToPoly(int degree)` - Uses a binomial expansion to expand the pricing function and represent it as a polynomial of the specified degree. If the pricing function is a polynomial, then it is returned unchanged.

`void setCoeff(double[] a), double[] getCoeff()` - Accessor functions.

AssetPriceProj/functions/PricingPolynomial.java

Purpose

A polynomial class to define a specific pricing function.

Dependencies

Extends: `PricingFunction`

Requires: `Vectors`

Class Variables

`degree` - `int` - the degree of the polynomial
Inherits `coeff` from `PricingFunction`

Constructors

`PricingPolynomial()` - Default constructor. Sets `degree` to 0 and `coeff` to 0.

`PricingPolynomail(double[] x)` - Sets `coeff` to `x`.

Functions

`double evaluate(double x)` - Evaluates the polynomial at `x`. Overrides the function in the parent class.

`void scalarMult(double z)` - Mutilplies each coefficient by `z` and replaces the polynomial with the new one.

`void display()` - Displays the polynomial to the screen. Overrides the function `display()` in the parent class.

`void setCoeff(double[] a), int getDegree()` - Accessor functions.

AssetPriceProj/functions/PricingPowerFunction.java

Purpose

A subclass to implement pricing functions of the form $f(x) = a(x+b)^g$, for constants a, b, g and where g is any positive real number.

Dependencies

Extends: `PricingFunction`

Requires: `PricingPolynomial`

Class Variables

Inherits `coeff` from `PricingFunction`.

Constructors

`PricingPowerFunction()` - Default constructor, sets `a=0, b=0, g=1`.

`PricingPowerFunction(double[] c)` - Sets `a=c[0], b=c[1], g=c[2]`.

Functions

`double evaluate(double x)` - Evaluates the function at `x`.

`void display()` - Displays the function to the screen.

`PricingFunction convertToPoly(int degree)` - Uses a binomial expansion to expand the `PricingPowerFunction` to the specified degree.

`double getA(), void setA(double a), double getB(), void setB(double b), double getGamma(), void setGamma(double g)` - Accessor functions.

AssetPriceProj/functions/Regeression.java

Purpose

Implements a learning algorithm where at each time step the agent performs a regression on the total history of the market data. The pricing rule must depend on the parameters in a linear way (the coefficients of a polynomial, for example). There is an option to reset the history so the agent only looks back so far.

Dependencies

Extends: `LearningRule`

Requires: `Matrix`

Class Variables

Inherits vectors `d` and `p` and integers `N` and `T` from `LearningRule`.

Constructors

`Regression(double[] d, double[] p, int N)` - User specifies the data.

Functions

`double[] update()` - Updates the pricing parameters by performing a complete linear regression on the history. If data has been reset, then only data from that point on is used. Returns the updated coefficient vector.

AssetPriceProj/functions/SchumakerSpline.java

Purpose

Implements a shape-preserving, quadratic Schumaker spline. Details on the spline can be found in Judd (1998), pp. 231-234. The notation used in the class was chosen to match Judd.

Dependencies

None

Class Variables

`t,z - double[]` - independent and dependent data, respectively
`T - int` - length of the data
`L,s,d - double[]` - used for approximating derivatives
`A1,B1,C1,A2,B2,C2 - double[]` - coefficients
`xi - double[]` - nodes

Constructors

`SchumakerSpline(double[] t, double[] z)` - User supplies the data. `t` is the independent variable and `z` ia the dependent.

Functions

`double evaluate(double x)` - Returns the interpolated function, evauated at `x`. If `x` is outside of the upper and lower limits of the original `t` vector, an exception is thrown.

`void computeDerivatives()` - Populates the vectors `L, s` and `d`.

`void computeXi()` - Computes the nodes for the interpolation.

`void computeCoefficients()` - Computes the coefficients for the interpolating function.

Exceptions

IllegalArgumentException - Thrown if the user asks to evaluate the spline outside of the original vector `t`, within a tolerance. That is, if `x` $<$ `t[0]-tol` or `x` $>$ t[T-1]+tol. We add the tolerance to avoid problems from round off.

AssetPriceProj/functions/SplineMatrix.java

Purpose

A class to hold a two dimensional array of `SchmakerSpline` objects.

Dependencies

Requires: `SchumakerSpline, ThreeDArray`

Class Variables

`S - SchumakerSpline[][]` - the two dimensional array of splines, one for each dividend and price pair
`v - ThreeDArray` - the agent's value function
`n2,n3 - int` - the dimensions of `v[i][][]`
`s - double[]` - holdings grid

Construstors

`SplineMatrix(BCKAgent a)` - User specifies an agent and the constructor extracts `v`, `s`, `n2` and `n3` from it.

Functions

`void setUpSplines()` - For each `j,k`, contructs a schmaker spline with data `s` and `v[][j][k]`.

`SchumakerSpline getSpline(int j, int k)` - Returns `S[j][k]`.

AssetPriceProj/functions/Tauchen.java

Purpose

This class implements a method to generate nodes and probabilities for converting a continuous AR(1) process into a discrete process. Specifically, the process is of the form $Z' = \rho Z + e'$, where e' is normally distributed with mean 0. For details see Heer and Maussner (2005), pp. 497-499.

Dependencies

Requires: `NormalCDF`

Class Variables

`rho` - `double` - the AR(1) coefficient
`sigma` - `double` - the standard deviation of the random shocks
`lambda` - `double` - the number of standard deviations to include in the discrete approximation
`m` - `int` - the number of nodes to generate
`nodes` - `double[]` - the nodes
`probs` - `double[][]` - the transition probability matrix

Constructors

`Tauchen(double rho, double sigma, double lambda, int m)` - User supplies all variables

Functions

`void makeNodes()` - Generates the `m` nodes and populates the `nodes` array.

`void computeProbs()` - Computes the probability matrix and populates the `probs` array.

`double[] getNodes(), double[][] getProbs()` - Accessor functions.

AssetPriceProj/functions/ThreeDArray.java

Purpose

This class provides a way for us to store and manipulate three dimensional arrays. Java naturally allows us to store objects such as these using `double[][][]`, however we are unable to manipulate them.

Dependencies

None

Class variables

`n1` - `int` - size of the grid in the first direction
`n2` - `int` - size of the grid in the second direction
`n3` - `int` - size of the grid in the third direction
`A` - `double[][][]` - stores the three dimensional data

Constructors

`ThreeDArray(int n1, int n2, int n3)` - Initializes a `ThreeDArray` with all entries equal to 0.

`ThreeDArray(int n1, int n2, int n3, double c)` - Initializes a `ThreeDArray` with all entries equal to c.

Functions

`void setup(int n1, int n2, int n3, double c)` - Input the sizes of the array and the initial value for the elements. Allocates the memory and creates a three dimensional `double` to store the data. Used only by the constructors.

`void put(int i, int j, int k, double a)` - Sets the (i, j, k) entry of the array to a.

`double get(int i, int j, int k)` - Returns the (i, j, k) entry.

`static void equals(ThreeDArray A, ThreeDArray B)` - A static function that replaces the elements of the `ThreeDArray A` with the elements of `ThreeDArray B`.
Throws an `IllegalArgumentException` if the arrays do not have identical dimensions.

`static ThreeDArray minus(ThreeDArray A, ThreeDArray B)` - A static function that subtracts two `ThreeDArrays` pointwise. Returns a new `ThreeDArray` with the elements of `A-B`. Throws an `IllegalArgumentException` if the arrays do not have identical dimensions.

`double absMax()` - Returns the maximum absolute value of the elements of the `ThreeDArray` which it is invoked upon.

`int[] getDims()` - Returns a vector of integers representing the dimensions of the array. The order is `(n1,n2,n3)`.

`void display()` - Displays the output to the screen. The output is formatted as in Matlab, where the third dimension is fixed at each of its values and the corresponding matrices are printed.

`void display(String fileName)` - Same as previous, except it writes the output ot the file specified in `fileName`. Note that `fileName` must be entered as a string, using double quotes.

`void round(int m)` - Rounds all the elements in the array to `m` decimals. Note: The original data will be replaced with the rounded values.

`double[][] fixDiv(int j)` - Fixes the second index at `j` and returns the two dimensional `double A[][j][]`.

`double[] fixIJ(int i, int j)` - Fixes the first two dimensions at `i` and `j`. Returns the resulting one dimensional vector `A[i][j][]`.

`double[] fixJK(int j, int k)` - Fixes the last two dimensions at `j` and `k`. Returns the resulting one dimensional vector `A[][j][k]`.

Exceptions

`IllegalArgumentException` - there is a problem with the dimensions of the arrays.

AssetPriceProj/functions/ThreeDIntArray.java

This class is identical to `ThreeDArray`, except it stores integers instead of doubles. All the functions are the same, with the obvious difference being that `double` is replaced by `int`.

AssetPriceProj/functions/TriDiagonalSolver.java

Purpose

Solves a tri-diagonal linear system of n equations. The algorithm was taken from Kincaid and Cheney (2002), p. 179.

Dependencies

None

Class Variables

`D` - `double[]` - a vector of length n with the diagonal elements
`subD` - `double[]` - a vector of length n-1 with the sub-diagonal elements
`superD` - `double[]` - a vector of length n-1 with the super-diagonal elements
`rhs` - `double[]` - a vector of length n with the ride hand side of the system

Constructors

`TriDiangonalSolver(double[] subD, double[] D, double[] superD, double[] rhs)` - user supplies all variables.

Functions

`double[] solve()` - Solves the tri-diagonal system using a backwards substitution. Returns the solution as a vector.

AssetPriceProj/functions/UpdateParams.java

Purpose

An abstract class containing a single static function for updating the agent's pricing function. Currently only functional for `PricingPolynomial` objects.

Dependencies

Requires: `PricingFunction`, `PricingPolynomial`

Class Variables

None

Constructors

None

Functions

`static PricingFunction pricing(PricingFunction currentPriceFun, double[] divs, double[] prices, double lambda, LearngingRule L)` - User inputs the agent's current function, the history and a learning rule. The parameter `lambda` is used to control the speed at which agents learn. `lambda` is between 0 and 1 and weights the old forecast and the new one. A `lambda` of 1 means the agent completely discards his old parameters and a `lambda` of 1 means they never update.

AssetPriceProj/functions/Vectors.java

Purpose

An abstract class that contains many very useful routines to manipulate arrays of doubles or integers. The class is abstract, so there is no such thing as a `Vector` object. Instead, we just pass arrays of doubles in and out and treat them as vectors.

Dependencies

Requires: `PrintWriter, FileWriter`

Class Variables

None

Constructors

None

Functions

`static double sum(double[] v)` - Computes the sum of the elements of the vector.

`static int[] greaterThan(double[] x, double a)` - Returns a vector of integers containing the index of each element of `x` that is strictly bigger than `a`. If there aren't any, it returns -1.

`static int[] locate(double[] x, double y)` - Returns a vector of two integers, `L` and `U`, such that `y` lies between `x[L]` and `x[U]`. Throws an exception if `y` is not within the upper and lower limits of `x`.

`static void display(double[] x)` - Displays the vector `x` to the console.

`static void display(int[] x)` - Same, but for integers.

`static double[] round(double[] x, int m)` - Returns the vector `x` with each element rounded to `m` decimal places.

`static int find(double[] x, double y)` - Returns `i` such that `y = x[i]`. If `y` is not in `x`, then -1 is returned.

`static double dot(double[] x, double[] y)` - Returns the dot product of vectors `x` and `y`. If the vectors do not have the same size, an exception is thrown.

`static double supNorm(double[] x)` - Returns the maximum element of `x`, in absolute value.

`static double[] scalarMult(double[] x, double a)` - Returns the vector with each element multiplied by `a`.

`static double concatenate(double[] x, double[] y)` - Returns a vector of length `x.length + y.length`. Concatenation is in the same order as the vectors are entered.

`static void writeToFile(double[] x, String s, String fileName)` - Writes the vector `x` to the file `fileName`. The variable `s` can be either `Row` or `Column`, specifying how to write the vector. An exception is thrown if the file cannot be accessed.

`static void writeToFile(int[] x, String s, String fileName)` - Same, but for integers.

AssetPriceProj/Jama

This package contains many useful classes for performing matrix operations. This package is one of the only parts of our code that was not written from scratch by us. It was written by a third party and thus no documentation is provided here. Complete documentation can be found at http://math.nist.gov/javanumerics/jama.

REFERENCES

Araujo, A. (1991). The once but not twice differentiability of the policy function. *Econometrica*, 59:1383–1393.

Arthur, W. B., Holland, J. H., LeBaron, B., Palmer, R., and Tyler, P. (1997). Asset pricing under endogenous expectations in an artificial stock market. In Arthur, W. B., Durlauf, S. N., and Lane, D. A., editors, *The Economy As an Evolving Complex System II*, pages 15–44. Addison-Wesley.

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3:133–181.

Beaumont, P. M., Culham, A. J., and Kercheval, A. N. (2007). Asset price dynamics in a heterogeneous agent equilibrium model. Working paper, Florida State University.

Beneveniste, L. M. and Scheinkman, J. A. (1979). On the differentiability of the value function in dynamic models of economics. *Econometrica*, 47:727–732.

Billingsley, P. (1995). *Probability and Measure*. John Wiley & Sons, third edition.

Campbell, J. Y. and Cochrane, J. H. (1999). Explaining the poor performance of consumption-based asset pricing models. NBER Working Paper N0. W7237.

Chiarella, C., Dieci, R., and Gardini, L. (1998). Asset price and wealth dynamics in a financial market with heterogeneous agents. *Journal of Economic Dynamics & Control*, 30:1755–1786.

Ehrentreich, N. (2004). A corrected version of the Santa Fe Institute artificial stock market model. Working paper. Available at http://www.aeaweb.org/annual_mtg_papers/2005/0108_0800_0817.pdf.

Evans, G. W. and Honkapohja, S. (2001). *Learning and Expectations in Macroeconomics*. Princeton University Press.

Frydman, R. and Phelps, E. S. (1983). *Individual forecasting and aggregate outcomes: Rational expections examined*. Cambridge University Press.

Harris, M. (1987). *Dynamic Economic Analysis*. Oxford University Press.

Heer, B. and Maussner, A. (2005). *Dynamic General Equilibrium Modelling*. Springer.

Honkapohja, S. and Mitra, K. (2002). Adaptive learning in stochastic nonlinear models: A new stability result. Working paper.

Judd, K. L. (1998). *Numerical Methods in Economics*. The MIT Press.

Kincaid, D. and Cheney, W. (2002). *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole Publishing Company, third edition.

LeBaron, B. (2002). Bulding the Santa Fe artificial stock market. Working Paper. Available at http://people.brandeis.edu/~blebaron/wps/sfisum.pdf.

LeBaron, B. (2004). Agent-based financial markets: Matching stylized facts with style. Working paper, Bradeis University.

LeBaron, B. (2006). Agent-based computational finance. In Tesfatsion, L. and Judd, K. L., editors, *Handbook of Computational Economics*, volume 2 of *Handbook of Computational Economics*, chapter 24, pages 1187–1233. Elsevier.

LeBaron, B., Arthur, W. B., and Palmer, R. (1999). Time series of an artificial stock market. *Journal of Economic Dynamics and Control*, 23:1487–1516.

Leroy, S. F. and Werner, J. (2001). *Principles of Financial Economics*. Cambridge University Press.

Li, T. (2007). Heterogeneous beliefs, asset prices, and volatility in a pure exchange economy. *Journal of Economic Dynamics & Control*, 31:1697–1727.

Ljungqvist, L. and Sargent, T. J. (2000). *Recursive Macroeconomic Theory*. The MIT Press.

Lucas, Jr., R. E. (1978). Asset prices in an exhange economy. *Econometrica*, 46(6):1429–1445.

Marcet, A. and Sargent, T. J. (1989). Convergence of least squares learning mechanisms in self-referential linear stochastic models. *Journal of Economic Theory*, 48:337–368.

Marimon, R., McGrattan, E., and Sargent, T. J. (1990). Money as a medium of exchange in an economy with artificially intelligent agents. *Journal of Economic Dynamics & Control*, 14(2):329–373.

Mehra, R. (2003). The equity premium: Why is it a puzzle? NBER Working Paper N0. W9512.

Mehra, R. and Prescott, E. C. (1985). The equity premium: A puzzle. *Journal of Monetary Economics*, 15:145–161.

Sargent, T. J. (1993). *Bounded Rationality in Macroeconomics*. Oxford University Press.

Schumaker, L. L. (1983). On shape preserving quadratric spline interpolation. *SIAM Journal on Numerical Analysis*, 20:854–864.

Simon, H. (1957). *Models of Man: Social and Rational*. John Wiley & Sons, New York.

Stokey, N. L. and Lucas, Jr., R. E. (1989). *Recursive Methods in Economic Dynamics*. Harvard University Press.

Tauchen, G. (1986). Finite state Markov-chain approximations to univariate and vector autoregressions. *Economics Letters*, 20:177–181.

Taylor, J. B. and Uhlig, H. (1990). Solving nonlinear stochastic growth models: A comparison of alternative solution methods. *Journal of Business & Economic Statistics*, 8(1):1–17.

Tesfatsion, L. (2003). Agent-based computational economics. ISU Economics Working paper No. 1.

BIOGRAPHICAL SKETCH

Andrew James Culham

Andrew James Culham, son of Gail Jiggins and James Culham, was born on the thirteenth day of June, 1979, in Cambridge, Ontario, Canada. After graduating from John G. Diefenbaker High School in Calgary, Alberta, Canada, Andrew enrolled at the University of Calgary in the fall of 1997. There he spent five years obtaining a double major in Applied Mathematics and Pure Mathematics with a minor in Economics. In the fall of 2002, Andrew accepted an assistantship with the Department of Mathematics at Florida State University for graduate study in Financial Mathematics. In April, 2004, Andrew was awarded a Master of Science degree and decided to continue his studies by pursuing a doctorate in Financial Mathematics under the joint direction of Dr. Alec Kercheval and Dr. Paul Beaumont. Andrew received his Ph.D. in the summer of 2007 for his work in non equilibrium asset pricing.

Outside of his academic accomplishments, Andrew has also twice been honored for excellence in teaching. In 2004, the Department of Mathematics awarded Andrew the Dwight B. Goodner award for outstanding teaching by a graduate student. Following that, in 2007, Andrew was awarded an Outstanding Teaching Assistant Award by the University. This award is one of fifteen given out campus wide and is thus a great honor to receive.

Andrew currently lives in Tallahassee, Florida, where he has lived since he enrolled in graduate school. Upon completion of his studies, he wishes to pursue a non academic career in the banking or energy sector, possibly in Canada or in sunny Florida. In his spare time, Andrew enjoys spending time with family and friends and watching the Toronto Blue Jays and Florida State Seminoles on television.