BANKING AND ASSET BUBBLES: TESTING THE THEORY OF FREE BANKING USING AGENT-BASED COMPUTER SIMULATIONS AND LABORATORY EXPERIMENTS

by

William A. McBride
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Economics

Committee:

[illegible] Director

[illegible]

[illegible]

[illegible] Department Chairperson

Richard E. Wagner Program Director

[illegible] Dean, College of Humanities and Social Sciences

Date: July 29, 2010

Summer Semester 2010
George Mason University
Fairfax, VA

Banking and Asset Bubbles: Testing The Theory Of Free Banking Using Agent-Based Computer Simulations And Laboratory Experiments

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

William Howard Alford McBride
Doctor of Philosophy
George Mason University, 2010

Director: Daniel Houser, Professor
Economics

Summer Semester 2010
George Mason University
Fairfax, VA

Copyright: 2010, William Howard Alford McBride
All Rights Reserved

DEDICATION

This is dedicated to Robyn Holley.

ACKNOWLEDGEMENTS

I would like to thank my committee members, Professors Daniel Houser, Lawrence H. White, and Robert Axtell, as well as Professor George Selgin.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

BANKING AND ASSET BUBBLES: TESTING THE THEORY OF FREE BANKING USING AGENT-BASED COMPUTER SIMULATIONS AND LABORATORY EXPERIMENTS

William Howard Alford McBride, PhD

George Mason University, 2010

Thesis/Dissertation Director: Professor Daniel Houser

Throughout much of the 18th and 19th centuries, many countries practiced free banking, i.e. competitive note issue without control by a central bank. Modern day proponents of free banking argue it is capable of limiting asset bubbles by making for a more stable relationship between the volume of base money present in an economy and corresponding levels of credit and aggregate spending. The main contribution of this dissertation is to provide a novel test of this claim by using the canonical laboratory double auction asset trading market (CLDAATM) initiated by Vernon Smith and others, which tends to generate asset price bubbles. Chapter 1 describes an agent-based model and simulation of CLDAATM using three types of trading strategies: value, momentum, and speculative. The results closely resemble the asset price bubbles generated in the laboratory. Chapter 2 augments the design of Chapter 1 by adding an agent-based model and simulation of banking, with the purpose of comparing both free and central banking

to CLDAATM. The results indicate that both free and central banking reduce the tendency to bubble as compared to CLDAATM, and free banking is most effective in this regard. Chapter 3 describes a laboratory experiment design which augments CLDAATM using simulated trading and banking agents.

CHAPTER 1: AN AGENT-BASED MODEL AND SIMULATION OF ASSET TRADING IN A DOUBLE AUCTION

1.0 Introduction

In this chapter I describe an agent-based model and computer simulation of the canonical laboratory double auction asset market as constructed by Smith et al. (1988). My main objective is to replicate the following empirical regularities found in the laboratory: 1) asset prices tend to bubble above and then crash back to fundamental value, 2) bubbles are typically preceded by a large volume of trade, 3) prices tend to start out well below fundamental value, and 4) subjects learn over time so that bubbles are less likely with experienced subjects. Secondarily, my objective is to replicate the same empirical regularities at much larger scales of traders than can be accomplished in the laboratory. The purpose is to generate a large scale asset trading market which can be used in conjunction with an agent-based simulation of banking, which will be developed in Chapter 2.

As noted by many agent-based computational economic (ACE) researchers, including Duffy (2006), Gode and Sunder (1993), and Chan et al. (1993), agent-based modeling has an advantage over human subject experiments in that the researcher can precisely control the motivations, preferences, and information-processing capabilities of agents. The cost, as with any model of human behavior, is that virtual agents can never be as complex or as diverse as the real thing. Indeed, as Axelrod (1997) calls it, the "keep-it-simple-stupid" (KISS) principle is the main principle ACE researchers use in modeling agent behavior. Furthermore, one general benefit of using agent-based computer simulations in conjunction with experiments is it provides a low cost test bed for experiment designs.

While all of these issues provided some motivation to use agent-based modeling and simulation, the main motivating factor was the issue of scale. Particularly, in order to fully test the banking hypothesis of Chapter 2, I need large numbers of asset trading bank customers, i.e. larger numbers than can be had in the laboratory. The reason is that, as Selgin (1988, pp. 73-78) illustrates, a large number of transactions provides bankers with an approximately continuous distribution of clearings. This will be made clearer in Chapter 2 where the banking simulation is discussed. For now, the objective is to first simulate the laboratory asset market conditions at the original scale, i.e. about 12 traders, and then scale this up to 100 and 225 traders.

The organization of the chapter is as follows. Section 2.0 reviews the literature of agent-based modeling as it pertains to asset trading markets. Section 3.0 develops the agent-based model used here and in Chapter 2, which is based on the model used by De Long et al. (1990a), where there are three types of trading strategies: momentum, value, and speculative. Section 3.1 discusses the parameters used in the model. Section 4.0 discusses the results and section 5.0 concludes.

Mainly, I find that the agent-based model and simulation developed herein successfully generates the important empirical regularities found in the laboratory. Further, it does so at scales of 12, 100, and 225 traders. I find that the most profitable trading strategy is value trading. Also, I focus on the monetary environment of the original laboratory experiments of Smith et al. (1988) and find that the money supply typically more than doubles over the course of each experiment as a result of dividend payments.

2.0 Literature Review

LeBaron (2001) says there are generally three ways to model traders in agent-based computational finance: 1) based on real world trading strategies, 2) minimal cognitive capacity of traders, as in Gode and Sunder's (1993) zero intelligence traders, and 3) bottom up evolution of strategies using genetic algorithms.

LeBaron (2000) discusses a number of bottom-up evolution type models which typically use genetic algorithms to evolve trading strategies. Rust et al (1992) report on the Santa Fe Institute Double Auction Tournament and find that the trading strategy that performs best is one that simply feeds off the others by waiting for the bid-ask spread to narrow before "stealing the deal". Being that my objective is to simulate as closely as possible the laboratory asset market of Smith et al (1988), genetic algorithms are not an appropriate learning mechanism. That is, laboratory experiments typically last less than two hours, during which time there is typically only one asset bubble. Thus, there are no generational selection effects, and within each session there is little time for trial and error of trading strategies. Rather, a typical subject most likely spends his time learning the trading environment and the asset dividend structure, while forming expectations about how others learn and from this he settles on a single trading strategy.

Duffy and Ünver (2006) present a very pertinent example of LeBaron's second variety of trader models, i.e. those which aim for minimal cognitive capacity of traders. In particular, they attempt to replicate the asset bubble results of Smith et al. (1988) using "near zero-intelligence" agents. Essentially, these agents form price expectations from two components: a random component, as in Gode and Sunder (1993), and a herding component based on the last price. Mainly, while I find their results impressive in their simplicity, it seems doubtful that "near zero-intelligence" agents bare much resemblance to real world traders in exchange environments either inside or outside the laboratory.

One pertinent example of LeBaron's first variety of trader models, i.e. those based on real world strategies, is by Chan et al (1999), who also use agent-based models to simulate the conditions of laboratory double auction trading markets. They are mainly interested in information aggregation and the ability to achieve the rational expectations equilibrium within the double auction institution, whereas my focus is more limited in that I am trying to replicate the experimental conditions of Smith et al (1988), where all traders have access to the same information at the same time. This results in a fundamentally different market structure. For example, Chan et al use a longer trading horizon than Smith et al (1988): 75 consecutive trading periods versus 15. Secondly, they reset endowments at the start of each period. In terms of how traders are modeled, Chan et al explore heterogeneous preferences by using agent-dependent payoffs, as well as heterogeneous information by giving some fraction of traders early access to signals of fundamental value. They model three types of trading strategies: 1) “empirical Bayesian”, which is a fairly complicated value trading strategy, 2) momentum, which not quite linearly extrapolates from the last two prices, and 3) “nearest-neighbor”, which is similar to empirical Bayesian but uses a longer moving average of asset prices.

Like Chan et al, I use LeBaron’s first approach to modeling traders, i.e. start with real world strategies. From there, I attempt to generate, mainly through simple trial and error, the empirical phenomena of Smith’s laboratory asset bubbles. This is the so-called “generative approach” commonly used among ACE researchers (Epstein and Axtell, 1996).

While rational expectations would seem to preclude what Fisher Black (1988) called “noise” trading, i.e. trading on relatively little information, De Long et al (1990b, 1991) argue it persists primarily for two reasons: 1) the so-called P.T. Barnum rule, in this case meaning new, inexperienced traders enter the market on a regular basis such that at any one time they make up a significant share, and 2) noise trading is risky and thus yields more profit. Value traders expose themselves to risk in betting against noise traders, and thus have an incentive to hedge their bets. De Long et al (1990a) take this reasoning one step further by addressing the existence of positive feedback trading strategies, i.e. buying when prices are going up and selling when prices are going down. They argue that it can result from either 1) extrapolative expectations about prices, i.e. trend chasing, 2) stop-loss orders, which issue sell orders after some specified price decline, or 3) liquidation of positions in order to meet margin calls. Furthermore, traders who expect others to trade this way will see the opportunity to profit from it, by trading ahead of them, thus further destabilizing prices. This is the speculative strategy. De Long et al point to evidence of such behavior both inside and outside the laboratory, including the trading experiments of Smith et al (1988). Most importantly for the present study, they point to the trading experiments of Andreas and Krauss (1988), who find that trend chasing increases as the trend becomes more evident, i.e. it depends on a significant number of observations not just the most recent price changes.

Alferano, Lux and Wagner (2005, pp. 26 - 28) use an “asymmetric herding model” in which there are two types of trading strategies, fundamental and noise trading, and agents’ propensity to switch strategies and herd are parameters. As in De Long et al. (1990b), the excess demand of noise traders is driven by random fluctuations in their mood, i.e. their optimism or pessimism. To be exact, whereas De Long et al model mood as an I.I.D. random variable, Alferano et al. model it as a random walk, which has the effect of reducing price volatility. In either case, I object to treating this sort of exogenous fluctuation in mood as a major factor for three reasons: a) it is not plausible in the experimental laboratory conditions created by Vernon Smith and others, particularly since these experiments typically last less than 2 hours, and b) this in fact casts doubt on the general validity of such claims, i.e. “animal spirits”, inside or outside the lab, and c) it defines away the problem of what causes such fluctuations in mood.

3.0 Agent-based Model

Thus, for the asset market, similar to De Long et al. (1990a), I model three types of trading strategies: momentum, value, and speculative. Momentum trading is rational in any sufficiently complex environment where learning is involved. That is, due to dispersed information and/or heterogeneous cognitive abilities and/or limited attention, it is rational to rely on the wisdom of others. This is almost certainly the case in Vernon Smith's laboratory experiments. In particular, momentum traders form price expectations based on a linear trend of the last two prices. Thus, willingness to pay at time *t*, $M{:}WTP_t$, is as follows:

$$M{:}WTP_{t+1} = p_t + (p_t - p_{t-1})$$

where p_t is the asset price at time *t*. Value traders form price expectations based on the fundamental value, i.e. the expected dividend value, $E(D_t)$, of the asset. Thus, willingness to pay at time *t*, $V{:}WTP_t$, is as follows:

$$V{:}WTP_{t+1} = E(D_t)$$

Speculative traders expect momentum traders to amplify price movements. Thus, if the current price is below fundamental value, speculative traders expect prices to increase steadily and eventually overshoot fundamental value, then collapse back to fundamental value. Even if the current price is at fundamental value, speculative traders can benefit by destabilizing prices, e.g. by bidding up the price in this period and then selling in the next period. This is somewhat different than the model of “informed rational speculators” used by De Long et al (1990a), where speculators receive news about the asset before other traders. Again, I am trying to model the conditions in the laboratory, where all traders have access to the same information at the same time. Furthermore, while De Long et al are able to generate a bubble using their framework, they are unable to generate an endogenous crash.

In particular, speculators start out bullish, expecting prices to increase at some constant rate, *SBR*. Since real-world bullish speculators seldom agree on how high prices are going, I make it a heterogeneous agent specific belief by drawing for each speculator from a uniform distribution, such that:

$$SBR = \mathrm{U}[0, MaxSBR]$$

Where *MaxSBR* is a parameter representing the maximum expected bullish return. Speculators switch from a bullish strategy to a value strategy on a price correction of *SC*, which is also seldom agreed upon in real world trading and so is similarly drawn from a uniform distribution such that:

$$SC = \mathrm{U}[0, MaxSC]$$

Where *MaxSC* is a parameter representing the maximum speculator correction. Also, speculators use a moving price average, $P(AL)_t$, to judge a price correction, where *AL* is a parameter representing the length of the price average. This is also common in real-world stock exchanges, where "technical" traders typically focus on the 50 and 200 day moving averages. Finally, in the last period, all remaining bulls switch to value trading. By this time, most bulls should have already switched. Thus, speculator willingness to pay at time *t*, $S{:}WTP_t$, is as follows:

$$S{:}WTP_{t+1} = p_t + (p_t * SBR), \qquad \text{while } p_t > (P(AL)_t * (1 - SC)) \text{ and } s < TP$$

$$S{:}WTP_{t+1} = p_t + [(E(D_t) - p_t)], \qquad \text{otherwise.}$$

Where *s* is the trading period and *TP* is total number of periods per experiment. Additionally, I apply a quadratic learning model to each trading strategy, as follows:

$$C_t = LF + LF * (s\text{-}1)^2$$

Where C_t is strategy confidence at time t, ranging from 0 to 1, and *LF* is a learning factor. I make learning ability a heterogeneous quality by drawing for each trader from a uniform distribution, such that:

$$LF = \mathrm{U}[0, MaxLF]$$

Where *MaxLF* is a parameter representing the maximum learning factor. Arguably, the value trading strategy is most time intensive, since it requires calculation of expected dividend value remaining in each period. Thus, I apply to the value strategy an additional learning factor, *VLF*, a parameter ranging from 0 to 1, to yield value strategy confidence, VC_t, as follows:

$$VC_t = VLF * C_t = VLF * (LF + LF * (s\text{-}1)^2)$$

Strategy confidence applies to each strategy, and being typically less than one, it gives inertia to prices. This can be seen in the following learning weighted strategies. For momentum trading:

$$M{:}WTP_{t+1} = p_t + (p_t - p_{t-1}) * C_t$$

For value trading:

$$V{:}WTP_{t+1} = p_t + [(E(D_t) - p_t)] * VC_t$$

And for speculative trading:

$$S{:}WTP_{t+1} = p_t + (p_t * SBR) * C_t, \qquad \text{while } p_t > (P(AL)_t * (1 - SC)) \text{ and } s < TP$$

$S{:}WTP_{t+1} = p_t + [(E(D_t) - p_t)] * C_t,$ otherwise.

Willingness to accept (*WTA*) is set equal to willingness to pay (*WTP*), i.e. zero transactions cost are assumed. All traders then use the same process to convert *WTP* and *WTA* to either a bid or offer, which is then submitted in accordance with the standard double auction institution. The particular rules here are, again, the same used in Smith et al. (1988), which in turn are taken from the trading rules used by the NYSE (Smith & Williams, 1983). Namely, the order book is closed, so that traders can only see the highest bid and lowest offer. They are restricted to progressive bid/offers, i.e. those that narrow the bid-ask spread.[1] A trader may have only one outstanding bid or offer at a time but may remove that bid/offer so long as it is not the standing best bid/offer. A trader may also displace his bid/offer by entering a better one, in which case his old bid/offer is cancelled. Likewise, if another trader displaces the standing best bid/offer by entering a better one then the old bid/offer is cancelled.[2]

A few additional assumptions were also made about the process by which *WTP* is converted to a bid or offer. First, if there are no standing bids or offers and a trader's

[1] However, there is the capability to allow for non-progressive bid/offers, which are put into a ranked bid or offer queue and advanced as contracts are made.
[2] In a previous version, I did not cancel such displaced standing bids/offers, but instead put them in the queue. This had the effect of greatly increasing the likelihood of bubbles. However, I dispensed with it upon confirmation that Smith et al (1988) did not allow this.

WTP exceeds zero, then the trader constructs a bid drawn from a uniform distribution, as follows[3]:

$$Bid = \mathrm{U}[MinBid,\ WTP], \qquad \text{where } MinBid = \mathrm{Minimum}(0.5 * p_t\ , WTP)$$

This has the effect of starting bids typically above zero and within half the current price, which appears to accord with the bid data in Smith et al (1988). In accordance with the requirement for progressive bids/offers, if there is a standing offer, then the new bid is additionally limited to below the standing offer, and if there is a standing bid, then the new bid is limited to above the standing bid. If the bid-offer spread is smaller than the difference between the trader's *WTP* and the standing offer, then the trader accepts the standing offer. This serves to increase the speed of the auction.

However, if *WTP* is below the high bid or equal to zero, then the trader constructs an offer. First, if there are no standing offers or bids then the offer is drawn from a uniform distribution, as follows:

$$Offer = \mathrm{U}[WTA,\ 3*WTA]$$

This factor of 3 is a judgment call based on the offer data in Smith et al (1988). Again, the progressive bid/offer rule applies if there is a standing bid or offer. Also, as

[3] This is similar to the design of Chan et al. (1999) who use a uniform distribution of orders over the bid-ask spread.

before, if the bid-offer spread is smaller than the difference between the standing bid and the trader's *WTA*, then the trader accepts the standing bid.

As for timing, unlike the laboratory experiments where each subject can submit a bid or offer at any time, with potentially multiple bids/offers arriving at once, for expedience I serialized the process such that only one trader at a time can form and submit a bid/offer. Each trading period contains 10 trading intervals in which each trader, in a random order, has the opportunity to submit a bid or offer.[4] There may be multiple contracts per trading interval. This is similar to the design of Chan et al (1999).

3.1 Parameters

Through a lengthy trial and error process of altering parameters, within reason, and comparing the resulting output to the empirical regularities of the experiments of Smith et al (1988), I settled on the following parameter values, shown in Table 1. First, I reasoned that among inexperienced or experienced subjects, value trading, being the behavior predicted by rational expectations, will be the most common strategy, if not the majority, while momentum trading will be the least common and speculative trading will fall somewhere in between. In fact, I settled on no more than 25% momentum traders

[4] In another previous version, I did not randomize the order of traders in each trading interval. Rather, all momentum traders went first, then all value traders, then all speculative traders. This also had the effect of greatly increasing the likelihood of bubbles. However, I dispensed with it because such a sequence is simply not realistically plausible at a scale any larger than about 10 traders.

since any more than that leads to a bubble without a crash, because value and speculative traders sell all their assets in the bubble phase leaving only momentum traders in the final periods. Beyond that, I found the results to not be very sensitive to the choice of trading strategy percentages, e.g. 40% value trading and 35% speculative trading produced similar results to 35% value trading and 40% speculative trading.

Given these constraints, I chose speculative parameters that a) fall within the typical range used in real-world speculative strategies, and b) generate bubbles. Namely, as mentioned earlier, I used this process to select the values for maximum expected bull return (*MaxSBR*), maximum correction (*MaxSC*), and the moving price average length (*AL*).

The only difference between the parameters used for 12 and 100 traders is the price correction cutoff and price average length. Essentially, the price average length in both cases is the typical volume in a period, and volume increases with 100 traders. A longer average allows for a less severe correction rule. Any more severe, or keeping the same parameters as those of the experiments with 12 traders, decreases an already lower likelihood of bubbles.

Table 1: Trading parameters.

Parameter	**Description**	**Value**
NumTraders	Total number of traders	12, 100
PercentMomentum	Percent of traders using a momentum strategy	25
PercentValue	Percent of traders using a value strategy	40
PercentSpeculative	Percent of traders using a speculative strategy	35
MaxSBR	Maximum speculative expected bull return	0.12
MaxSC	Maximum speculative correction	0.1 for 12 traders, 0.17 for 100 traders
AL	Moving price average length	10 for 12 traders, 30 for 100 traders
MaxLF	Maximum learning factor	0.2
VLF	Additional learning factor for value strategy	0.05
EndowClass1Cash	Cash endowment of Class 1 agents	$2.25
EndowClass1Assets	Asset endowment of Class 1 Agents	3
EndowClass2Cash	Cash endowment of Class 2 agents	$5.85
EndowClass2Assets	Asset endowment of Class 2 Agents	2
EndowClass3Cash	Cash endowment of Class 3 agents	$9.45
EndowClass3Assets	Asset endowment of Class 3 Agents	1
Dividend1	First of four equally likely dividend possibilities.	$0.0
Dividend2	Second of four equally likely dividend possibilities.	$0.08
Dividend3	Third of four equally likely dividend possibilities.	$0.28
Dividend4	Fourth of four equally likely dividend possibilities.	$0.60
TP	Total number of trading periods.	15
TI	Total number of trading intervals per period.	10

As for learning, I assumed that momentum and speculative traders learn fastest, achieving full confidence within the first few periods, while value traders learn slowest, taking the entire 15 periods to achieve full confidence. As mentioned earlier, my reasoning is that the value trading strategy arguably involves the most complex set of calculations. Also, the laboratory bidding data indicates that in the first few periods typically all bids are below fundamental value. Thus, the learning parameters shown in Table 1 mean that, for a sufficiently large number of traders, the average momentum or speculative trader starts in period 1 with a strategy confidence of 10% which doubles in period 2 and then grows quadratically to reach 100% in period 4 and remains at 100% throughout the experiment. The average value trader starts with a strategy confidence of 0.5% which doubles in period 2 and then grows quadratically to reach 98.5% in period 15.

I replicated Designs 4 and 5 of Smith et al (1988, Table 1, p. 1126), which use three endowment classes distinguished by the cash and asset levels shown in Table 1. Also, I used the same uniform distribution of dividends = U[\$0.0,\$0.08,\$0.28,\$0.60], which leads to an expected dividend per period, $E(D)$, of \$0.24. Lastly, I ran each experiment for 15 periods, each with 10 trading intervals.

4.0 Results and Discussion

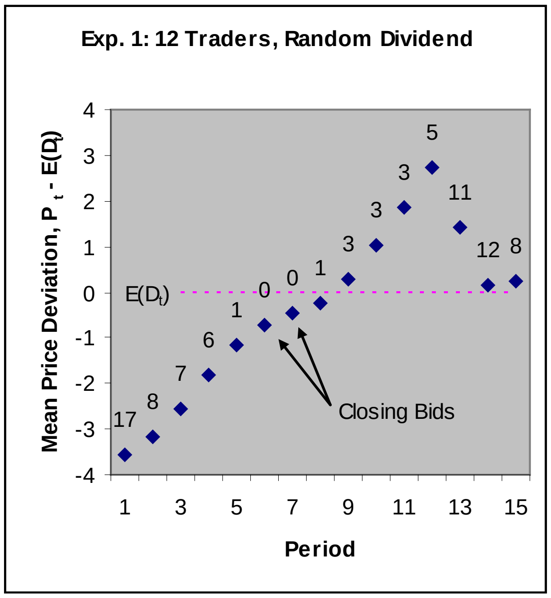
Using the parameters listed in Table 1, I ran a series of experiments at three scales: 1) a small scale of 12 traders, as in the original laboratory experiments, 2) a larger scale of 100 traders, and 3) a still larger scale of 225 traders.

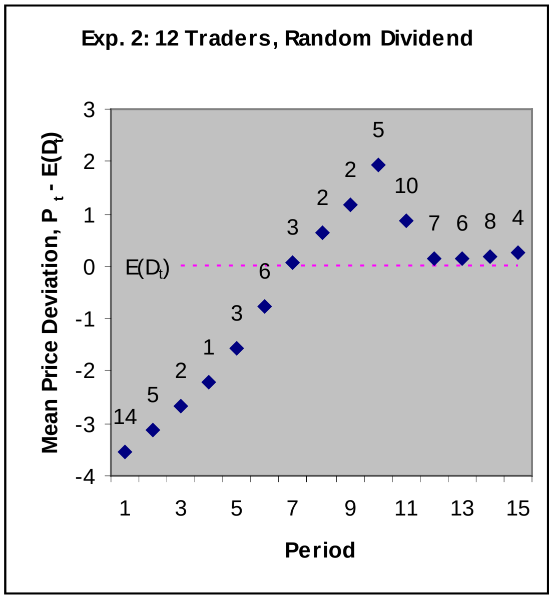
4.1 Laboratory Scale: 12 Traders

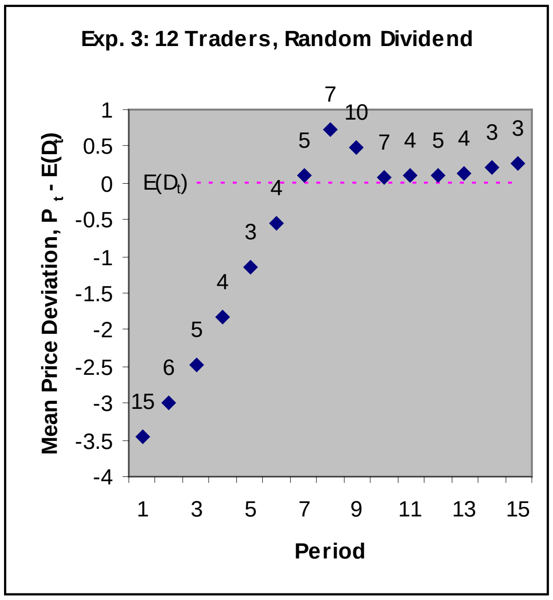
Figure 1 shows 10 experiments using a random dividend, i.e. randomly drawn from a uniform distribution = U[0,8,28,60], as in the experiments of Smith et al. (1988). Based on the graphs, these results appear to closely resemble the modal laboratory result, i.e. an asset bubble and crash where trade begins well below fundamental value, rises above and then crashes back to fundamental value by the end of the experiment. Figure 2 shows the bids, offers, and contracts for experiment 7. This also matches well the bid and offer data presented in Smith et al. (1988).

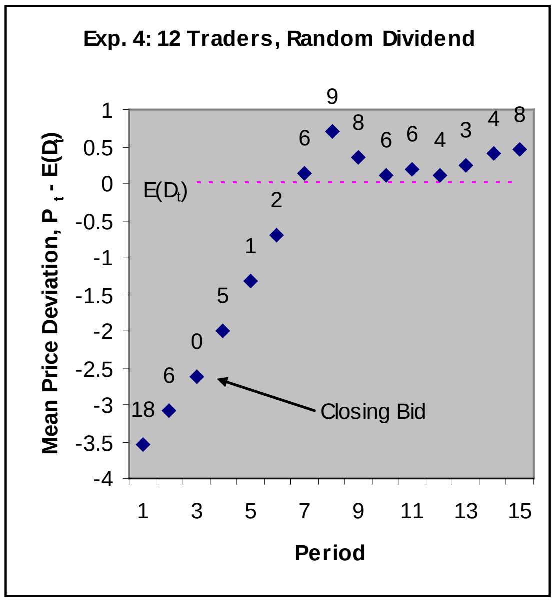
The laboratory results exhibit more variation than I have been able to produce, despite randomizing speculative strategies and learning. Particularly, there is too little variation in the first contract price, which always falls near zero. This is driven by the low learning levels I have assumed for value traders. Perhaps in future experiments I will try introducing a risk aversion model, in addition to learning, which is how Smith et al (1988, p. 1149) explain the empirical regularity of low initial prices. I did experiment with an endowment effect parameter as an additional means for generating low initial prices. However, I found that the same effect could be accomplished with a learning model, and so I opted for simplicity, i.e. a learning model without an endowment effect.

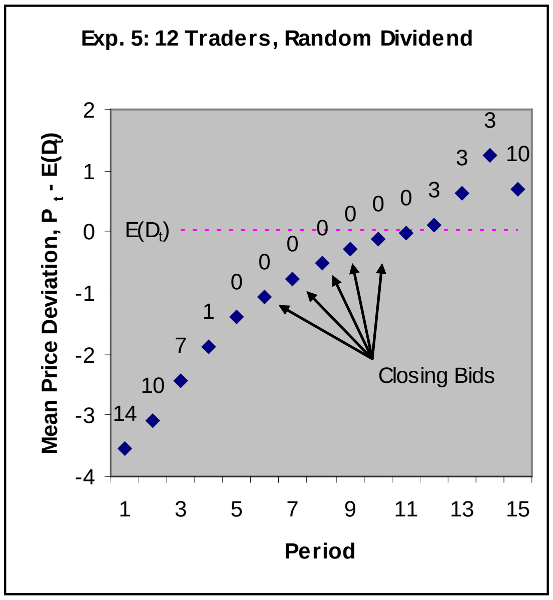
Lastly, these results do not fully match the laboratory empirical regularity of high volume in periods preceding bubbles.

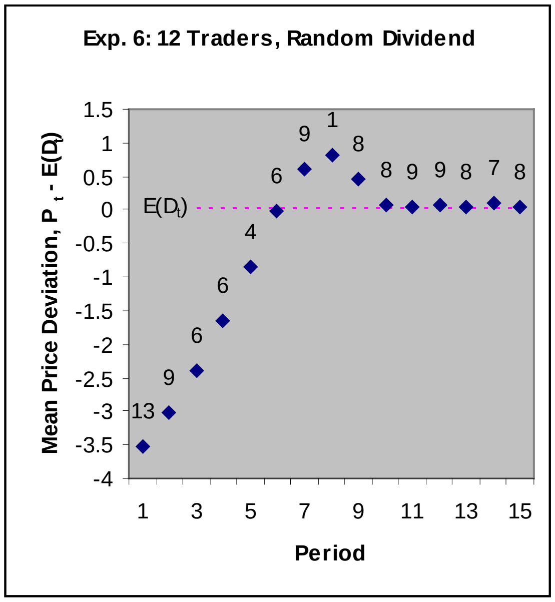





Exp. 3: 12 Traders, Random Dividend
Mean Price Deviation, $P_t - E(D_t)$
Period
$E(D_t)$
15 6 5 4 3 4 5 7 10 7 4 5 4 3 3
1 0.5 0 -0.5 -1 -1.5 -2 -2.5 -3 -3.5 -4
1 3 5 7 9 11 13 15


Exp. 4: 12 Traders, Random Dividend
Mean Price Deviation, $P_t - E(D_t)$
Period
$E(D_t)$
Closing Bid
18 6 0 5 1 2 6 9 8 6 6 4 3 4 8
1 0.5 0 -0.5 -1 -1.5 -2 -2.5 -3 -3.5 -4
1 3 5 7 9 11 13 15


Exp. 5: 12 Traders, Random Dividend
Mean Price Deviation, $P_t - E(D_t)$
Period
$E(D_t)$
Closing Bids
14 10 7 1 0 0 0 0 0 0 0 3 3 3 10
2 1 0 -1 -2 -3 -4
1 3 5 7 9 11 13 15


Exp. 6: 12 Traders, Random Dividend
Mean Price Deviation, $P_t - E(D_t)$
Period
$E(D_t)$
13 9 6 6 4 6 9 1 8 8 9 9 8 7 8
1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5 -3 -3.5 -4
1 3 5 7 9 11 13 15

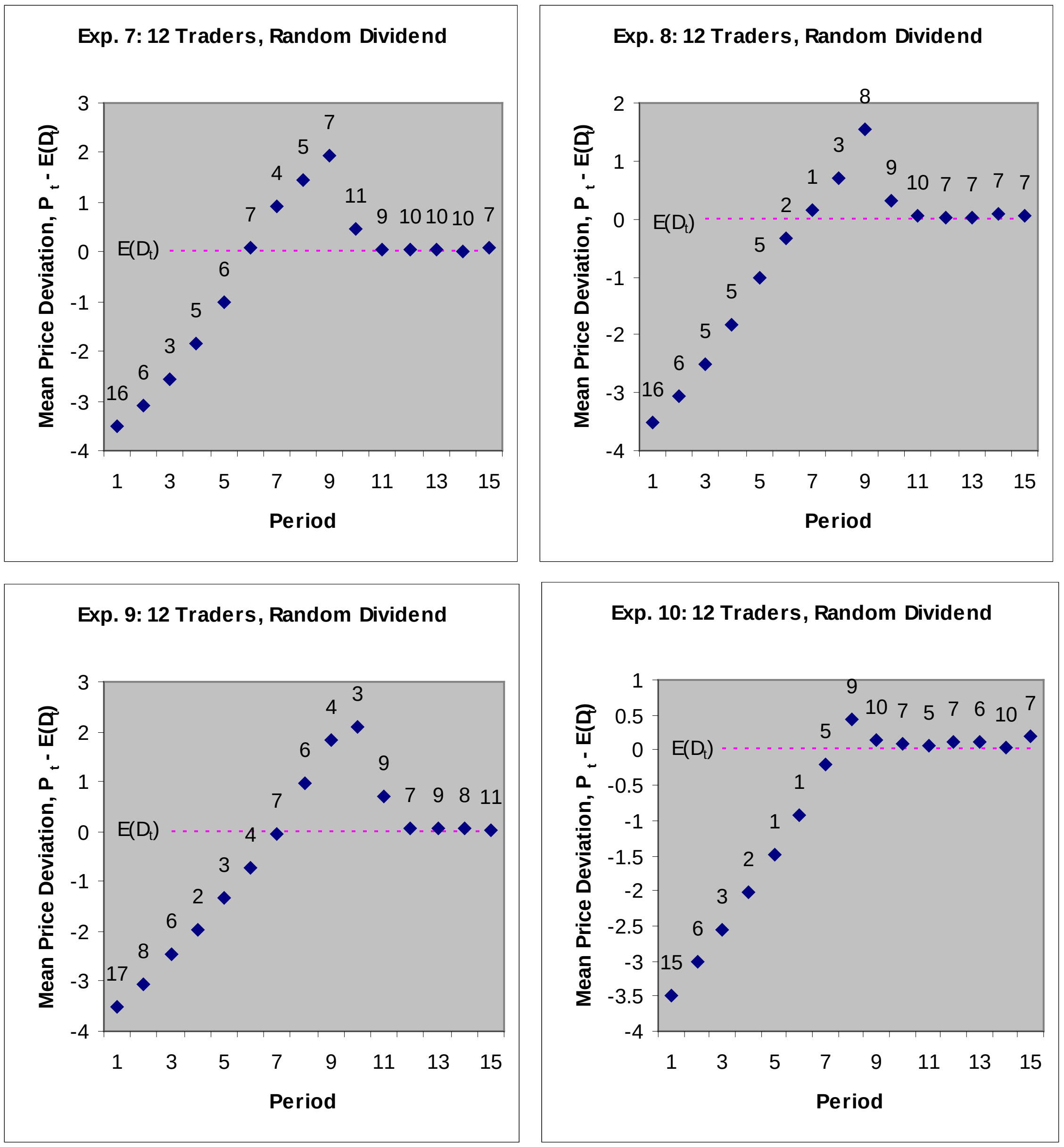


Figure 1: 10 experiments using 12 traders and a random dividend. That is the dividend is drawn randomly from four possibilities [0,8,28,60]. Volume in each period is shown above the mean price deviation.

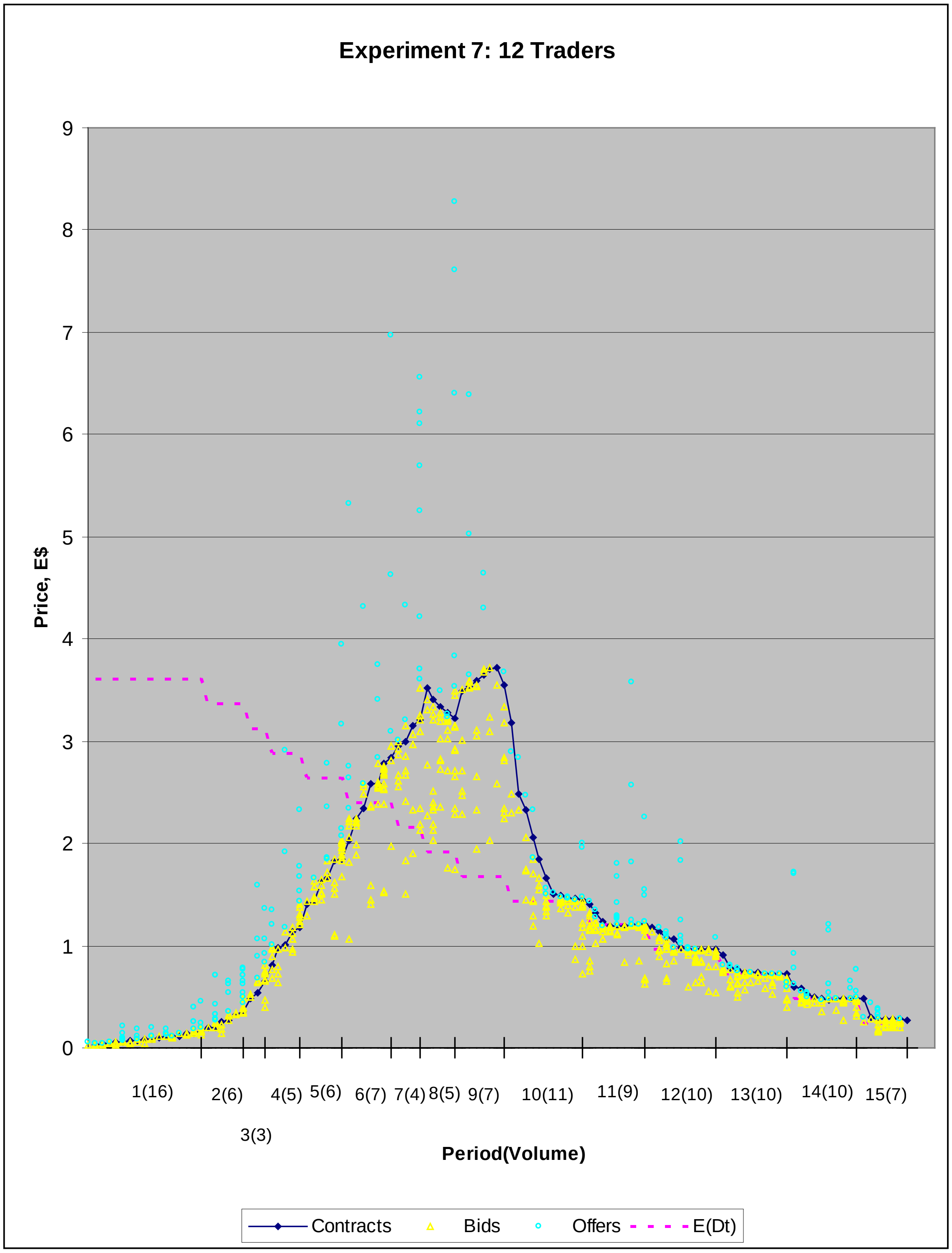


Figure 2: Experiment 7, showing all bids, offers, and contracts.

4.11 Monetary Environment

The following Figures, 3 through 12, show the money supply for experiments 1 through 10. The money supply is comprised of the cash endowment of all traders, $70.20, plus the random dividend at the end of each period, which in accordance with the range of dividend possibilities adds to the money supply one of four amounts = [$0, $1.92, $6.72, $14.40]. The expected value of the money supply at the end of each experiment is $156.60, which means that over the course of each experiment the dividend payments typically more than double the money supply. Without doubt, this additional liquidity facilitates greatly the tendency to bubble. In other experiments, which are not shown, I look at high and low dividend streaks and find that bubbles are strongly correlated with the size of dividend payments. Figures 3 through 12 illustrate this correlation by mapping the money supply along with prices, although there is a significant amount of noise and a variety of patterns relating these two time series. For instance, experiments 2, 7, and to a lesser extent 8 (Figures 4, 9 and 10), illustrate one type of relationship where dividends and the money supply remain low for the first few periods and then accelerate just before the peak of the bubble. Another type of relationship is found in experiment 9 (Figure 11), where the money supply increases tremendously in the first few periods and then, despite flattening out during the peak, a bubble ensues nonetheless. Looking at the experiments that do not exhibit bubbles, the money supplies in experiments 3 and 10 (Figures 5 and 12) increase almost linearly and at a near average rate. In experiment 4 (Figure 6), money supply remains below average until the last period of the experiment. In summary, it appears bubbles are more likely when the money supply is above average and/or accelerating.

These results point to the importance of specifying the monetary environment in such laboratory experiments. While Smith et al. (1988) vary the average dividend payout to some degree, there are simply too few samples to identify this as a source of bubble activity. Hussam, Porter, and Smith (2008) increase cash endowments and dividend uncertainty, simultaneously, and find that this treatment rekindles bubbles among experienced subjects. Further, contra earlier experiments, e.g. Smith et al. (1988), they find that even when subjects are twice experienced in this new environment a bubble tendency remains. While Hussam et al (2008) conflate increased dividend uncertainty with increased cash endowments, my results suggest that they effectively amount to the same thing, i.e. increased liquidity, and only differ in terms of timing. Indeed, in earlier laboratory experiments, Caginalp, Porter, and Smith (1998) vary cash endowments only and find more cash leads to more bubbles. Likewise, Porter and Smith (1995) vary dividend uncertainty only and find that more uncertainty leads to more bubbles.

Furthermore, Noussair, Robin, and Ruffieux (2001) attempt to isolate what they identify as two causal factors behind bubbles: frequent dividend payments and a changing fundamental value. They run a set of experiments which are similar to those in Smith et al. (1988) except they keep fundamental value constant by centering the dividend distribution around zero and at the end of each experiment paying out $3.60 per asset. They find that bubbles remain in this environment, which points to frequent dividend payments as the causal factor. Again, my results cast a different light on their results, indicating that frequent dividend payments are a proximate cause while increased liquidity through dividend payments is the ultimate cause. This is supported by the fact that the bubbles found in Noussair et al. are smaller and less frequent than those in Smith et al., and I believe this is because of the smaller per period increase in money supply in Noussair et al. resulting from their zero-mean dividend distribution. In particular, the average per period increase in money supply in Noussair et al. is of course zero versus 8.2% in Smith et al., and the maximum per period increase in money supply in Noussair et al. is 3.6% versus 20.5% in Smith et al.. Instead, Noussair et al. conjecture that their smaller bubbles result from a constant fundamental value.

Smith et al. (2000) more directly address the issue of money supply, among other explanatory hypotheses, by introducing a treatment which pays no dividends per period but rather a single final payout at the end of the experiment. They compare this to the baseline treatment of Smith et al. (1988) as well as a treatment which is essentially half way between the two, i.e. per period dividends are small but not zero and there is also a small final payout. They find clear evidence to support the "income hypothesis", i.e. that income from frequent dividend payments causes bubbles, when looking at the baseline treatment versus the zero dividend treatment. In fact, only 1 out of 10 experiments bubbles when no dividends are paid, and the one bubble occurs under conditions of high initial cash endowments. However, they reject the income hypothesis on the evidence that the baseline treatment is not "systematically" different, i.e. according to all of their statistical tests, from the "half way" treatment of small dividends and a small final payout.[5] My view is that this is a distinction based on degree not kind, and it certainly deserves further investigation.

In summary, ever since Smith et al. (1988), researchers have investigated a number of experimental treatments with the express purpose of extinguishing the bubbles found therein, and while they have found limited success, it remains the case, as noted by Smith et al. (2000, p. 568), that "controlled laboratory markets price bubbles are something of an enigma". My results indicate that focusing on money supply conditions, as was done here, might resolve much of the mystery.

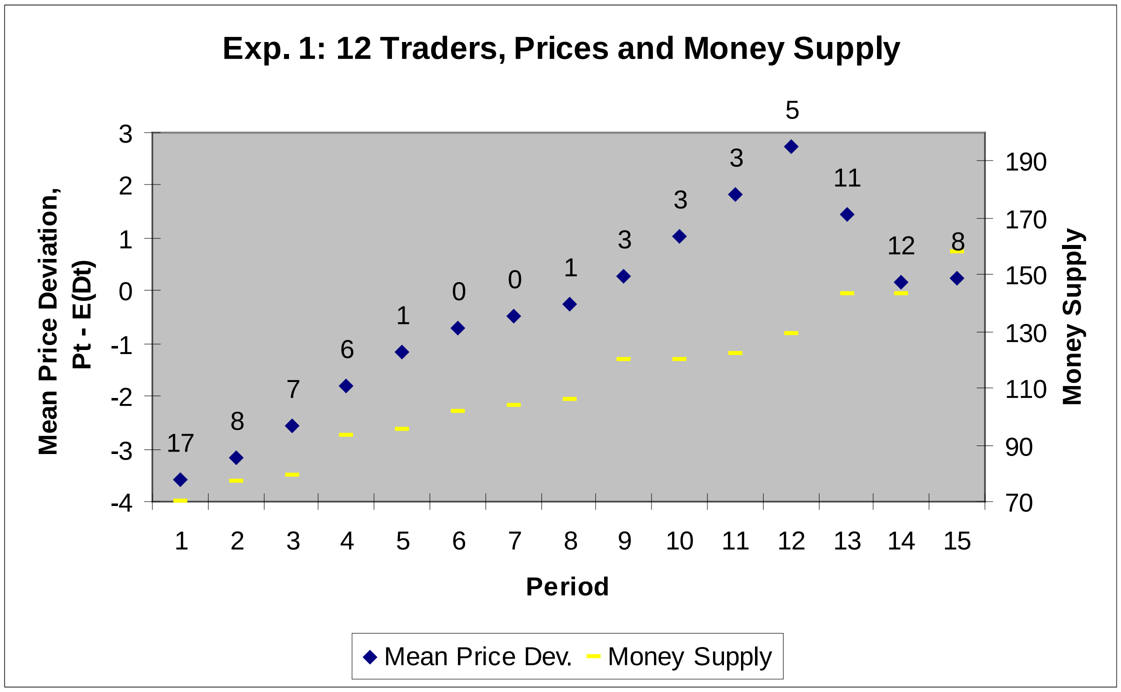


Figure 3: Experiment 1 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

[5] The same statistical tests lead Smith et al. (2000) to reject the hypothesis that experience reduces bubbles, something that is considered an empirical regularity.

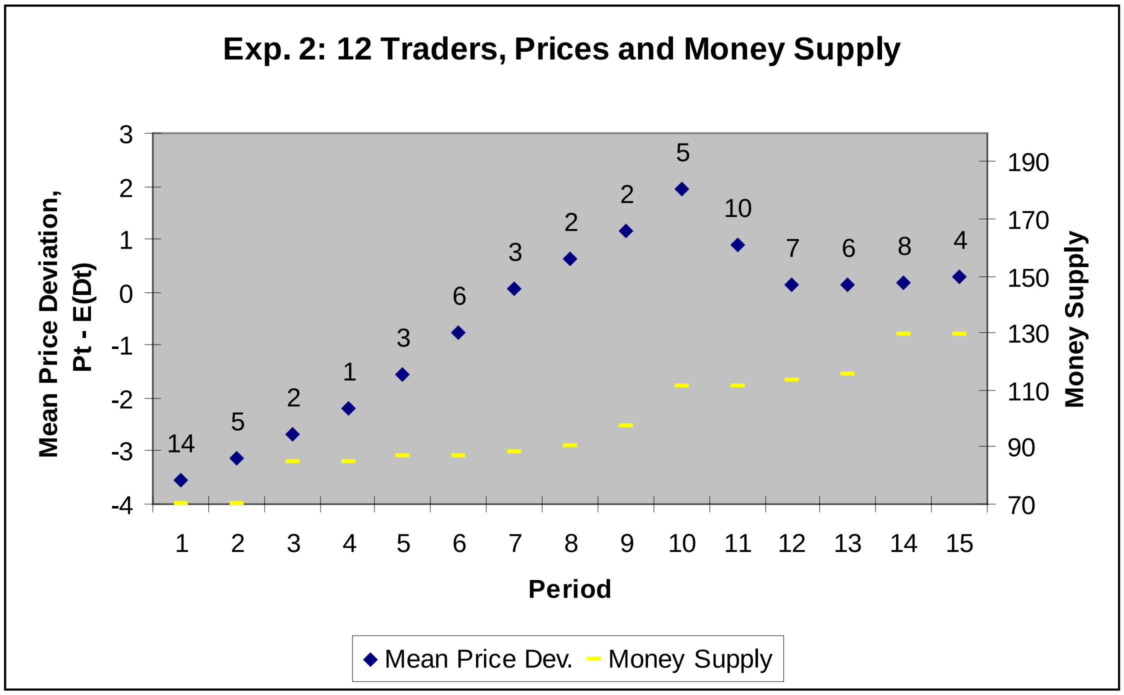


Figure 4: Experiment 2 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

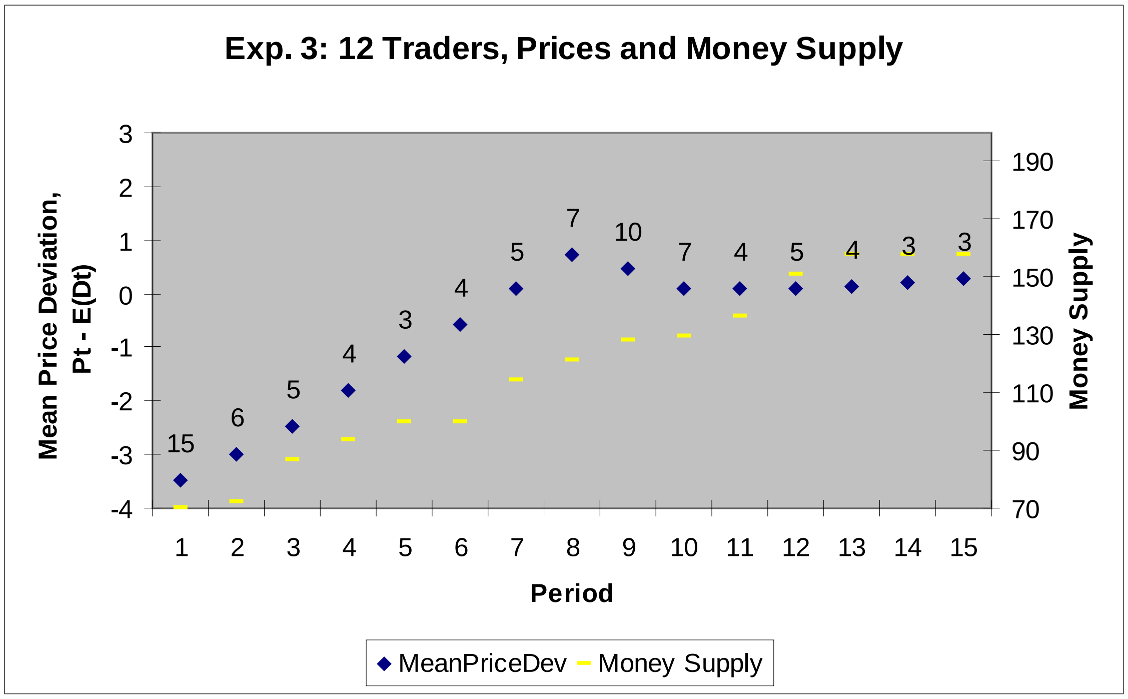


Figure 5: Experiment 3 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

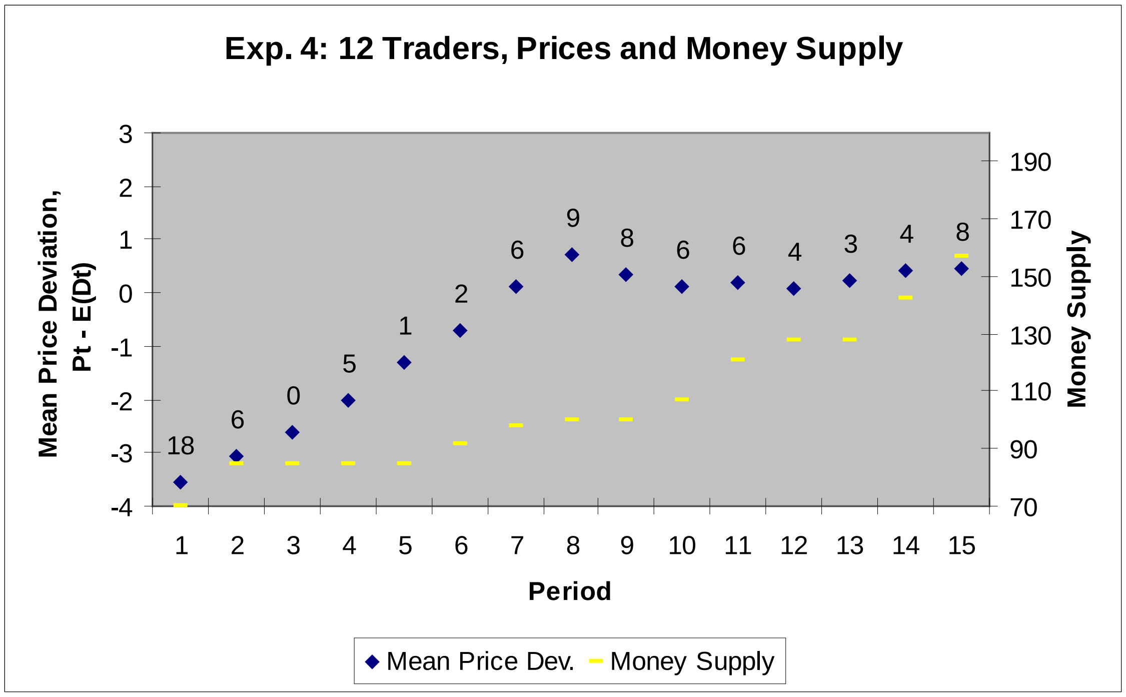


Figure 6: Experiment 4 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

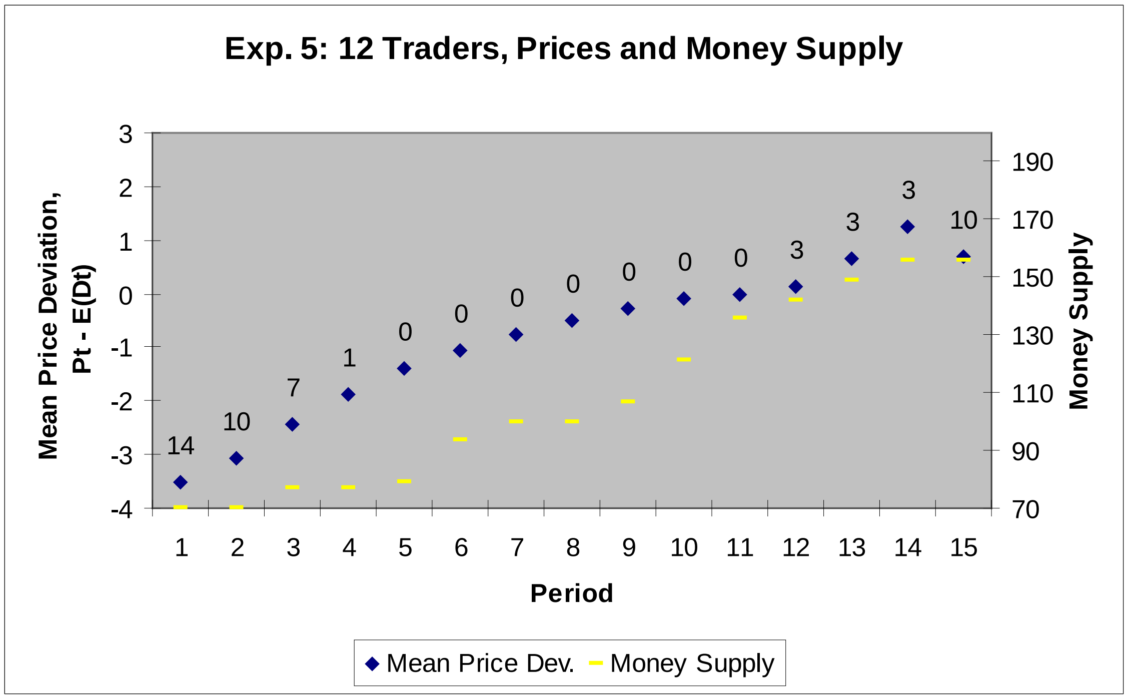


Figure 7: Experiment 5 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

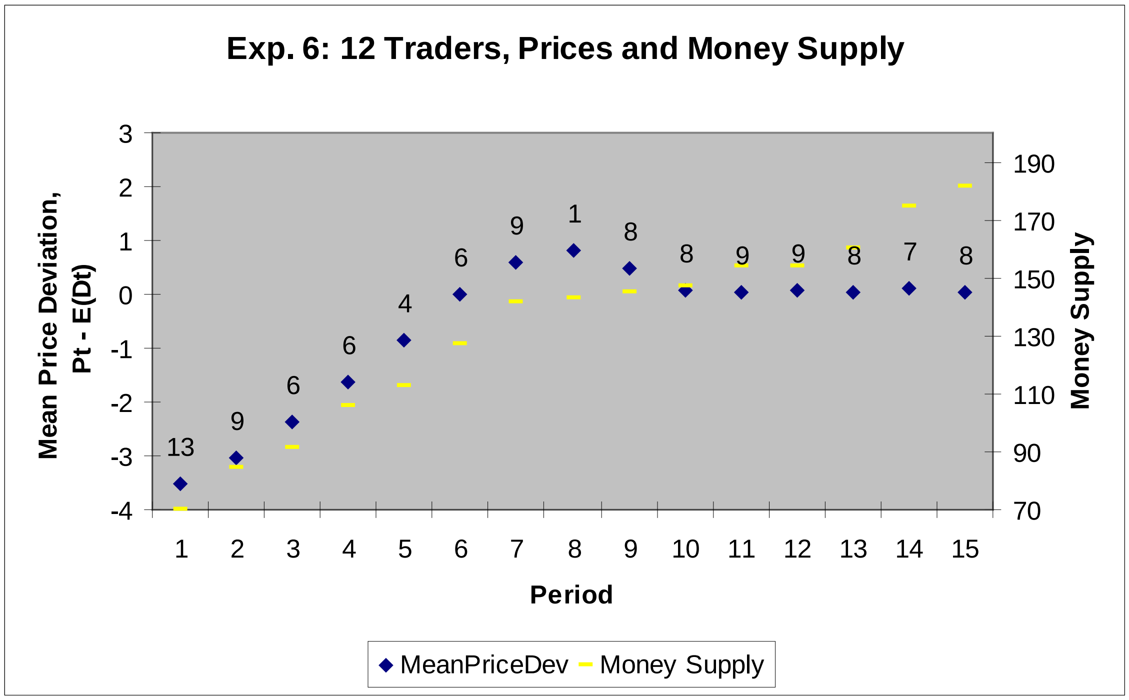


Figure 8: Experiment 6 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

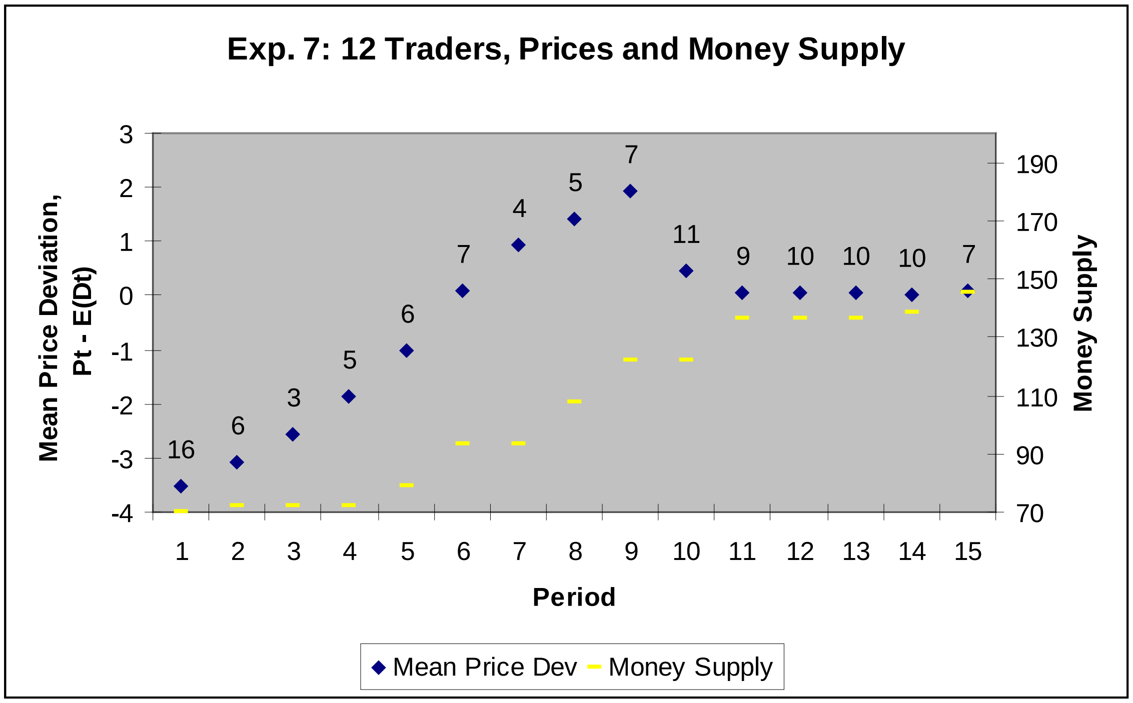


Figure 9: Experiment 7 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

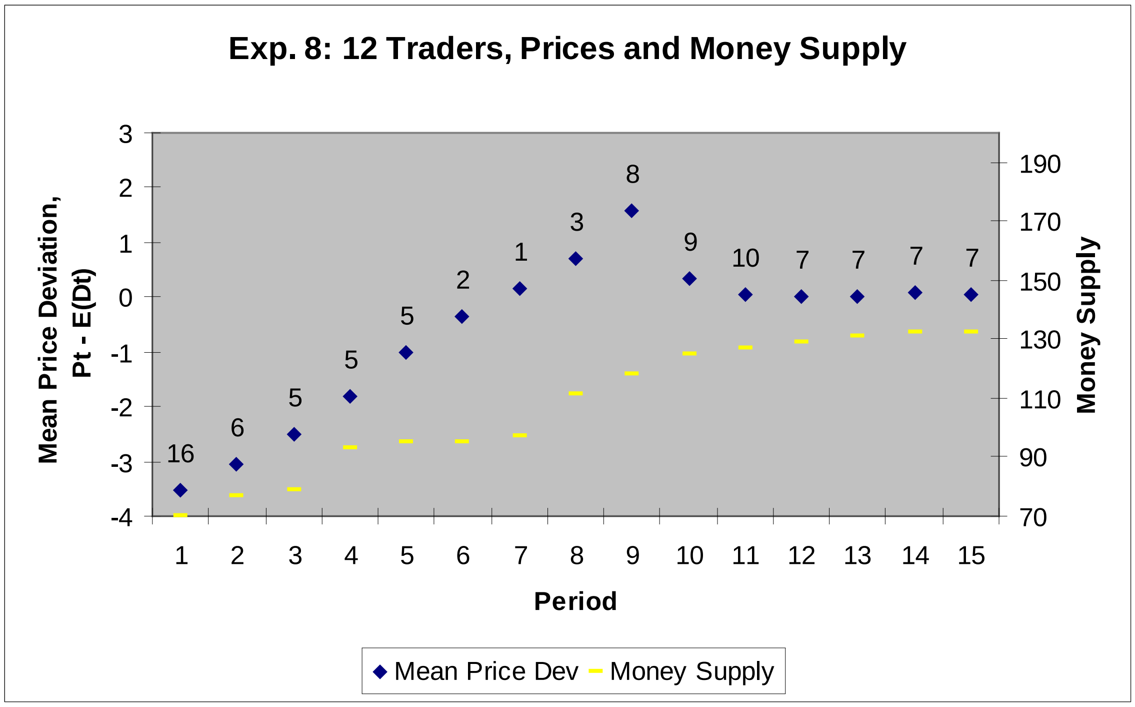


Figure 10: Experiment 8 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

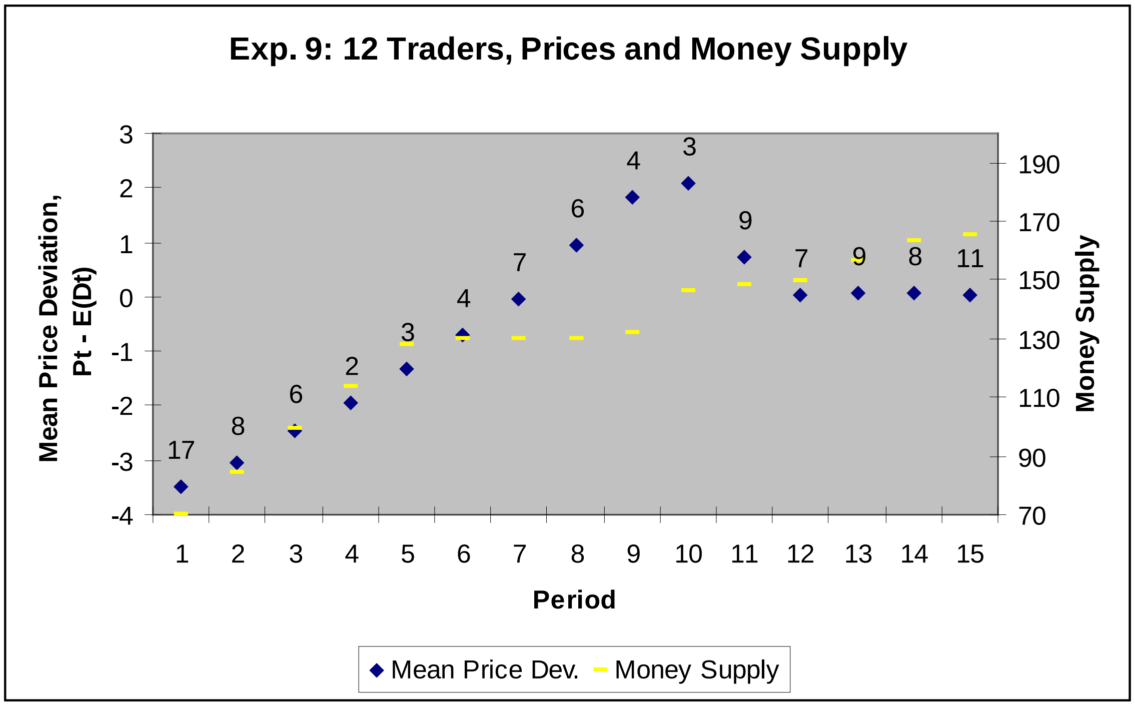


Figure 11: Experiment 9 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

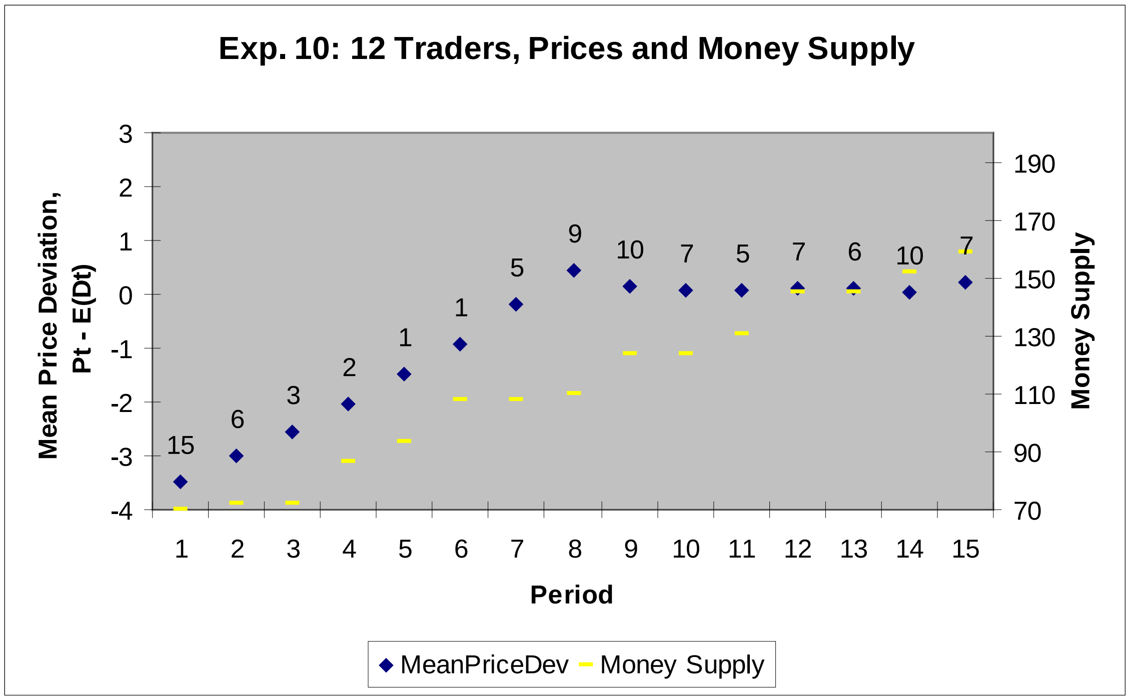


Figure 12: Experiment 10 showing the mean price deviation and the money supply. Volume in each period is shown above the mean price deviation.

4.12 Strategy Performance

To gauge strategy performance, I group all 10 random dividend experiments together, i.e. experiments 1-10, and look at total cash earnings. The results are summarized in Table 2, where it is clear that the value strategy is the most profitable. All 20 of the top earners are value traders, and the only value traders that perform below average are those with exceedingly low strategy confidence. For value traders, the correlation between earnings and the learning factor is 0.71, meaning the stronger the strategy the higher the earnings. The average earnings for value traders is $25.31, versus

$5.18 for momentum traders and $6.07 for speculative traders. There is also a slight correlation (0.17) between earnings and endowment class, meaning a high cash-low asset endowment facilitates earning for value traders. The value strategy succeeds because the initial periods, in which price is below fundamental value, are spent accumulating assets cheaply and then they are liquidated during the bubble phase.

The same applies to speculative traders, for whom there is a high correlation (0.8) between earnings and endowment class. However, learning is not correlated with earnings for speculative traders. While there is a slight correlation between earnings and the randomly drawn bullish return (0.16), there is no correlation between earnings and the randomly drawn correction percentage (0.02). For speculators, it appears not to matter much how bullish or how cautious you are, nor how quickly you learn.

For momentum traders, there is some correlation between earnings and learning (0.44), and earnings and endowment class (0.6). This indicates that for them also early accumulation of assets is the key.

Table 2: Trader earnings summary and correlations.

				Correlations with Earnings			
Trading Strategy	**Mean Earnings (E$)**	**Std. Dev. Earnings (E$)**	**Earnings Range (E$)**	**Learning Factor**	**Endowment Class (Cash/Asset Ratio)**	**Speculative Expected Bull Return**	**Speculative Correction Percentage**
Value	25.31	30.07	110.89 - 0.62	0.71	0.17		
Momentum	5.18	3.09	14.38 - 1.15	0.44	0.60		
Speculative	6.07	2.93	11.42 - 1.72	0.00	0.80	0.16	0.02

4.13 Experience

Another empirical regularity in the laboratory is that experience tends to reduce the tendency to bubble. In these simulations, experience comes in either by increasing *MaxLF*, the maximum learning factor, or increasing the percentage of value traders. Figure 13 shows two experiments with *MaxLF* increased to 1.0 from 0.2 in experiments 1 through 10. Clearly, price reaches fundamental value much faster, typically in period 4. I ran 10 experiments using this higher *MaxLF*, and found that only 1 of those produced any bubble contracts. Thus, this appears to be an effective parameter setting, representing once or twice experienced subjects.

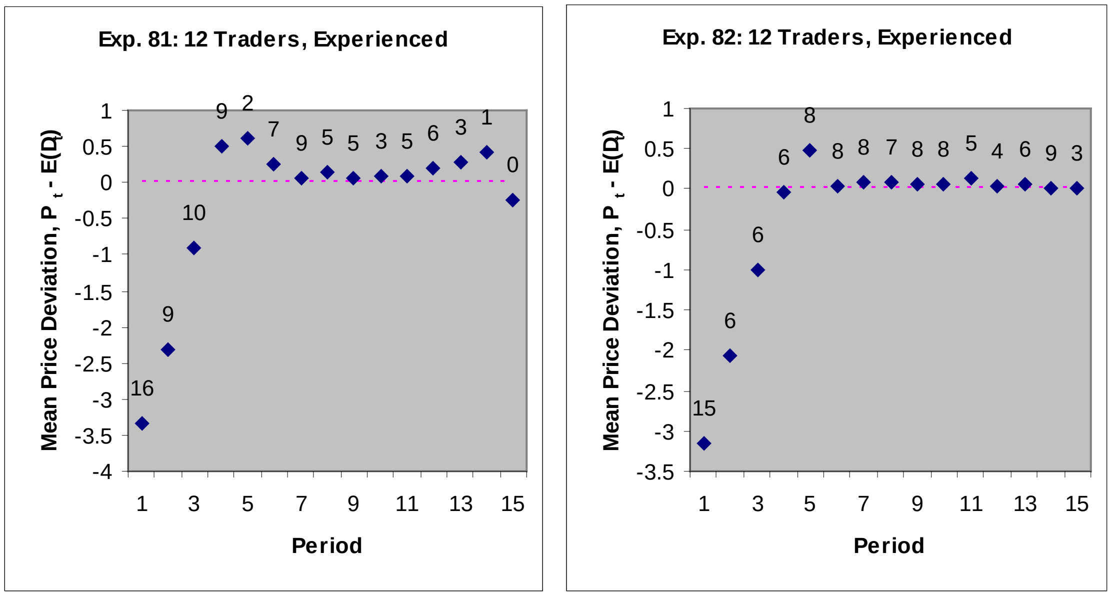


Figure 13: Two experiments of ten, illustrating experience. MaxLF = 1.0. Volume in each period is shown above the mean price deviation.

4.14 Learning Model

There is no doubt that the shape of the bubble is in part enforced by the learning model. To observe what part the learning model is doing, I removed the trading strategies and instead used just the learning model, as follows. For momentum trading:

$$M{:}WTP_{t+1} = p_t + C_t$$

For value trading:

$$V{:}WTP_{t+1} = p_t + VC_t$$

And for speculative trading:

$$S{:}WTP_{t+1} = p_t + C_t,$$

Figure 14 shows two experiments using just the learning model. It appears that the learning model alone is capable of generating much of the bubble up stage, but not the subsequent peak, plateau, or crash. In fact, trading volume goes to zero after periods 3 or 4.

I also ran a number of experiments (not shown) using a linear form for the learning model. The results indicate that a linear model tends to smooth out price movements in the latter periods, particularly because momentum traders are still less than 100% confident. Since this doesn't match up very well with the laboratory data, I used a quadratic form.

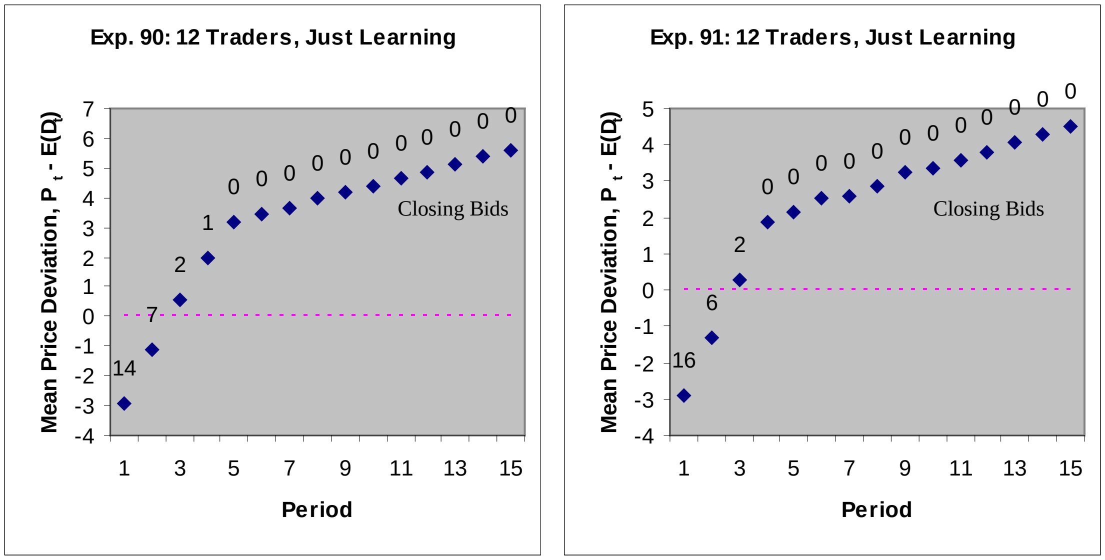
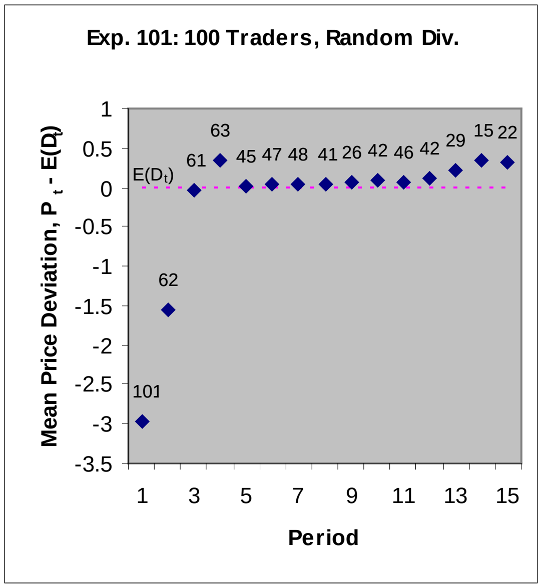
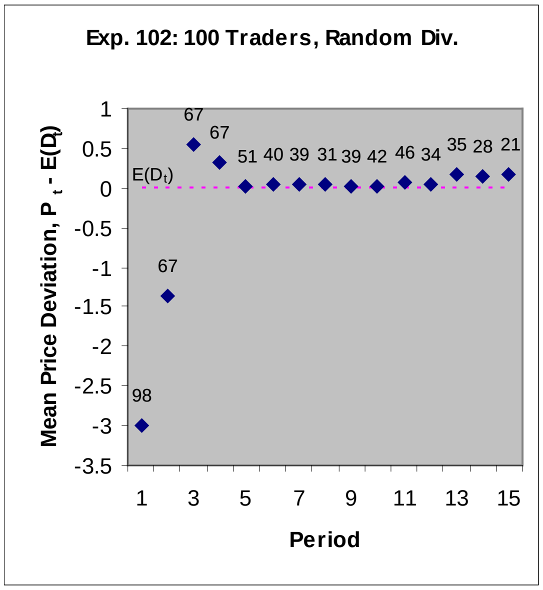


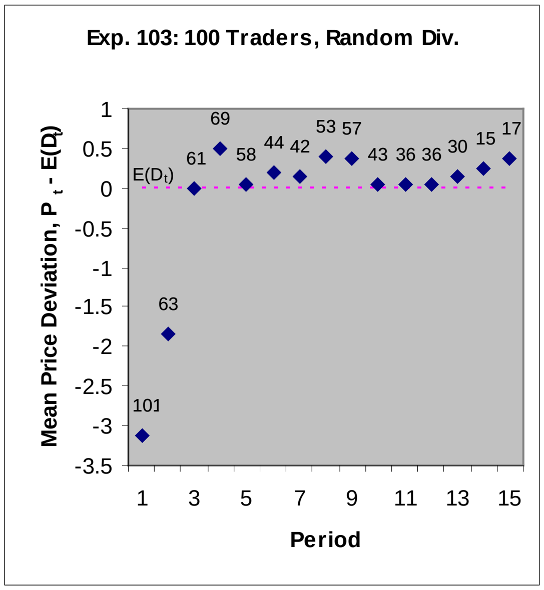
Figure 14: Two experiments with the learning model alone. That is, without any trading strategy. Volume in each period is shown above the mean price deviation.

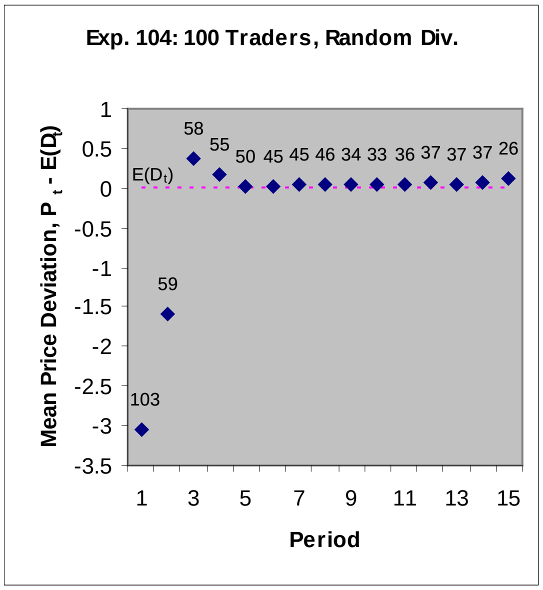
4.2 Larger Scale: 100 Traders

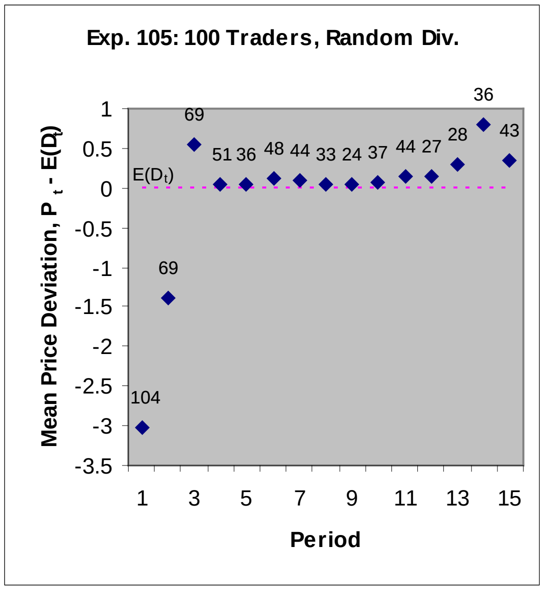
Figure 15, which shows all 10 experiments with 100 traders, indicates that the tendency to bubble decreases with scale. This is the case despite my efforts to maximize the bubble tendency by altering the parameters. For instance, using the same parameter values from the experiments with 12 traders, i.e. the correction cutoff and price average length, reduces the tendency to bubble yet further. Also, it is likely that if the learning model depended on contracts rather than periods, then this would reduce the tendency to bubble yet further.

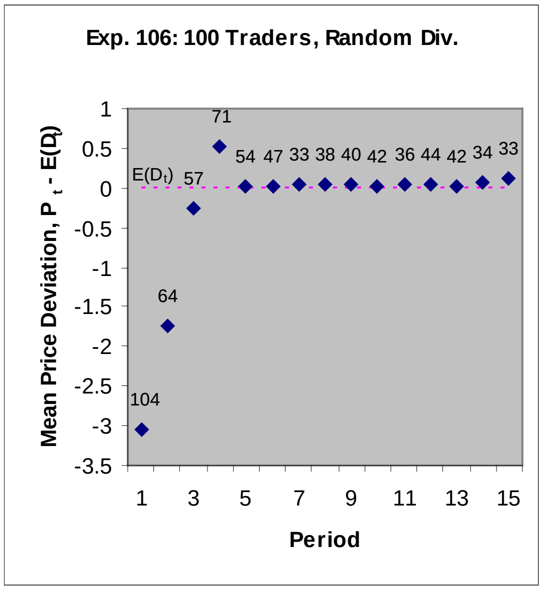





Exp. 103: 100 Traders, Random Div.
Mean Price Deviation, P_t - E(D_t)
Period
E(D_t)
101
63
61
69
58
44
42
53
57
43
36
36
30
15
17


Exp. 104: 100 Traders, Random Div.
Mean Price Deviation, P_t - E(D_t)
Period
E(D_t)
103
59
58
55
50
45
45
46
34
33
36
37
37
37
26


Exp. 105: 100 Traders, Random Div.
Mean Price Deviation, P_t - E(D_t)
Period
E(D_t)
104
69
69
51
36
48
44
33
24
37
44
27
28
36
43


Exp. 106: 100 Traders, Random Div.
Mean Price Deviation, P_t - E(D_t)
Period
E(D_t)
104
64
57
71
54
47
33
38
40
42
36
44
42
34
33

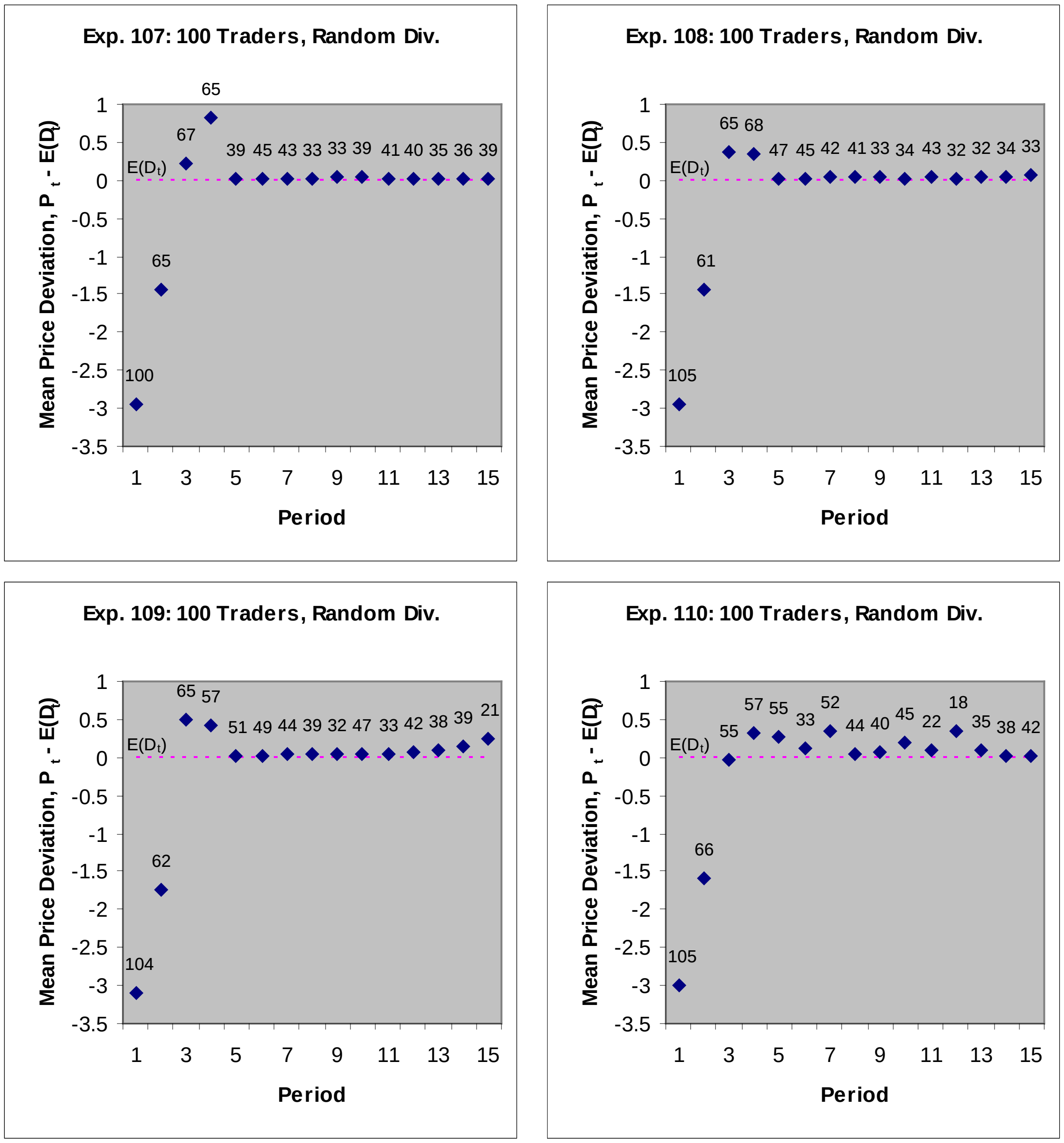
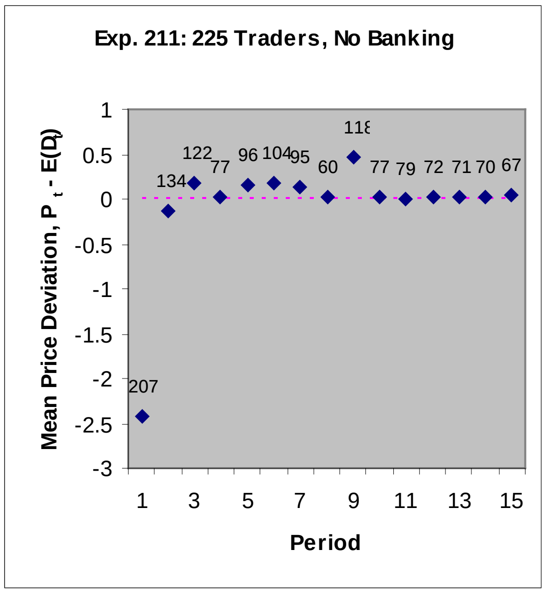
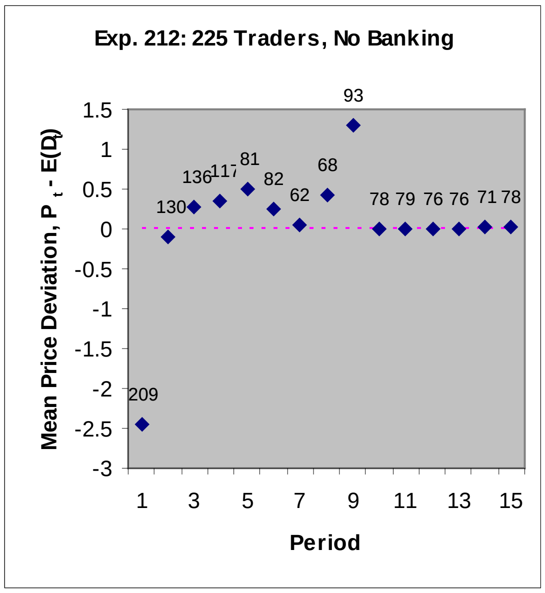


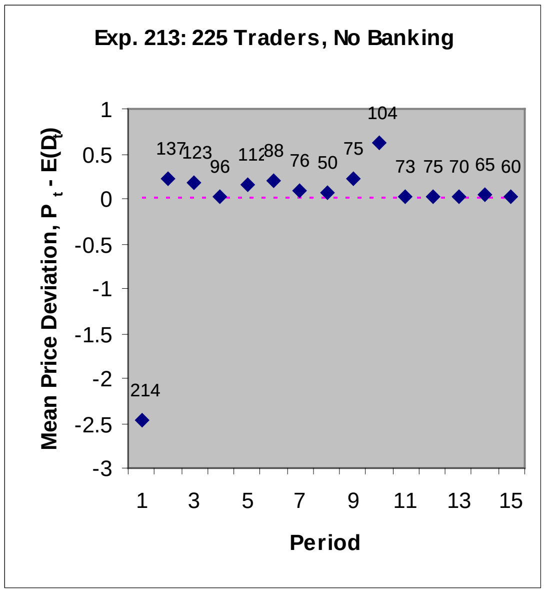
Figure 15: All 10 experiments using 100 traders and a random dividend. That is, the dividend is drawn randomly from four possibilities [0,8,28,60]. Volume in each period is shown above the mean price deviation.

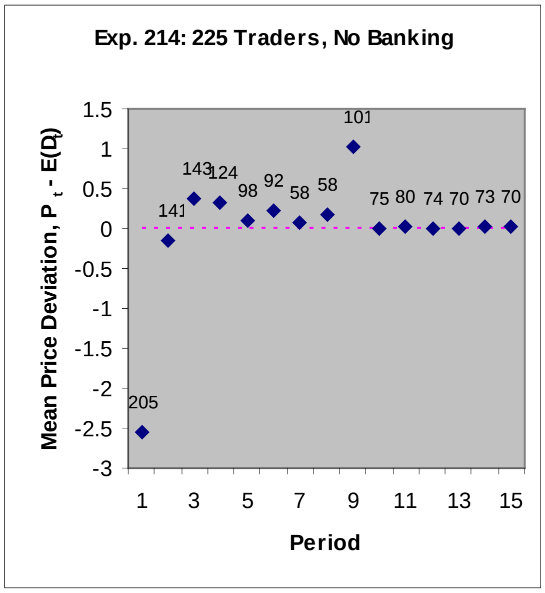
4.3 Largest Scale: 225 Traders

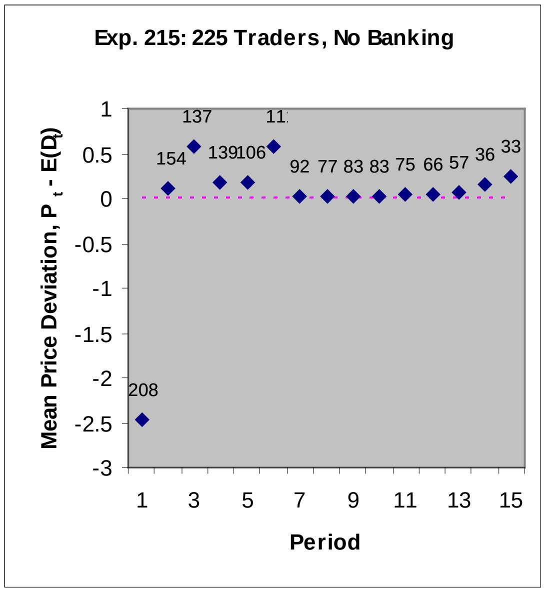
Figure 16, which shows 10 of the 20 experiments with 225 traders, indicates that this scale also exhibits a reduced tendency to bubble as compared to the laboratory scale of 12 traders. However, this scale exhibits a greater tendency to bubble than the scale of 100 traders. This can be seen in Table 3 as well, which lists all the main experiments across scales, along with various measures of bubble tendency. It is not clear why there would be such a non-linearity across scales. However, it is fortunate, as 225 will be the scale used in Chapter 2 to compare bubble tendency across banking treatments.

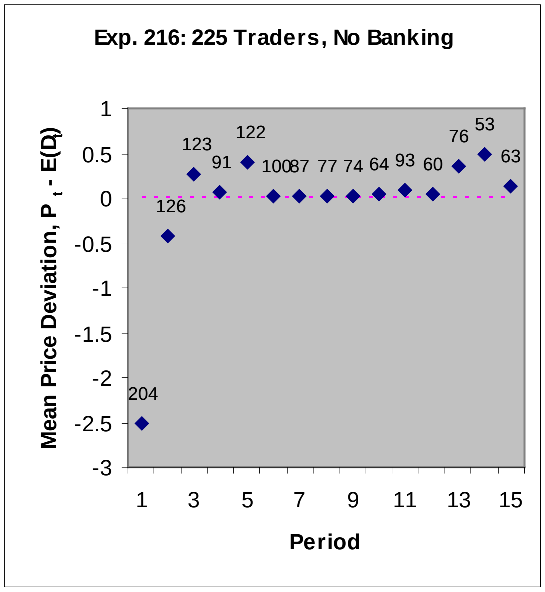





Exp. 213: 225 Traders, No Banking
Mean Price Deviation, P_t - E(D)
Period
214
137
123
96
112
88
76
50
75
104
73
75
70
65
60


Exp. 214: 225 Traders, No Banking
Mean Price Deviation, P_t - E(D)
Period
205
141
143
124
98
92
58
58
101
75
80
74
70
73
70


Exp. 215: 225 Traders, No Banking
Mean Price Deviation, P_t - E(D)
Period
208
154
137
139
106
92
77
83
83
75
66
57
36
33


Exp. 216: 225 Traders, No Banking
Mean Price Deviation, P_t - E(D)
Period
204
126
123
91
122
100
87
77
74
64
93
60
76
53
63

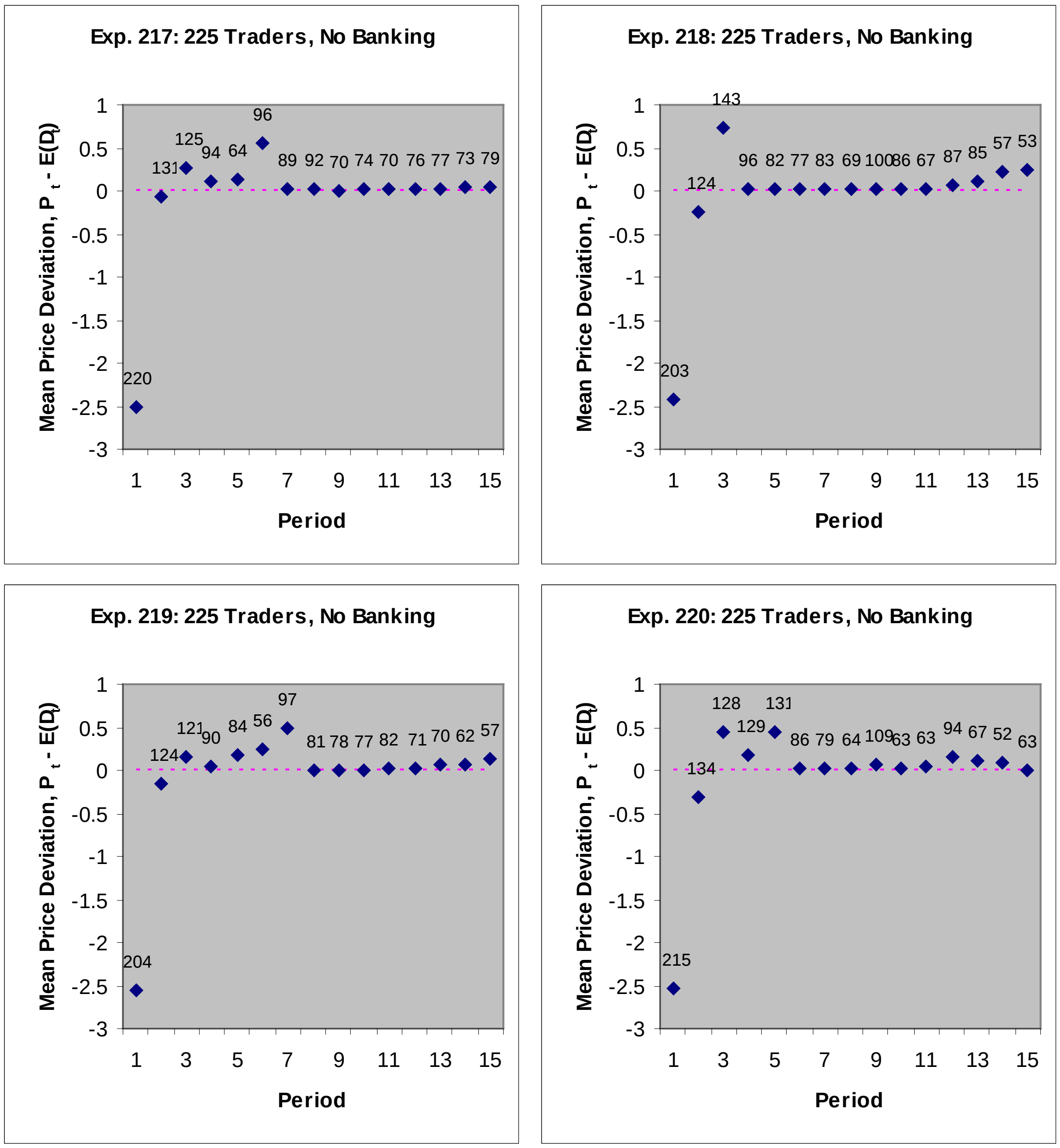


Figure 16: 10 experiments with 225 traders. Volume in each period is shown above the mean price deviation.

Table 3: Main experiments across scales. Bubble tendency is measured using a) the percent of experiments with a mean (over contracts in a period) price bubble, where a bubble is defined as a price deviation exceeding 1.0, b) percent with any price bubble contracts, c) the mean (over experiments) number of bubble contracts, and d) the mean (over experiments) percent of contracts which bubble.

Experiments	Traders	Number of Experiments	Percent with a Mean (over period) Price Bubble	Percent with Any Bubble Contracts	Mean (over experiments) Number of Bubble Contracts	Mean (over experiments) Percent Bubble Contracts
1-10	12	10	60	70	7.1	7.5
101-110	100	10	0	70	9.3	1.3
201-120	225	20	30	95	52.5	3.8

5.0 Conclusion

In summary, this chapter describes an agent-based model and simulation of the canonical laboratory double auction asset market as originally designed by Smith et al (1988). The agent-based approach allows for scaling up of traders, a necessary condition for testing the banking hypothesis described in Chapters 2. For traders, I assume three types of trading strategies: value, momentum, and speculative. In addition, I apply a learning model to all traders, as learning qua adaptive expectations features prominently in the laboratory experiments, where only experience consistently reduces bubble tendency. My results successfully match many of the empirical regularities found in the laboratory, most importantly the tendency for prices to start below fundamental value and then bubble above and crash back to fundamental value by the end of the experiment. I am also able to generate such bubbles, though to a lesser degree, at larger scales of 100 and 225 traders. While the learning model used here appears to perhaps heavy-handedly enforce the direction of the bubble up phase, it does not account for the peak, plateau, or the crash. However, it is certainly a worthwhile research project to find a combination of trading strategies which relies less on learning to generate bubbles.

In addition, unlike the laboratory experiments, I focus on the money supply conditions implicit in the design of the dividend structure, which reveals that this is an environment where money supply typically more than doubles over the course of the experiment. In light of the results of Porter and Smith (1995) and Caginalp, Porter, and Smith (1998) who find that increasing liquidity increases the tendency to bubble, it should be clear that this environment is inherently prone to bubble.

Lastly, in terms of trading strategy performance, I find that value trading is the most profitable, which makes sense within the context of the laboratory design, not to mention the fact that the world’s most successful trader, Warren Buffet, is also a value trader.

CHAPTER 2: AN AGENT-BASED MODEL AND SIMULATION OF ASSET TRADING IN A DOUBLE AUCTION PLUS BANKING

1.0 Introduction

The ongoing financial crisis and resulting government intervention has called into question the wisdom of our current banking regime. Some reform proposals would expand the regulatory powers of the Federal Reserve, while others would curtail or even repeal those powers. It is worthwhile then to consider a world without central banking, a world which has not existed in practice since at least the Great Depression. Before this time and throughout much of the 18th and 19th centuries, many countries, such as Canada, Australia, and Scotland, practiced so called “free banking” or competitive note issue without control by a central bank (see White, 1995, and Dowd, 1992). Free banks enjoyed important freedoms, particularly the right to issue their own notes, contractually redeemable in a market determined monetary base (typically specie), and without any statutory reserve requirements.

Modern day proponents of free banking, including George Selgin (1988, pp. 52-85, 1994) and Lawrence H. White (1995, pp. 12-17, 107-115), argue it is capable of limiting fluctuations in the general price level by making for a more stable relationship between the volume of base money present in an economy and corresponding levels of credit and aggregate spending.

A separate literature, the experimental asset bubble research stemming from Vernon Smith et al. (1988), appears to support the notion that unsustainable booms are behavioral in origin - the result of Keynesian “animal spirits” and as such an unavoidable aspect of unfettered free markets. To be more precise, a bubble is an extended period of time during which an asset trades above fundamental value, which outside the lab typically can only be known with any degree of certainty ex post when the bubble has burst. Inside the lab, fundamental value can be induced and thus known with complete certainty ex ante (Smith, 1976). Smith et al. (1988) and numerous experimenters since have found that, even when subjects are informed ex ante of fundamental value, asset bubbles occur frequently and are robust to various treatment conditions. Subject experience tends to reduce bubbles, but this result is sensitive to changes in treatment conditions. Hussam et al. (2008) find that bubbles can be rekindled among experienced subjects by changing either the asset dividend structure or subject endowments.

The experimental asset bubble literature is largely orthogonal to the free banking literature. The free bankers’ theory of money supply does not allow for animal spirits and resulting asset bubbles. Vernon Smith’s experiments, on the other hand, do not include a well-defined monetary environment, and so do not indicate the extent to which the likelihood of asset bubbles may depend on the nature of banking and monetary arrangements.

In this chapter, I extend the experimental asset bubble research by incorporating specific banking arrangements into Smith’s basic design. In order to do so, as described in Chapter 1, I scale up the number of traders beyond what is possible in the laboratory using agent-based modeling and simulation. The purpose of the scaling is to generate a sufficiently large number of transactions so that bankers have an approximately continuous distribution of clearings by which to judge their optimal precautionary reserves (see Selgin, 1988, pp. 73-78). The tradeoffs of this approach are discussed more fully in Chapter 1. To this framework I add an agent-based model and computer simulation of banking. There are three main treatments:

1) Baseline, which is a simulation of the double auction asset trading environment of Smith et al. (1988), scaled up from 12 to 225 traders. This is described in detail in Chapter 1, as is the fact that the stock of money, which is in experimental dollars, is not constant but rather, as dictated by the original laboratory set up (Smith et al., 1988), consists of the cash endowment plus a random asset dividend payment in each period.

2) Central Banking, wherein two additional agents are added as bankers, and they have all the significant rights and functionalities of bankers except the right to issue notes. Mainly, bankers accept deposits from traders and then lend some portion of that back out to traders at competitively determined interest rates. In other words, this is a system of free banking in deposits, where neither bank has the right to issue notes. There is no representation of the "central bank" beyond the restriction on note issue, but one can assume that the stock of *base* money, which is in experimental dollars, is created exogenously by a central bank. As in the Baseline Treatment, base money enters the system in the form of cash endowments and also per period dividend payments, and the base money is also the currency. The monetary environment is different in two ways: a)

each trader’s money endowment consists of a deposit and a loan with one of the bankers, in such a way that the total money supply, i.e. the total of traders’ cash holdings and deposits, is constant initially across treatments, and b) changes thereafter in the total money supply are determined partly by dividend payments but also by the amount of credit issued by the two bankers according to each banker’s calculation of their demand for precautionary reserves.

3) Free Banking, which also allows the two bankers to issue notes redeemable in experimental dollars. This is a mature free banking economy, so that bank notes have come to fully replace experimental dollars as the currency, while the stock of experimental dollars represents the stock of bank reserves as well as base money. This is implemented by giving bankers a preference for issuing banknotes and traders a preference for using banknotes as currency. Each banker brings the other banker’s notes to the clearinghouse on a regular basis, where they are cleared without cost or discounting. As will be discussed further in the agent-based model section, the settlement process is somewhat different between the two banking treatments. Under central banking, all transactions are in base money including deposits, withdrawals, and loans, in what might be called real-time gross settlement. Under free banking, all transactions, except those involving endowments and dividend payments, are in banknotes and traders have the right, though as mentioned not the preference, to redeem banknotes in base money. On a regular basis, the two free bankers settle each others banknote balances at the clearinghouse using base money, in what might be called end-

of-day net settlement.[6]

My hypothesis, based on the theory of free banking as reviewed in the following section, is that both banking treatments will tend to reduce the frequency and size of asset bubbles as compared to the baseline treatment, and the free banking treatment will be the most effective in this regard. I find some support for this hypothesis, despite a challenging monetary environment of increasing supply of base money. Particularly, while I do not find a significant difference between the banking treatments, in terms of reduction of bubble tendency, I do find both banking treatments significantly reduce bubbles and they do so by constraining the total money supply. This latter result represents a novel demonstration of the principle of adverse clearings, as described by Selgin (1988, pp. 37 – 42), whereby free banks are limited in their note issue by the total stock of base money and the need to settle in base money with other banks any outstanding banknote balance. As Selgin explains, the principle of adverse clearings, though controversial, is merely a generalization of the more commonly accepted rule of excess reserves, whereby in a system with monopolized currency supply, an individual bank is limited in the loans and investments it may issue by the fact that total reserves are finite and it must settle in base money all currency demands and clearing balances owed to other banks. Thus, my results provide support for both the principle of adverse clearings and the rule of excess reserves.

Lastly, it should be noted that there is clearly more to central banking in practice than the stylized version used in this study, which represents only the aspect of monopoly

[6] This is how Lawrence H. White has described it.

of note issue. However, there is little agreement on what completely defines central banking. In an earlier version of this paper I proposed a more complicated experiment in which the central bank, in addition to monopolizing note issue, manipulates the interest rate and the money supply through open market operations. I may eventually return to that idea. In the meantime it is a fairly safe conjecture that, under reasonable assumptions, this more complete representation of central banking would perform far worse than the current one in terms of bubble reduction. The reason is, as Selgin (1988, p. 49) notes, "Unless some external short-run control is imposed on it, a monopoly bank of issue even when its issues are convertible into commodity money can for some time at least pursue any loan-pricing policy it desires, arbitrarily expanding or contracting the money supply and causing widespread changes in nominal income and prices." Since the current design does not allow for such central bank initiated fluctuations in the money supply, the results should be interpreted as a comparison of free banking to a rather idealized version of central banking.

2.0 Theory of Free Banking

To briefly review the theory of free banking it is useful to consider the equation of exchange:

$$MV = Py,$$

where M is the money supply, V is the velocity of money, P is the (output) price level, and y is real income or output. For any given value of real income, the general price level will remain stable only if MV remains stable, i.e. only if changes in the money supply serve to just offset changes in money's velocity. A free banking system is supposed to tend automatically to stabilize MV, given some fixed stock of base money, owing to the tendency for any increase in MV to go hand-in-hand with an increase in the volume of interbank clearings, which in turn raises the demand for precautionary bank reserves by increasing the risk of a settlement default associated with any given aggregate reserve stock. As a result, the increase in aggregate spending generates an excess demand for reserves, forcing banks to reduce their loans and to thereby contract the outstanding supply of bank money until the quantity of reserves demanded is once again equal to the quantity on hand. In other words, under an assumption of fixed reserves, in this scenario of an increase in MV, the only way for free banks to bring about equilibrium in reserves is to reduce the demand for reserves to its original level, which requires that MV be reduced to its original level.

Further, Selgin (1994, p. 1453-57) shows that this result, where free banks tend to adjust the money supply so as to completely offset changes in velocity, depends critically on the freedom of banks to issue notes. When this freedom is revoked, money supply adjustments are less than complete (1994, p. 1457). The derived money supply equations are reproduced here. Equation 6 (p. 1453), representing the money supply under a system of free banking, is as follows:

$$M^* = R/(VQ)[(\varphi\, y)/(2(b\text{-}1))]^{1/2} \qquad (6)$$

Where R is the total stock of reserves, b is the number of banks, φ is the proportion of real income transactions to total transactions, and Q essentially represents banker conservatism but will be discussed in greater detail later. Equation 6 implies that the stock of money adjusts in inverse proportion to changes in velocity. Contrast this with the money supply under a system of monopolized note issue, shown as follows (p. 1457):

$$M^* = R/(VQ)[(\varphi\, y)/(2(1 - c)(b\text{-}1))]^{1/2} + R/c$$

Where c is the ratio of currency to deposits. Due to the second term, reflecting the reduction in reserves resulting from base money used as currency, the stock of money does not fully adjust so as to offset changes in velocity. Therefore, in the simulations I expect bubbles to be least prevalent in the free banking treatment, somewhat more prevalent in the central banking treatment, and most prevalent in the baseline treatment.

Finally, Selgin (1994, p. 1452) also discusses the meaning and origin of Q, or banker conservatism, stating:

> A well-known conclusion of the literature on precautionary reserve demand, beginning with Edgeworth, is that a bank's demand for reserves will be

proportional to the standard deviation of its net reserve losses (Baltensperger, 1980, p. 9; Laidler, 1991, p. 186). If we denote a bank's demand for reserves as σQ, then the factor of proportionality, Q, reflects the representative bank’s desired level of security against the prospect of a settlement default, chosen to equate the marginal opportunity costs (forgone interest) and marginal benefits (reduced anticipated costs of reserve shortages) of reserve holding. The value of Q will in general be positively related to reserve shortage penalties and negatively related to the rate of interest (Olivera, 1971, pp. 1102-3).

Q is treated as a parameter in the model to be discussed in the next section, and the process of selecting the value of *Q* is discussed in the parameters section (3.1).

3.0 Agent-based Model and Simulation Software

Figure 17 is a flow chart of the simulation software. A significant challenge for the asset trading and banking simulation was to begin the asset trading under conditions of monetary equilibrium, where money demand equals money supplied. Also, bankers should ideally be able to calculate and achieve their optimal precautionary reserve demand as quickly as possible, i.e. before the bubble action starts. The solution I settled on was to introduce a preliminary period in which each trader’s cash endowment, given in experimental dollars (E$), is split into a portion which is deposited and a portion which is borrowed and held in cash. In this way, the initial money supply is held constant across all treatments. The portion which is deposited, α, is determined by the initial reserve ratio, a parameter, as follows:

$$\alpha = 1/(2 - \mathit{InitialReserveRatio})$$

The remaining portion of the endowment, 1- α, is borrowed and held in cash. Thus, for example, a 50% initial reserve ratio results in two thirds of the cash endowment deposited and one third borrowed and held in cash. The banks are then left with a reserve ratio of 50%.

I used an initial reserve ratio of 50% because I found that this reserve ratio was close to where bankers settled in later periods; somewhat more at smaller trading scales and somewhat less at larger scales. The initial reserve ratio is in place until bankers have a sufficient clearing sample by which to judge their optimal reserve ratio and precautionary reserve demand. The clearing sample is used by bankers to estimate the clearings distribution, or, in other words, the net reserve loss distribution. As mentioned earlier, the banking literature finds that a bank's precautionary reserve demand is proportional to the standard deviation of its net reserve losses (Selgin, 1994, p. 1452). For the clearing sample length, I settled on roughly one period's worth of clearings: 10 clearings under free banking and 500 under central banking. The disparity is due to the previously mentioned distinction between real-time gross settlement, in the case of central banking, and end-of-day net settlement, in the case of free banking. Particularly, under free banking, clearings only occur at the clearinghouse, at the end of each trading interval, and there are 10 intervals per period. At the clearinghouse, bankers submit each others banknotes, then the clearinghouse calculates each bank's balance, without charge or note discounting, and finally requests settlement in base money (E$). Under central banking, any sort of bank transaction counts as a clearing, since all currency is in base money (E$) and thus any transaction directly effects base money reserves. For example, a deposit or withdrawal counts as a clearing, which can happen as often as once per trader per interval, so clearings are always much more frequent under central banking.

As mentioned earlier, precautionary reserve demand is proportional to the standard deviation of net reserve losses, σ, and the factor of proportionality is Q. Thus, precautionary reserve demand, *PRD*, is as follows:

$$PRD = \sigma Q,$$

Where,

$$\sigma = [(SSD_C)/(ClearingSampleSize - 1)]^{1/2},$$

And SSD_C is the sum of the squared deviations of clearings around the mean.

Precautionary reserve demand is calculated at the beginning of each trading interval. The optimal reserve ratio, *ORR*, is calculated on an as needed basis and is based on *PRD* as follows:

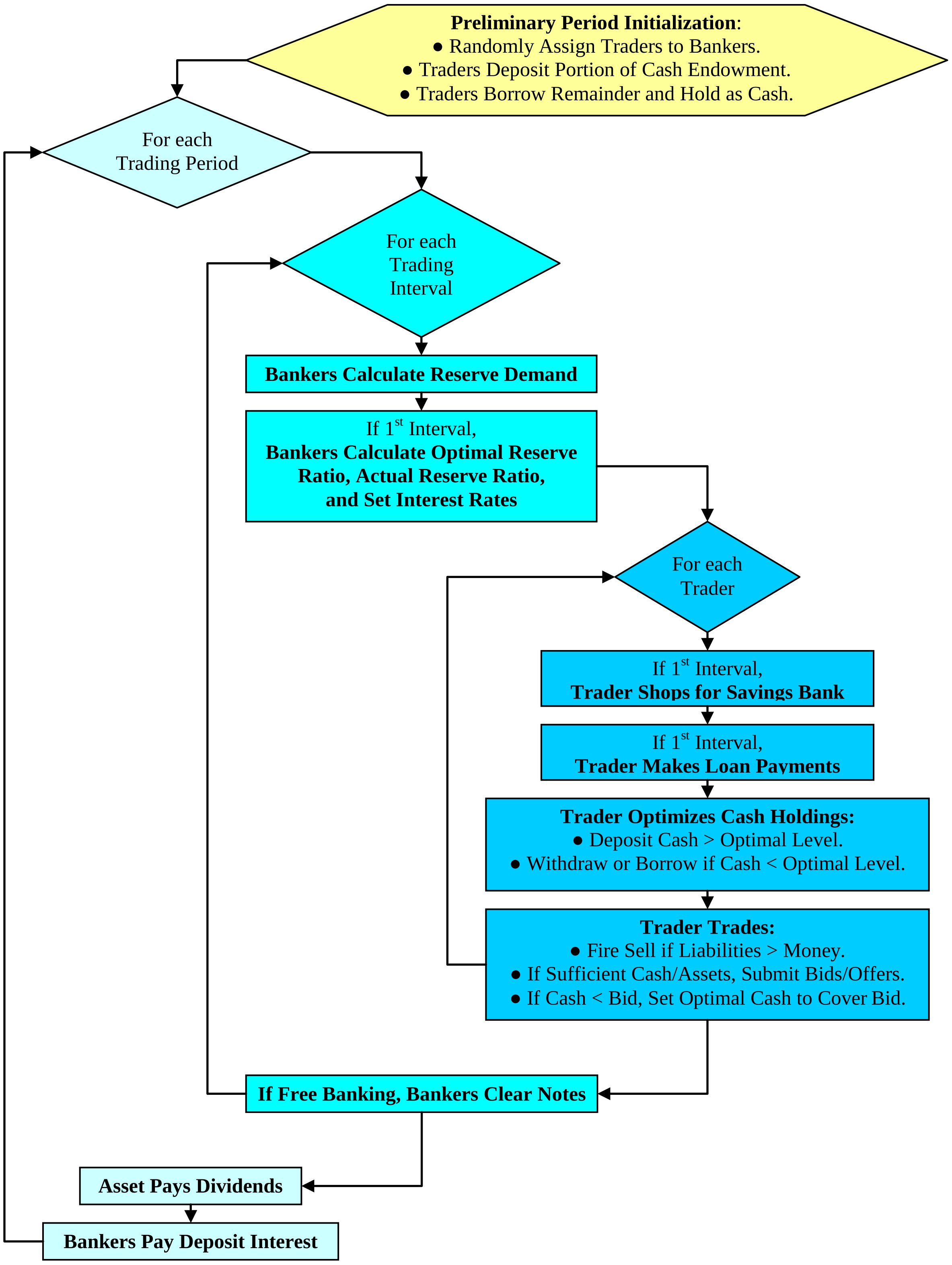
$$ORR = PRD/(Deposits + IssuedNotes)$$

Likewise, the actual reserve ratio, *ARR*, is calculated on an as needed basis as follows:

$$ARR = Reserves/(Deposits + IssuedNotes)$$

Base money (E$) counts as reserves under both free and central banking. Additionally, under free banking, other bank's notes count as reserves.

Under free banking, so as to quickly achieve a mature free banking economy where notes have fully replaced base money as currency, whenever a trader withdraws or borrows money the bank pays that money in newly issued bank notes. Thus, in the preliminary period when endowments are split into deposits and borrowed cash, the borrowed cash is in notes from the bank where the money was deposited. Further, the preferred money for each trader is always notes from his depository bank. Therefore, assets are always purchased in a trader's depository bank notes. Furthermore, after the preliminary period, the only time base money (E$) enters the hands of traders is when dividends are paid in base money at the end of each trading period. Since this is not preferred money, traders immediately deposit this with their depository bank.


Preliminary Period Initialization:
● Randomly Assign Traders to Bankers.
● Traders Deposit Portion of Cash Endowment.
● Traders Borrow Remainder and Hold as Cash.
For each Trading Period
For each Trading Interval
Bankers Calculate Reserve Demand
If 1st Interval, **Bankers Calculate Optimal Reserve Ratio, Actual Reserve Ratio, and Set Interest Rates**
For each Trader
If 1st Interval, **Trader Shops for Savings Bank**
If 1st Interval, **Trader Makes Loan Payments**
Trader Optimizes Cash Holdings:
● Deposit Cash > Optimal Level.
● Withdraw or Borrow if Cash < Optimal Level.
Trader Trades:
● Fire Sell if Liabilities > Money.
● If Sufficient Cash/Assets, Submit Bids/Offers.
● If Cash < Bid, Set Optimal Cash to Cover Bid.
If Free Banking, Bankers Clear Notes
Asset Pays Dividends
Bankers Pay Deposit Interest
Figure 17: Simulation Software Flow Chart.

Under both free and central banking, in the preliminary period traders are randomly assigned to banks so that each bank has the same number of traders as well as the same distribution of types of traders. As mentioned earlier, a portion of the endowment is then deposited at the assigned bank and the remainder of the endowment is borrowed from the same bank as a short term loan. There are two types of loans available to traders, short and long term, where the duration of the short term loan is 2 periods and that of the long term loan is 4 periods. Interest is due at the beginning of each period that the loan is outstanding, and the principle is due in the final loan period. For example, a short term loan taken out in period 3, has interest due in period 4 and 5, and principle due in period 5. Loans may be taken out at anytime during the period, i.e. any trading interval, and the interest due is unaffected. Loans may not be taken out in later periods when the due date is beyond the end of the experiment. Traders are allowed a maximum of one short term loan and one long term loan at a time. Both short and long term loans are of size E$5. The preliminary short term loan is special as the size varies depending on the endowment class of the trader, but as can be seen from the parameters used (section 3.1) this loan ranges from E$0.75 for endowment class 1 to E$3.15 for endowment class 3.

The preliminary loan has interest due in the first trading period and principle and interest due in the second trading period. So as to equalize the initial money supply across treatments, I chose initial interest rates such that, given a 50% initial reserve ratio, depository interest earned by traders would be offset by interest paid on the preliminary short term loan. This also means bankers make no money in the preliminary period. Thus, the initial depository rate is 2% and the short term loan rate is 4%. The long term loan rate is 6%, simply a larger number meant to cover some of the cost of increased exposure to default. Traders prefer short term loans, and a trader will only request a long term loan if he already has a short term loan.

After the preliminary period, bankers are free to adjust their interest rates at the beginning of each period. They do so by comparing their optimal and actual reserve ratios, as follows:

$$iRate_{t+1} = iRate_t + iRateCF * (ORR - ARR)$$

Where $iRate_t$ is the interest rate, either depository, short or long term loan, at time t, and *iRateCF* is the correction factor, which is a parameter set to 0.005. The correction factor was chosen so that over the course of a typical experiment interest rates fluctuate over a range of about plus or minus one percentage point.

The banking simulation fits within the asset trading simulation, so that the timing and structure of asset trading also determines banking. Specifically, each trader banks immediately prior to trading, once per trading interval. If it is the first trading interval, i.e. the beginning of the period, traders make loan payments if any due and potentially move their depository account to the bank paying the highest depository rate. The savings rate elasticity of demand is a parameter used to slow the movement of depositors, without which effectively a bank run would ensue. I found that a savings rate elasticity of demand of 0.7 leads to about 5% of traders switching depository banks in each period.

Next, and at the beginning of every trading interval, traders optimize cash holdings by depositing cash in excess of an optimal level and borrowing when cash is below the optimal level. The optimal level is driven by the demand to buy assets, i.e. if a bid is formed which cannot be supported with cash or savings, then the optimal level of cash is boosted so as to potentially resubmit that bid in the next trading interval. This would usually require a loan issued in the next trading interval, unless an asset is sold or a dividend payment comes in.

The standard bankers use for issuing loans is first determined by calculation of the requesting trader's asset-to-liability ratio, where assets include the tradable assets at market value plus deposits and cash. Liabilities are other loans. The minimum asset-to-liability ratio is a parameter, *MinALR*, set to 0.9, which was chosen in conjunction with other parameters so as to be binding only a small percentage of the time. Secondly, and a much more frequently binding requirement, the banker's actual reserve ratio must exceed his optimal reserve ratio.

Finally, bankruptcy is possible for both traders and bankers, and merely results from an inability to pay claims. In the case of bankers, it is treated as a rare calamity and the experimental session is considered a bankrupt session. In the case of a trader bankruptcy, it is treated as a fairly routine liquidation. First, it is made public, which means no banker will lend to the trader again. Then the immediate creditor is paid all remaining cash and savings. Over the next few trading intervals, all remaining assets are liquidated using market orders, and the proceeds are given to the largest creditor. Thereafter, the trader is essentially out of the business of trading or banking. Traders are keen to avoid such an experience, and they do so by keeping track of their liabilities and when they are due, and then “fire selling” accordingly. If, for example, a trader has a loan due next period and is E$1 short, he will first remove all bids and then discount his willingness to pay, *WTP*, and accept as follows:

$$WTP_{t+1} = discount * WTP_t$$

Where,

$$discount = 0.3 + (TI - s) * 0.1$$

And,

TI is the total number of trading intervals and *s* is the current trading interval. If there are fewer than three trading intervals left, then the trader sells his assets using market orders. I chose parameters such that trader bankruptcy is fairly uncommon, occurring for about 2% to 3% of traders over the course of the experiment, and yet “fire selling” is also uncommon, typically occurring either not all in a period or at most for 5% of traders in a period.

3.1 Parameters

Many of the banking parameters have been discussed in the previous section, but Table 5 provides a formal list of them. Table 4 lists the parameters used in the asset trading portion of the simulation, which are the same as those used in Chapter 1 for both large scale simulations. The reasoning for the choice of the trading parameters is discussed in Chapter 1. I determined that 225 traders split evenly between 2 bankers is close to the minimum number of traders per banker necessary to generate a sufficiently stable and low optimal reserve ratio. Namely, fewer traders than this results in a reserve ratio that seldom goes below 50%. As it is, the reserve ratio typically ranges between 15% and 30%. Alternatively, using more traders, ceterus paribus, and without any banking, i.e. the Baseline treatment, results in bubbles which are of lower intensity and which peak in the first or second period, as illustrated in Chapter 1. Not shown in Chapter 1 are experiments run with 500 traders, in which bubbles occur less frequently and always in the first or second period. This is the result of proportionately higher volume per period and a speculative strategy which is based on expected return between the current and next contract price (see Chapter 1 for details). However, it is certainly possible and a worthy future project to find a combination of strategies and parameters that generates bubbles at scales of 500 or more traders.

Table 4: Trading Parameters.

Parameter	**Description**	**Value**
NumTraders	Total number of traders	225
PercentMomentum	Percent of traders using a momentum strategy	25
PercentValue	Percent of traders using a value strategy	40
PercentSpeculative	Percent of traders using a speculative strategy	35
MaxSBR	Maximum speculative expected bull return	0.12
MaxSC	Maximum speculative correction	0.17
AL	Moving price average length	30
MaxLF	Maximum learning factor	0.2
VLF	Additional learning factor for value strategy	0.05
EndowClass1Cash	Cash endowment of Class 1 agents	E$2.25
EndowClass1Assets	Asset endowment of Class 1 Agents	3
EndowClass2Cash	Cash endowment of Class 2 agents	E$5.85
EndowClass2Assets	Asset endowment of Class 2 Agents	2
EndowClass3Cash	Cash endowment of Class 3 agents	E$9.45
EndowClass3Assets	Asset endowment of Class 3 Agents	1
Dividend1	First of four equally likely dividend possibilities.	E$0.0
Dividend2	Second of four equally likely dividend possibilities.	E$0.08
Dividend3	Third of four equally likely dividend possibilities.	E$0.28
Dividend4	Fourth of four equally likely dividend possibilities.	E$0.60
TP	Total number of trading periods.	15
TI	Total number of trading intervals per period.	10

Table 5: Banking Parameters.

Parameter	**Description**	**Value**
NumBankers	Total number of bankers	2
FreeBanking	Switch that allows banks to issue notes.	True or False
LoanSize	Size of both short and long term loans	E$5
InitialReserveRatio	The reserve ratio used by bankers in the initial periods until a sufficient sample of clearings is established. Also the ratio used to determine the portion of cash endowment which is initially borrowed.	0.5
ClearingSampleSize	Number of clearings used to calculate precautionary reserve demand.	10 under free banking, which is exactly 1 period, or 500 under central banking, which is typically less than 1 period.
Q	Banker conservatism, used to calculate precautionary reserve demand.	1 to 150. As shown in Figures 2 and 3, Q of 40 maximizes profits for bankers under central banking, while Q of 2 does so under free banking.
InitialSavingsRate	Initial interest rate paid by bankers on deposits.	2%
InitialShortLoanRate	Initial interest rate charged by bankers on short term loans.	4%
InitialLongLoanRate	Initial interest rate charged by bankers on long term loans.	6%
ShortLoanDuration	Duration of short term loan, in periods.	2
LongLoanDuration	Duration of long term loan, in periods.	4
SavingsRateElasticity	Savings rate elasticity of demand. Without some degree of inelasticity, all traders move their deposits to the bank with the highest rate, i.e. essentially a bank run.	0.7
WTPshortRate	Willingness of traders to pay interest for short term loans.	6%
WTPlongRate	Willingness of traders to pay interest for long term loans.	8%
WTAsavingsRate	Willingness of traders to accept interest for their deposits.	0.001%
iRateCF	Interest rate correction factor.	0.005
MinALR	Minimum asset-to-liability ratio for bankers to issue loans.	0.9

As mentioned earlier, *Q* is a critical banking parameter used by bankers to calculate precautionary demand for reserves and the optimal reserve ratio. Figures 18 and 19 illustrate how I optimized *Q* based on bank profit maximization subject to the requirement of no bankrupt banks within 20 experimental sessions. More precisely, I assumed industry and firm profits are zero over 20 sessions when there is a single bankruptcy. A less severe rule, such as one that makes a single bankruptcy result in zero firm and industry profits for that session only, would produce the same or a very similar outcome, i.e. a single-peaked curve of profit versus *Q* with the same peak profit at the same value of *Q*. Generally, as *Q* decreases bankruptcy becomes more likely. So, looking at Figure 2, with two central banks I found that a *Q* of 20 or less led to multiple bankruptcies within 20 experimental sessions. So my search approach was to run another set of experiments at double the highest bankrupt *Q*, which was at 40, where I found no bankruptcies in 20 sessions. Likewise for all higher *Q* values that were tried up to 150. Profits clearly decline with *Q*, and so the profits are maximized at *Q* of 40. The same approach was used for free banking, as shown in Figure 19. This process could certainly be improved upon by automating this iterative search and increasing the number of experiments and thus the precision, but for now it serves our purposes as it represents a consistent criterion across banking treatments.

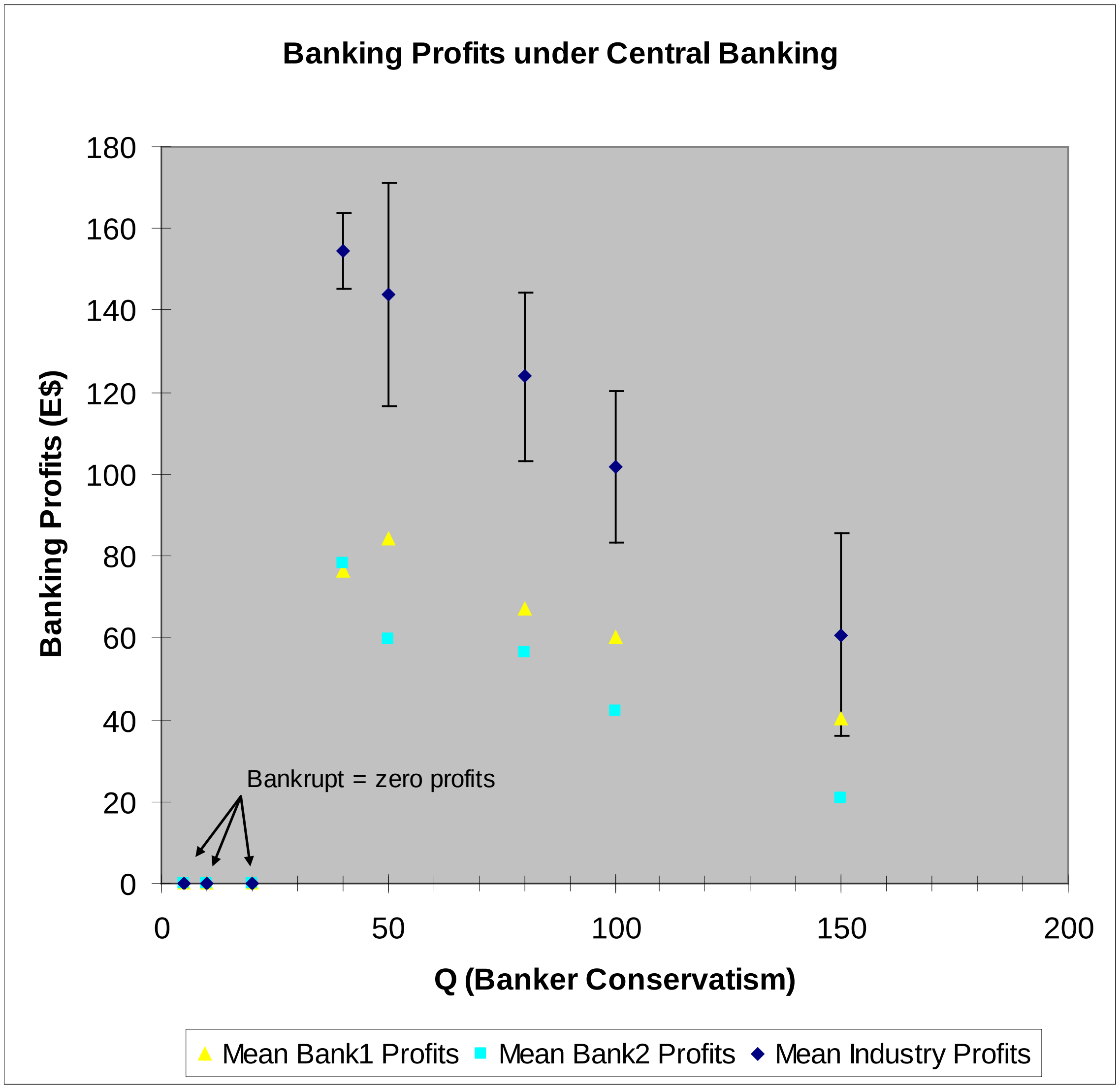


Figure 18: Mean banking profits over a range of *Q* values, under Central Banking. Industry and Bank2 profits are maximized at *Q* = 40, while Bank1 profits are maximized at *Q* = 50. There are 10 experiments at each *Q* value above 40, and up to 20 experiments at each *Q* value at or below 40. Bankruptcy is more likely as *Q* decreases, and a single bankruptcy in 20 experimental sessions amounts to zero mean profits at that *Q* value. The error bars represent 1 standard deviation around the mean of 10 experiments.

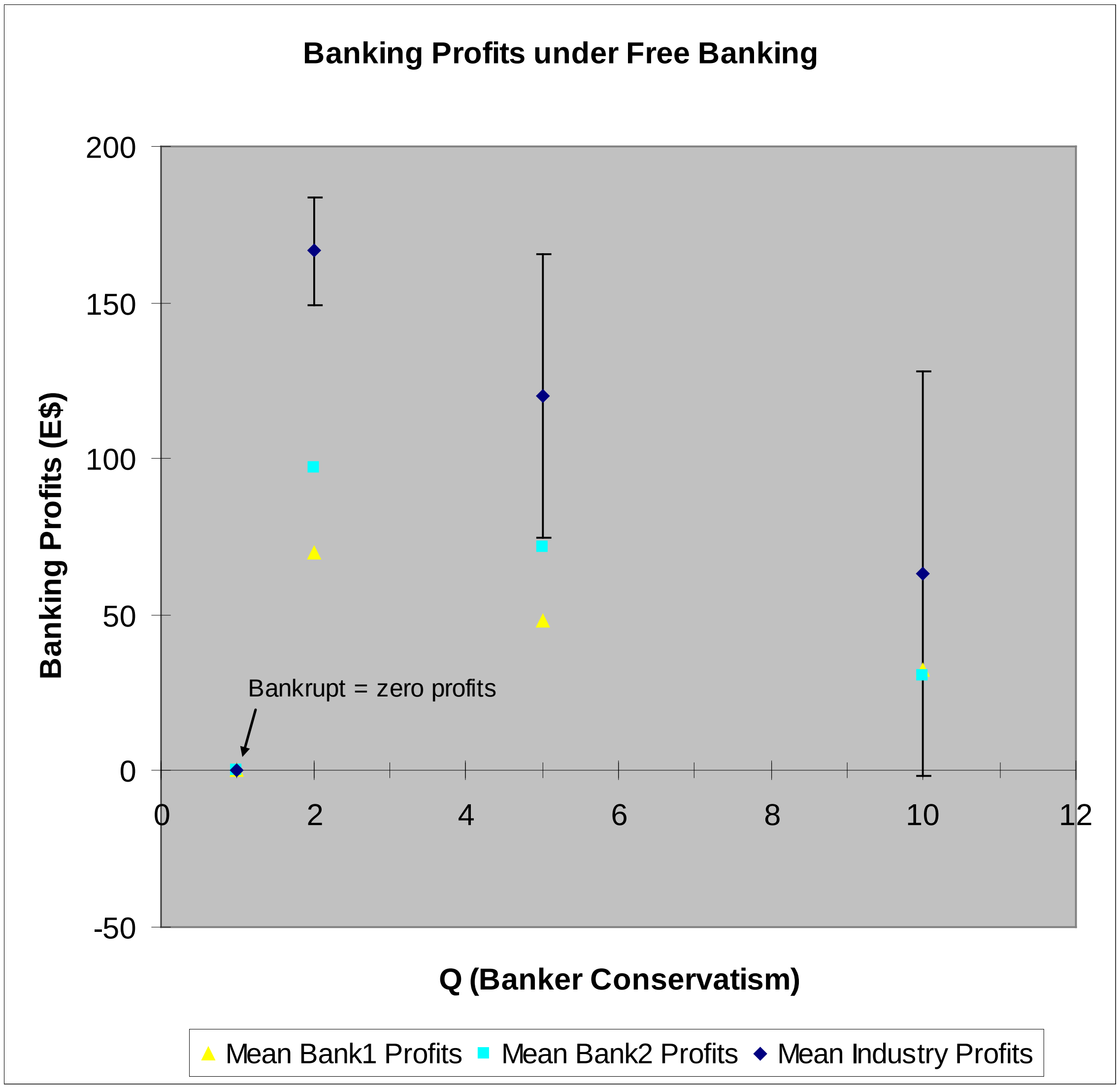


Figure 19: Mean banking profits over a range of *Q* values, under Free Banking. Industry and firm profits are maximized at *Q* = 2. There are 10 experiments at each *Q* value above 2, and up to 20 experiments at each *Q* value at or below 2. Bankruptcy is more likely as *Q* decreases, and a single bankruptcy in 20 experimental sessions amounts to zero mean profits at that *Q* value. The error bars represent 1 standard deviation around the mean of 10 experiments.

3.2 Discarded Parameters and Experiments

I should note a few of the more important parameters, parameter values, and experiments which were ultimately dropped from the final analysis, but contributed to the final design. First, I experimented with various ways to implement a demand for money. As Selgin (1988, p. 52) makes clear, the demand for money is rightly thought of as the demand to hold, and not spend, money, where money is either currency or deposits. In most of the early experiments, I implemented a demand for money using a constant optimal cash level, usually around \$5, which I reasoned was enough to buy a single asset. One problem with this design was that it created a substantial increase in the demand for money at the beginning of the experiment, when the optimal level of cash went from zero to \$5, and then demand for money fluctuated only slightly over the course of the experiment as the net buyers of assets were depleted of cash in the bubble up stage and vice versa when the net sellers of assets found themselves with excess cash in the crash stage. It also kept a good deal of cash out of the banking system, since all traders, both net buyers and net sellers, kept at least the optimal level of cash on hand. The solution, and current implementation, is to tie the optimal level of cash, and through that the demand for money, to trading activity, so that only those traders interested in buying and who lack sufficient cash increase their optimal cash level. Otherwise, it remains zero, thus maximizing the amount of money within the banking system and increasing the sensitivity of bankers to fluctuations in the demand for money.

For similar reasons, I experimented with ways to tie the willing to pay and accept interest to trading activity. However, I found that requires a significantly more sophisticated and complex set of assumptions regarding how each trader estimates his expected return in the asset market versus expected return in the money market. Ultimately, I decided to keep it simple by assuming a constant willingness to pay and accept interest, so that it operates similarly to a constant liquidity preference. This, however, is an area worth pursuing in future research.

In free banking, I experimented with the use of a limit on note issue based on deposits, i.e. free bankers were not allowed to issue notes in excess of some multiple of their deposits. I implemented it because it was a fairly common practice in historical free banking. I had found at smaller scales of traders, e.g. fewer than 225, without such an additional limit on note issue, free bankers were unstable and went bankrupt over a wide range of *Q* values, owing to the fact that their estimates of the standard deviation of clearings were based on small samples of clearings. The problem this introduced was that now the note issue limit was typically a more binding requirement than the optimal reserve ratio, and this left free bankers rather unresponsive to changes in velocity and the demand for money as measured by the volume of clearings. The solution was to keep increasing the number of traders until free banking without a note issue limit became sufficiently stable, in terms of reliable estimates of the standard deviation of clearings, and this occurred at around 225 traders.

Also, I did not select the number 225 randomly. I had originally planned to look at a range of competitive scenarios, ranging from 2 to 5 bankers. After some experimentation, I realized the scale and instability issues meant at most I could compare the 2 banker scenario to the 3 banker scenario. Specifically, the design was to keep the ratio of traders to bankers constant across treatments and look at the effects of more competition in banking. Thus, I ran experiments with 2 bankers and 100 traders and compared that to experiments with 3 bankers and 150 traders. Next, I ran experiments with 2 bankers and 150 traders and compared that to experiments with 3 bankers and 225 traders. The plan was to continue this pattern up to 338 traders and then at 507 traders. However, I eventually determined that even at these larger scales of traders, the instability issues meant that no more than 2 bankers could be used. So as to properly implement such an investigation of banking competiveness, among other reasons as mentioned earlier, it remains a worth while future project to find the right combination of strategies and parameters that generates bubbles in later periods. Along the same lines, it would be worth while to investigate the effect of a more heterogeneous banking industry, in terms of the values of *Q* as well as initial interest rates and reserve ratios.

Further, I had also intended to investigate the effects of various initial interest rates and interest rate spreads, and thus profits for bankers. As mentioned in the discussion of the model (section 3.0), the choice of 2% for the initial savings rate and 4% for the initial short term loan rate was driven by the need to keep the initial money supply constant across treatments, given an initial reserve ratio of 50%. However, this could also be accomplished by, for example, halving the initial interest rates to 1% for savings and 2% for short term loans. Time constraints did not permit such experiments, and it remains a worth while future project. However, I did run quite a few, hundreds in fact, of experiments at smaller scales of traders with an initial savings rate of 1% and a short term loan rate of 5% or 6% (and with a long term loan rate of 8% or 9%). I found that, at such smaller scales of 12 and 100 traders, central and especially free banking were still largely unprofitable and prone to bankruptcy even with this initial interest rate spread, creating much of the impetus to scale up to 225 traders. I also found that bubbles were greatly reduced, even eliminated under many parameter configurations. It became evident that this was merely a liquidity effect, as will be discussed in the results section, and thus it was important to maintain a constant initial money supply across treatments.

Lastly, I also experimented with having a third family of agents which do not trade but do bank. The purpose was to maintain the same scale of traders as found in the laboratory (12), or at least close to that, while achieving a large number of bank transactions. I did successfully implement this design in software, however, the results proved rather useless except to further show that the banking theory in question could only be tested through maximal interaction between bankers and traders, i.e. not non-traders.

4.0 Results and Discussion

Table 6 shows all experiments used to compare bubble tendency. Every measure of bubble tendency supports the hypothesis, i.e. bubbles are most prevalent, in terms of both size and frequency, under the Baseline (no banking) treatment, less prevalent under

Central Banking, and least prevalent under Free Banking. The first and second measures, in columns 3 and 4, are measures of frequency, while the third and fourth measures, in columns 5 and 6, are measures of size as well as frequency. Looking at the fourth measure, in column 6, i.e. the mean percent bubble contracts, both the Central and Free Banking treatments have significantly fewer bubble contracts than the Baseline treatment. However, the Central and Free Banking treatments are not significantly different from each other.

Also, it appears that much of the difference between the banking and non-banking treatments results from differences in trading volume, as shown in the last column. Banking results in roughly 65% more trading volume, as compared to non-banking. Furthermore, the increased volume is largely in the middle periods, as can be seen from Figures 20, 21, 22, and 23 indicating that banking eases the cash-in-hand constraint on trading.

The monetary environment of these experiments, as was mentioned in the introduction and illustrated in Chapter 1, consists not of a constant stock of base money but rather, as dictated by the original laboratory set up (Smith et al, 1988), consists of the cash endowment plus a positive random asset dividend payment in each period, i.e. a monotonically increasing money supply. Chapter 1 shows there is a strong correlation between the size of these dividend payments and the tendency to bubble. This can be seen in Figure 20 as well, where asset prices and the money supply are plotted for a sample of 10 experiments with 225 traders without banking. Particularly, notice in Figure 20 that in every experiment, where there is a spike in the mean price deviation there is also a spike in the money supply. As discussed in Chapter 1, these spikes in the money supply are due to high dividend draws ($0.60). Thus, it is not surprising that both free and central banking fail to completely eliminate bubbles. Rather these results, by showing some reduction in bubble tendency under free banking, lend support to the theory of free banking even in a non-ideal and challenging monetary environment. Figures 21 and 22, which show the money supply and asset prices for a sample of central and free banking experiments, as well as Figure 23, which shows a side-by-side comparison of the three treatments by averaging across 10 experiments, indicate that banking tends to absorb the influx of base money, reducing the money supply in later periods and thus decreasing the tendency to bubble. There is no obvious distinction between free and central banking in terms of the mean money supply per period, which indicates that any price stability advantages of free banking are occurring at either the trading interval level or the transaction level.

The result that free banking limits bubbles by limiting money supply represents a novel demonstration of the principle of adverse clearings, as described by Selgin (1988, pp. 37 – 42), whereby free banks are limited in their note issue by the total stock of base money and the need to settle in base money with other banks any outstanding banknote balance. As Selgin explains, the principle of adverse clearings, though controversial, is merely a generalization of the more commonly accepted rule of excess reserves, whereby in a system with monopolized currency supply, an individual bank is limited in the loans and investments it may issue by the fact that total reserves are finite and it must settle in base money all currency demands and clearing balances owed to other banks. Thus, my results, showing that both kinds of banking reduce bubbles by limiting money supply, provide support for both the principle of adverse clearings and the rule of excess reserves.

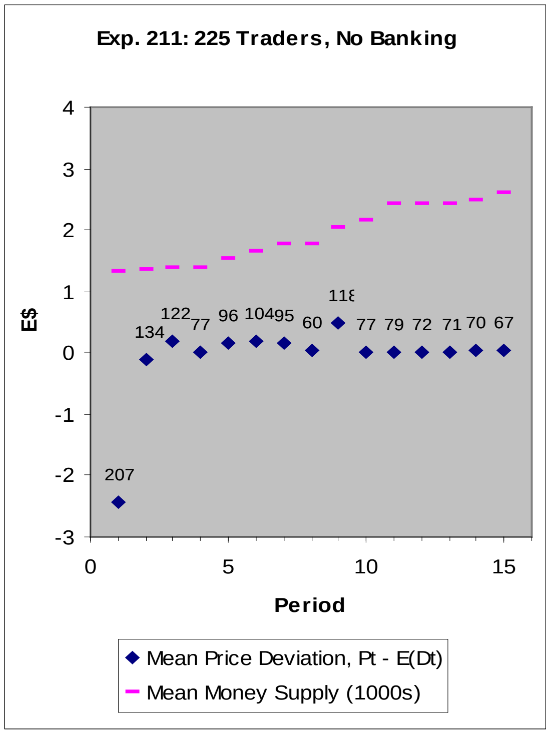
Also, it should be noted that while the *Q* values used to generate this data were selected based on profit maximization, Figures 18 and 19 indicate there is a range of profitable *Q* values. I found that central banking almost completely eliminated all bubble contracts at higher values of *Q*, while free banking did not. It is not clear why this would be. However, I found that free banking performed relatively better, in terms of bubble reduction, at larger trading scales. For example, at the scale of 100 traders I found that free banking reduced bubbles, but not profitably, while central banking did so profitably. Further, free banking performance improved at the scale of 150 traders and again at 225 traders. This indicates that free banking might continue to improve at larger scales, say 500 traders. However, as mentioned earlier, it becomes much more challenging at these larger scales to generate bubbles beyond the first or second periods, but it remains a project worth pursuing.

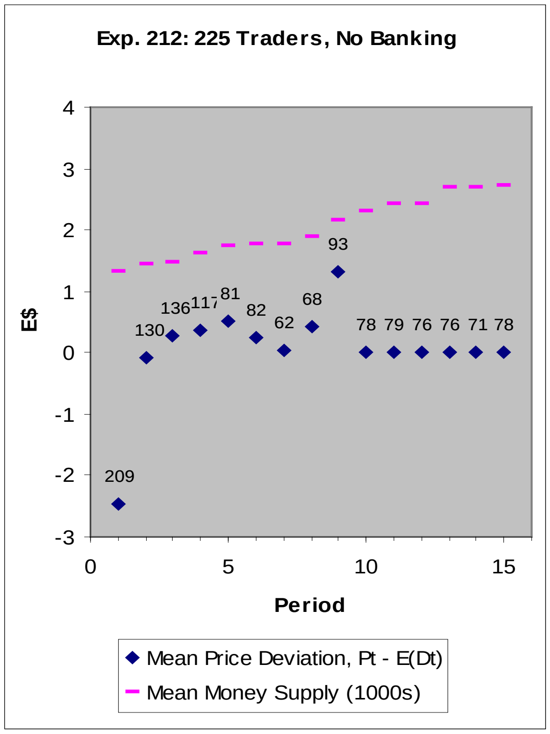
Lastly, while I found no significant distinction between central and free banking in terms of bank stability, nor was it the focus of this dissertation. In fact, the possibility was essentially designed away as part of the process of selecting *Q*, i.e. *Q* was selected by maximizing profits subject to no bankruptcies within 20 experimental sessions. However, using the parameter *SavingsRateElasticity*, it is possible to increase the incidence of deposit transfers and thus bank runs. In future work it would be interesting to observe how central and free banking compare under such destabilizing conditions.

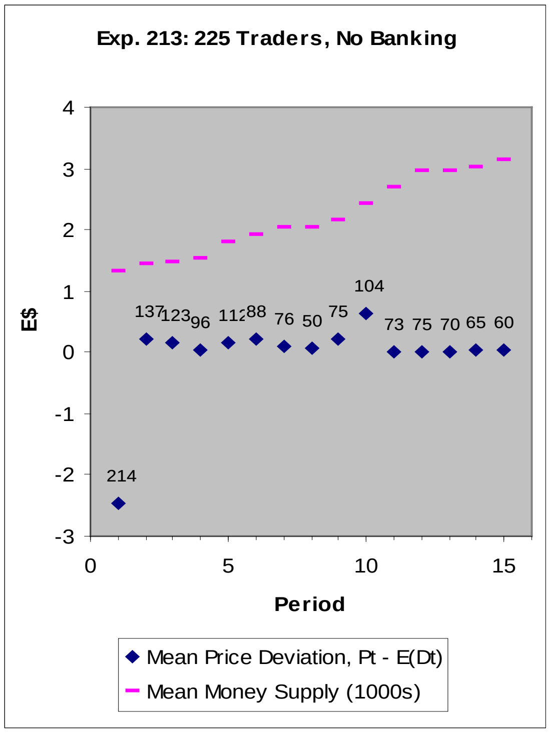
My conjecture is that free banking would perform better, since the right to issue notes somewhat cushions free banks from such deposit withdrawals.

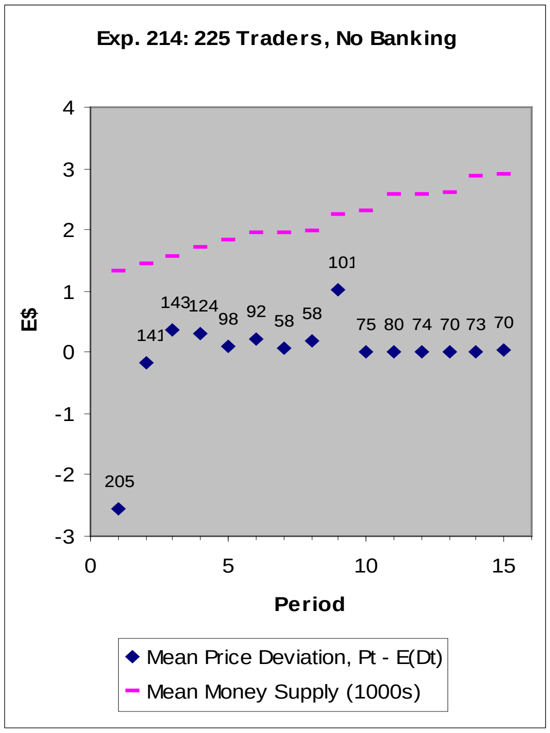
Table 6: List of all experiments with 225 traders used to compare bubble tendency. Bubble tendency is measured using a) the percent of experiments with a mean (over contracts in a period) price bubble, where a bubble is defined as a price deviation exceeding 1.0, b) percent with any price bubble contracts, c) the mean (over experiments) number of bubble contracts, and d) the mean (over experiments) percent of contracts which bubble. P values are shown directly below the measure in question, and are calculated from two-tailed T tests.

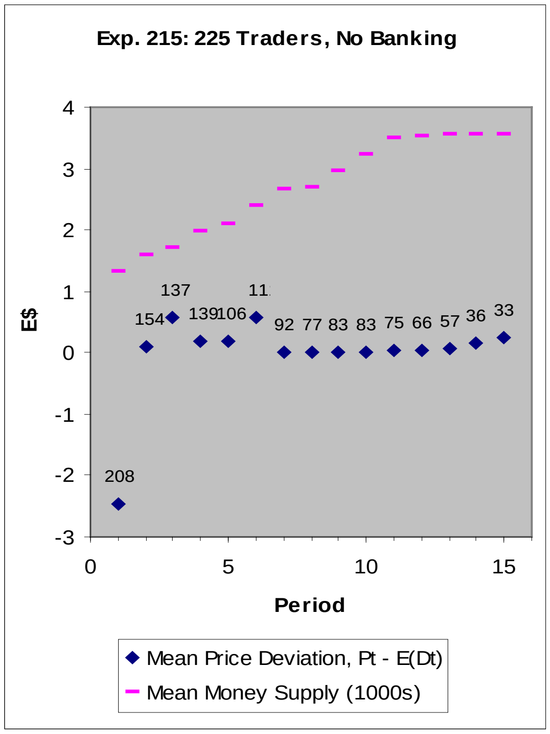
Treatment	**Experiments**	**Percent with a Mean (over period) Price Bubble**	**Percent with Any Bubble Contracts**	**Mean (over experiments) Number of Bubble Contracts**	**Mean (over experiments) Percent Bubble Contracts**	**Mean Trading Volume**
Baseline (No Banking)	20	30	95	52.50	3.76	1412.75
Central Banking	20	0	85	37.00	1.62	2322.15
Free Banking	20	0	60	27.25	1.15	2348.3
P values						
No Banking versus Central Banking				0.30	0.04	
No Banking versus Free Banking				0.09	0.01	
Central versus Free Banking				0.37	0.31	

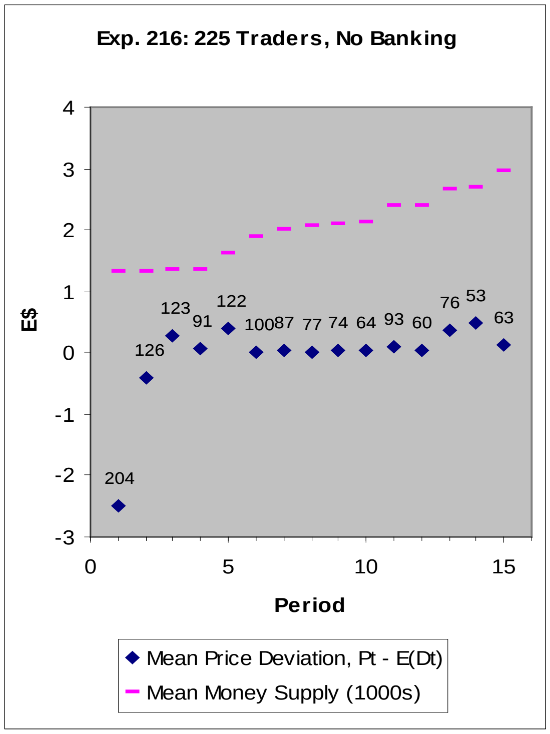

Exp. 211: 225 Traders, No Banking
E$
Period
207
134
122
77
96
104
95
60
118
77
79
72
71
70
67
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)

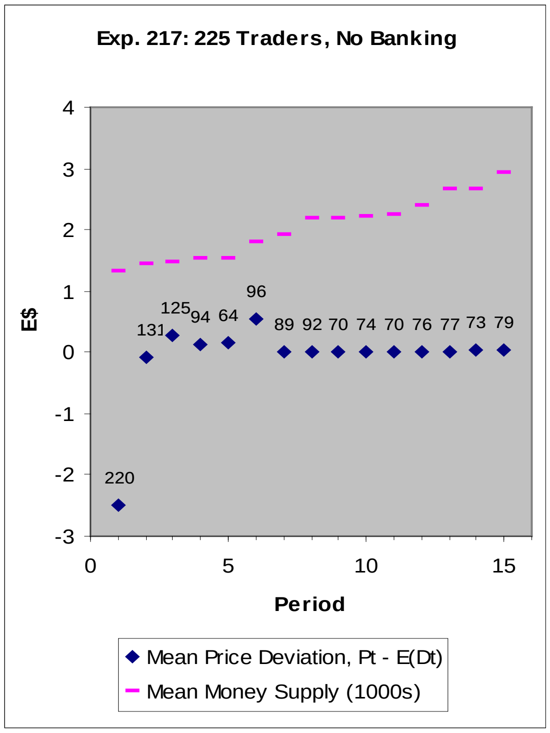

Exp. 212: 225 Traders, No Banking
E$
Period
209
130
136
117
81
82
62
68
93
78
79
76
76
71
78
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)

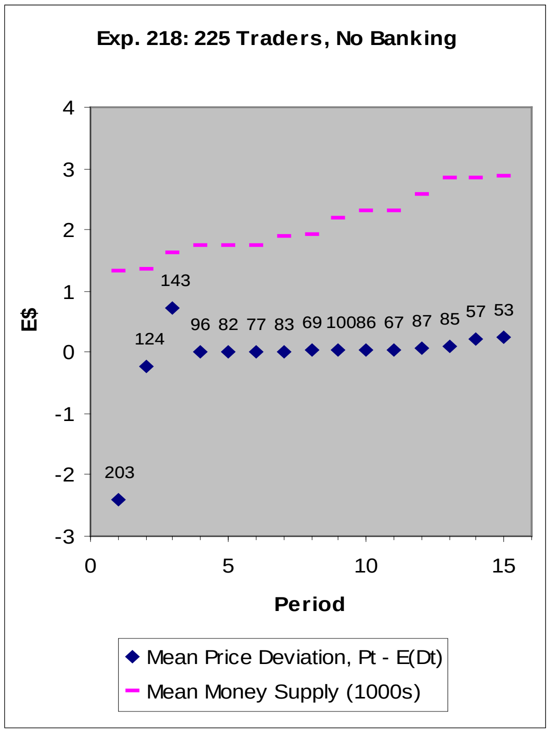

Exp. 213: 225 Traders, No Banking
E$
Period
214
137
123
96
112
88
76
50
75
104
73
75
70
65
60
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)


Exp. 214: 225 Traders, No Banking
E$
Period
205
141
143
124
98
92
58
58
101
75
80
74
70
73
70
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)


Exp. 215: 225 Traders, No Banking
E$
Period
208
154
137
139
106
11
92 77 83 83 75 66 57 36 33
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)


Exp. 216: 225 Traders, No Banking
E$
Period
204
126
123
91
122
100 87 77 74 64 93 60
76
53
63
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)


Exp. 217: 225 Traders, No Banking
E$
Period
220
131
125
94 64
96
89 92 70 74 70 76 77 73 79
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)


Exp. 218: 225 Traders, No Banking
E$
Period
203
124
143
96 82 77 83 69 100 86 67 87 85 57 53
◆ Mean Price Deviation, Pt - E(Dt)
▬ Mean Money Supply (1000s)

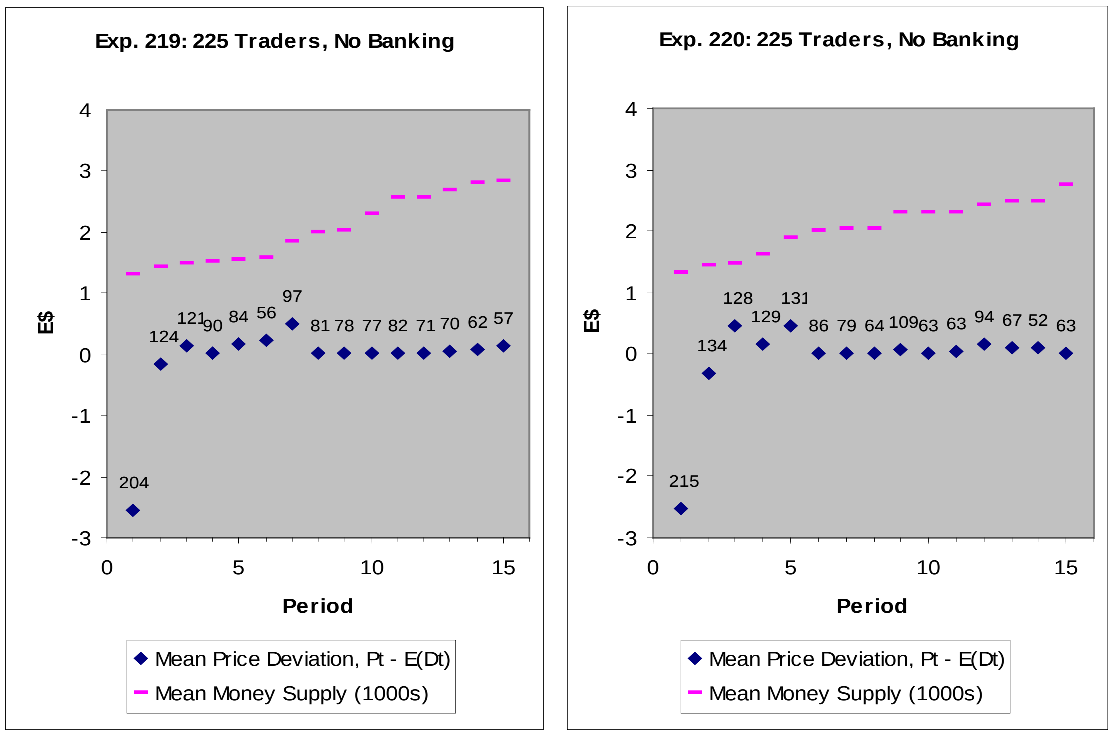
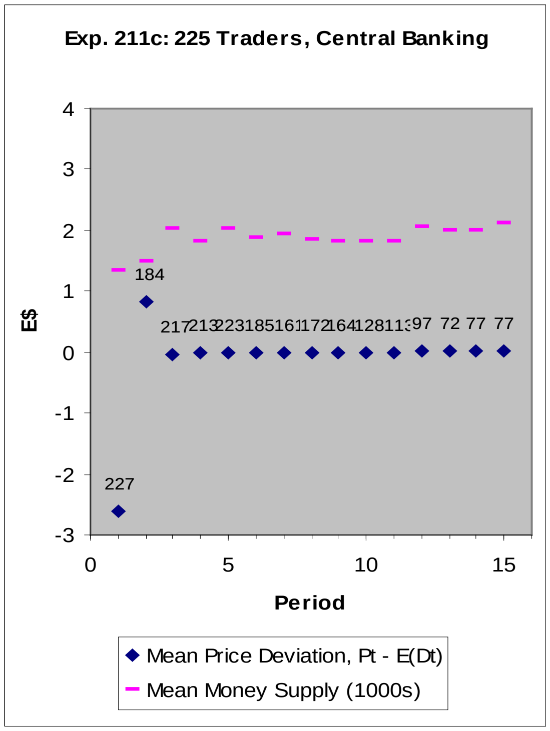
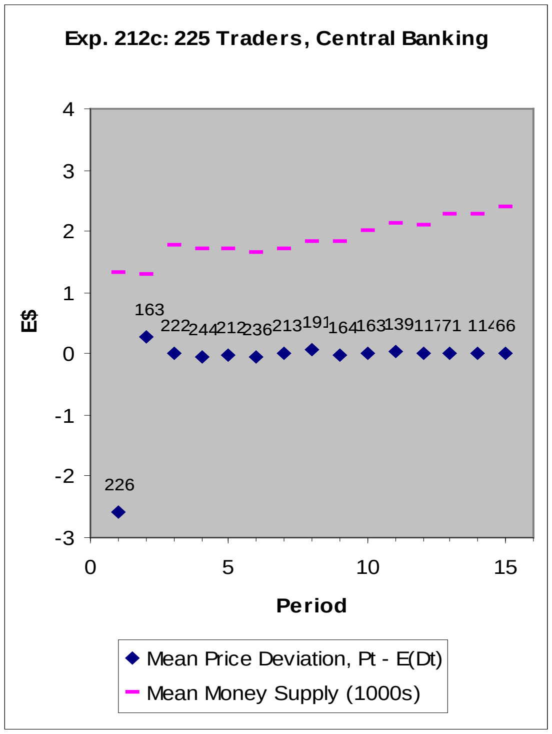
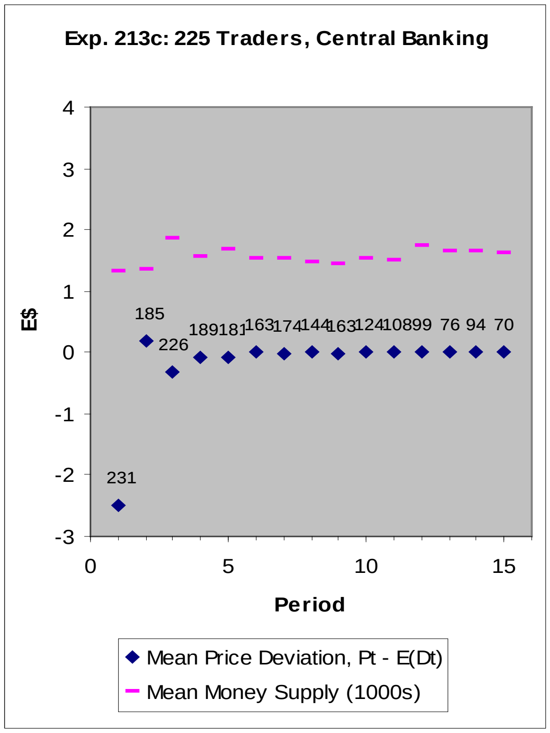
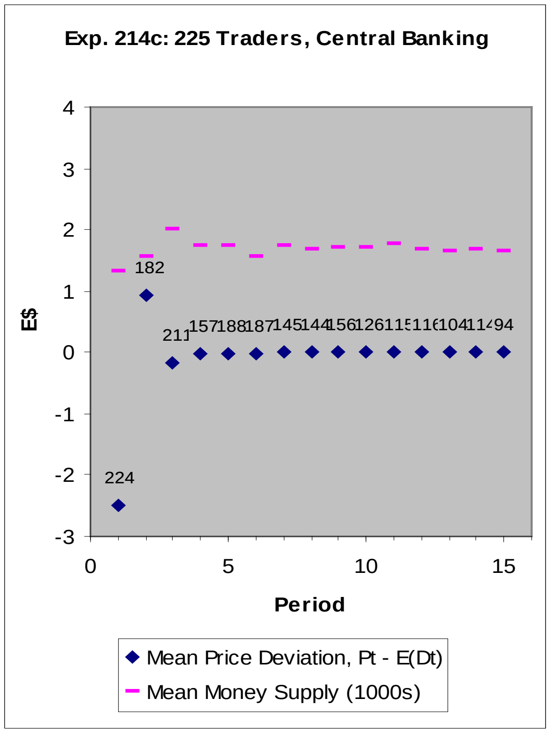


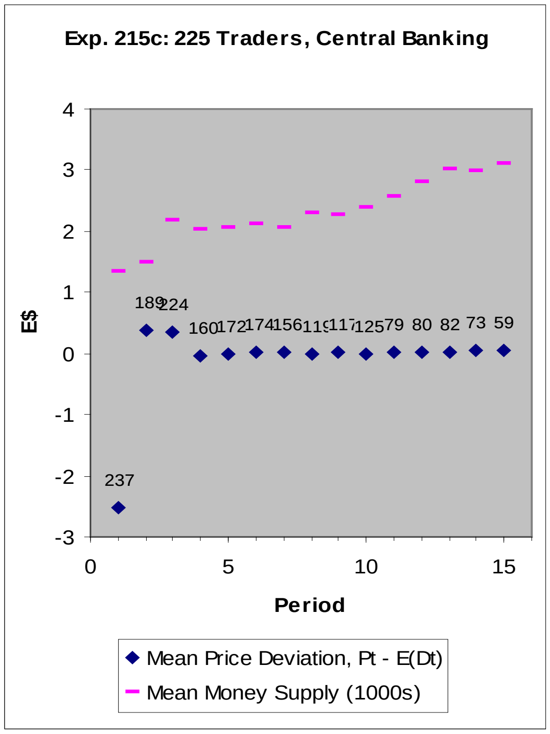
Figure 20: 10 of 20 experiments with 225 traders and no banking. It shows the money supply, the mean price deviation, and, above that, the volume in each period.

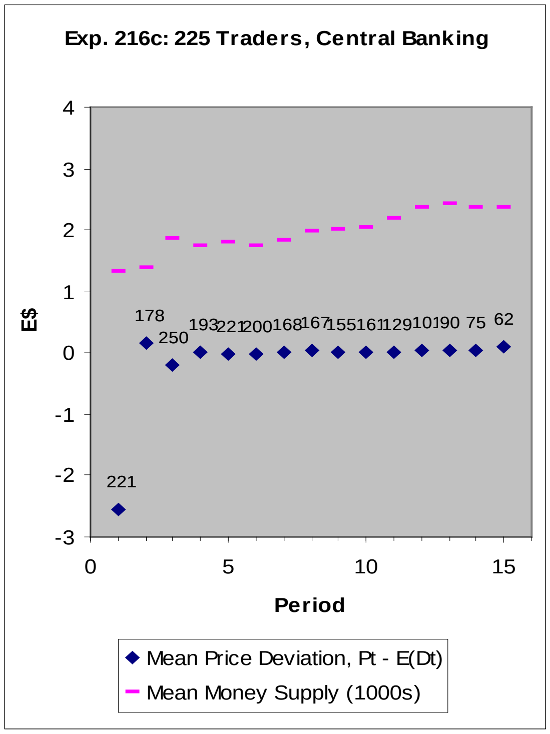

Exp. 211c: 225 Traders, Central Banking
E$
Period
227
184
217 213 223 185 161 172 164 128 113 97 72 77 77
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)

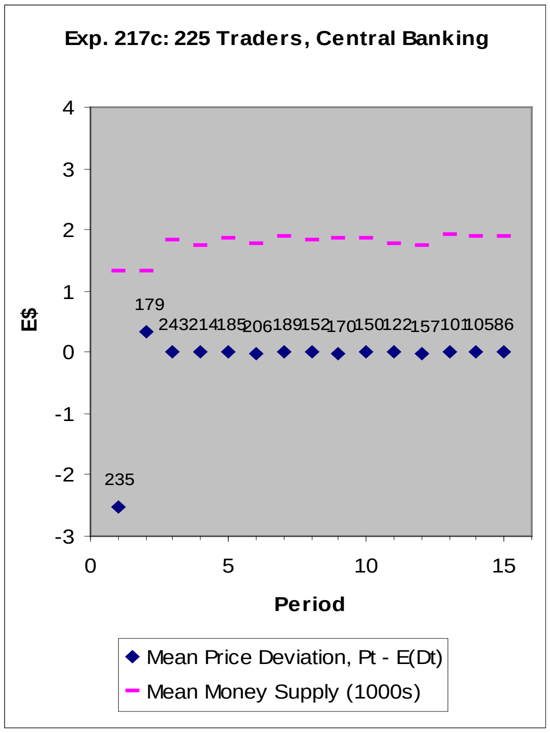

Exp. 212c: 225 Traders, Central Banking
E$
Period
226
163
222 244 212 236 213 191 164 163 139 117 71 114 66
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)

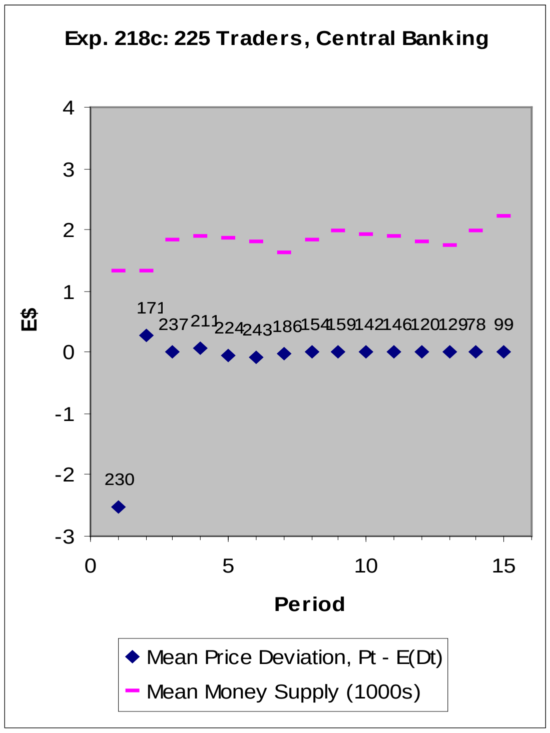

Exp. 213c: 225 Traders, Central Banking
E$
Period
231
185
226
189 181 163 174 144 163 124 108 99 76 94 70
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 214c: 225 Traders, Central Banking
E$
Period
224
182
211
157 188 187 145 144 156 126 115 116 104 114 94
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 215c: 225 Traders, Central Banking
E$
Period
237
189
224
160 172 174 156 119 117 125 79 80 82 73 59
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 216c: 225 Traders, Central Banking
E$
Period
221
178
250
193 221 200 168 167 155 161 129 101 90 75 62
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 217c: 225 Traders, Central Banking
E$
Period
235
179
243 214 185 206 189 152 170 150 122 157 101 105 86
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 218c: 225 Traders, Central Banking
E$
Period
230
171
237 211 224 243 186 154 159 142 146 120 129 78 99
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)

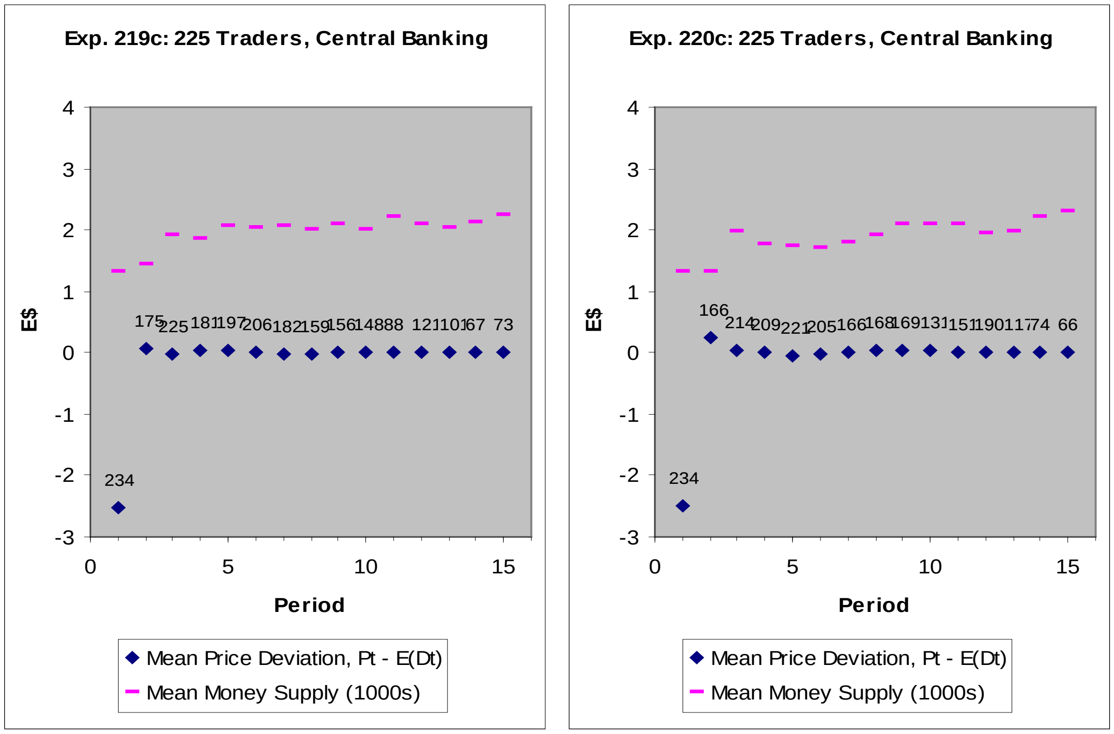
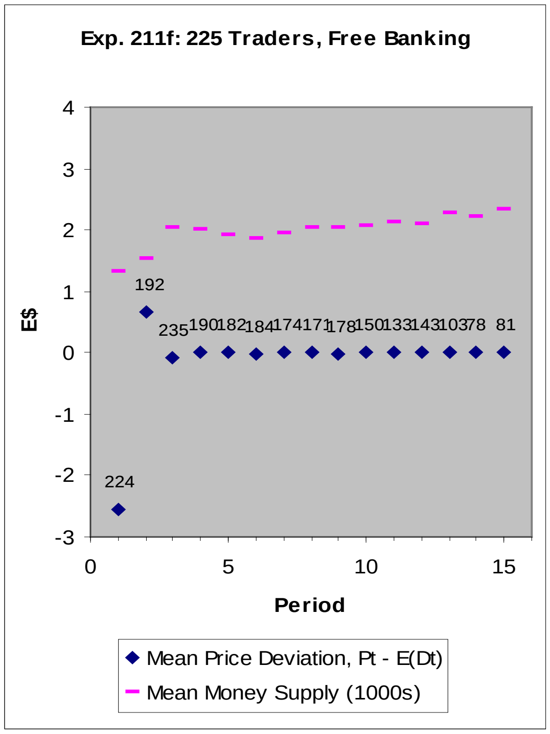
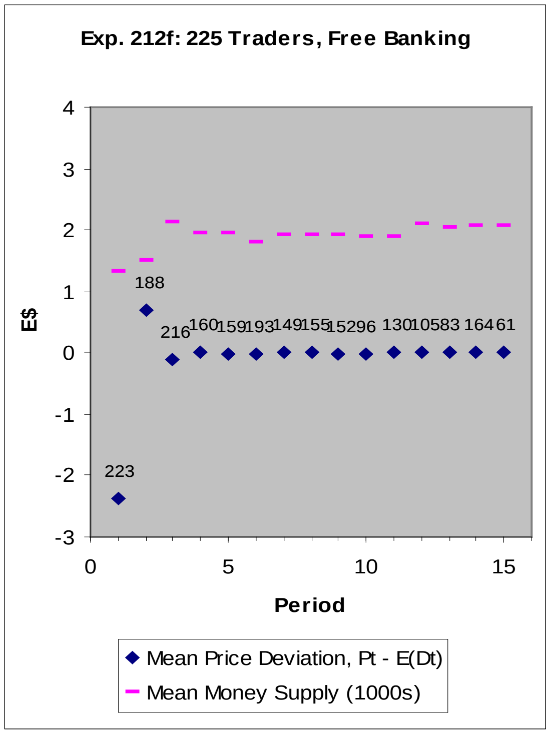


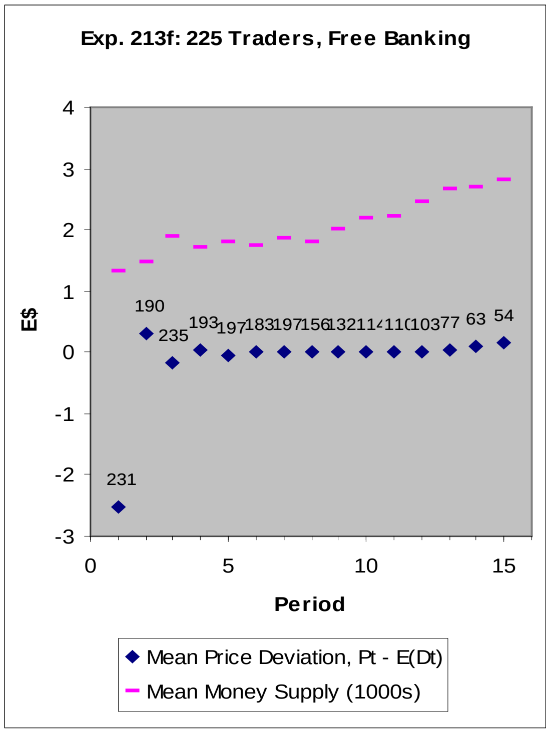
Figure 21: 10 of 20 experiments with 225 traders and central banking. It shows the money supply, the mean price deviation, and, above that, the volume in each period.



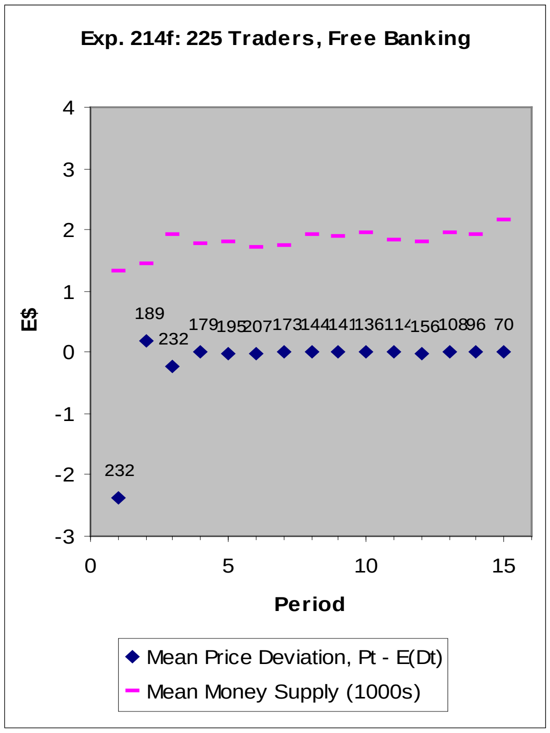
Exp. 211f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
224
192
235
190
182
184
174
171
178
150
133
143
103
78
81
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)



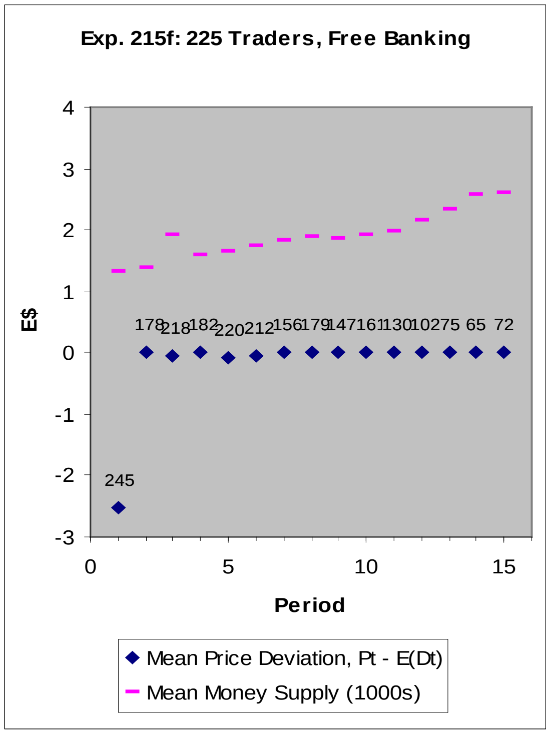
Exp. 212f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
223
188
216
160
159
193
149
155
152
96
130
105
83
164
61
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)

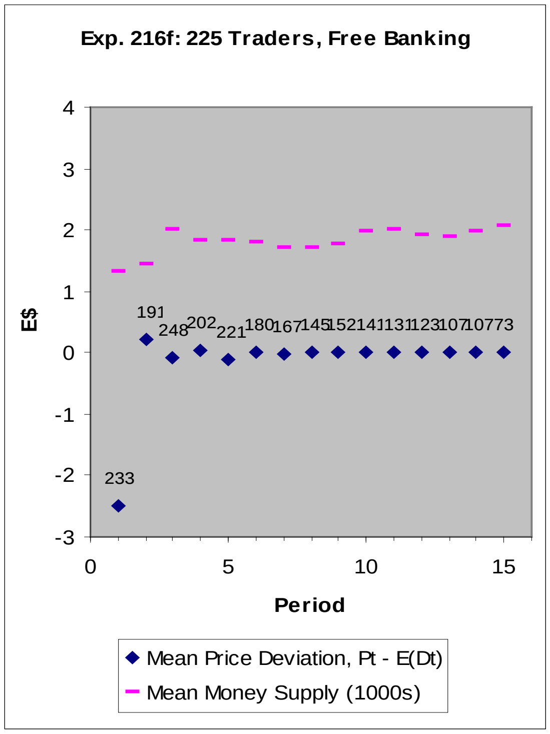


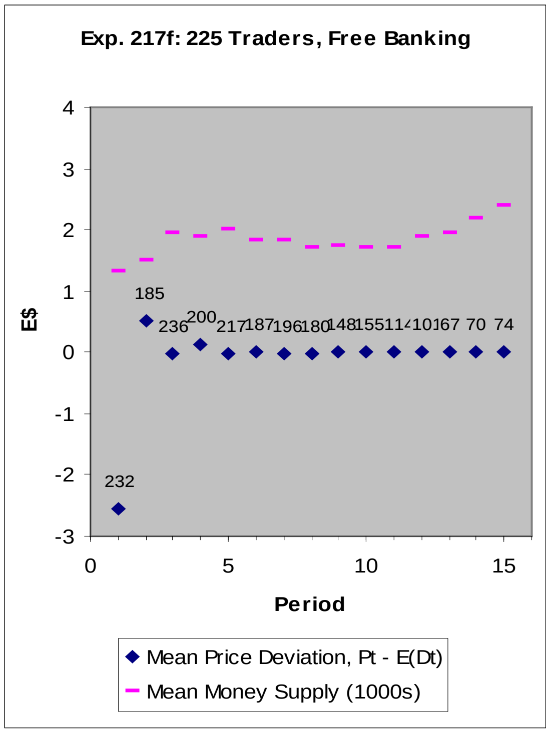
Exp. 213f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
231
190
235
193
197
183
197
156
132
114
110
103
77
63
54
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)

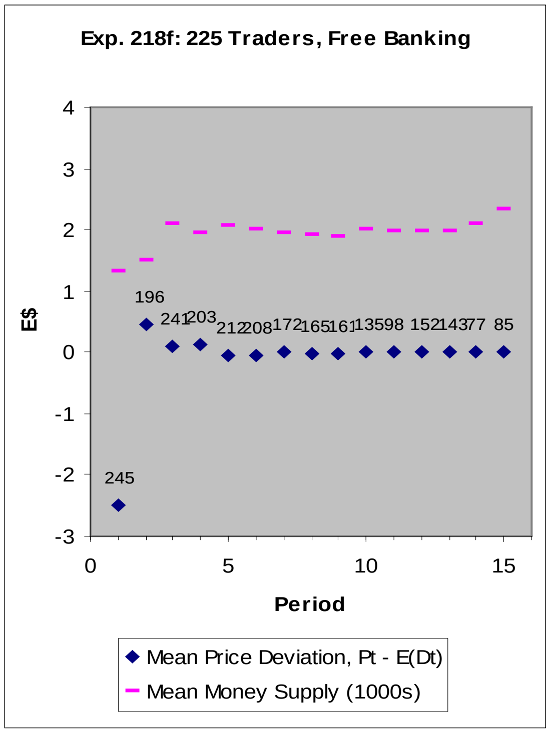


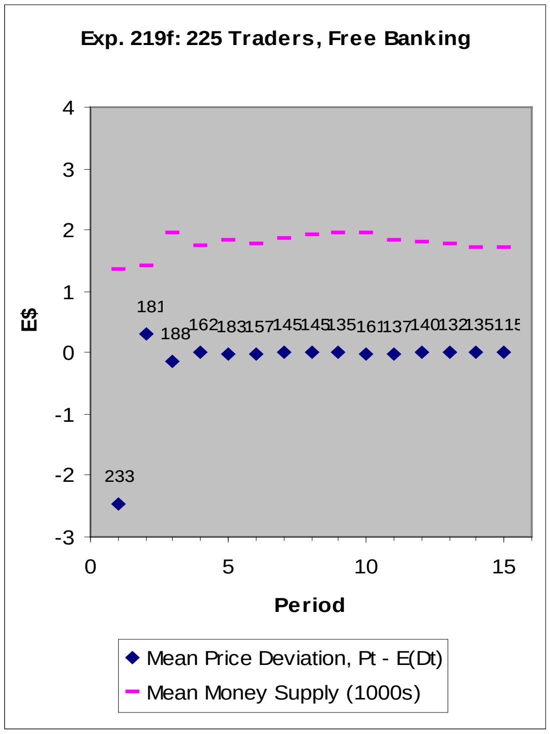
Exp. 214f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
232
189
232
179
195
207
173
144
141
136
114
156
108
96
70
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 215f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
178 218 182 220 212 156 179 147 161 130 102 75 65 72
245
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 216f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
191 248 202 221 180 167 145 152 141 131 123 107 107 73
233
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 217f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
185 236 200 217 187 196 180 148 155 114 101 67 70 74
232
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)


Exp. 218f: 225 Traders, Free Banking
E$
4
3
2
1
0
-1
-2
-3
0
5
10
15
Period
196 241 203 212 208 172 165 161 135 98 152 143 77 85
245
Mean Price Deviation, Pt - E(Dt)
Mean Money Supply (1000s)



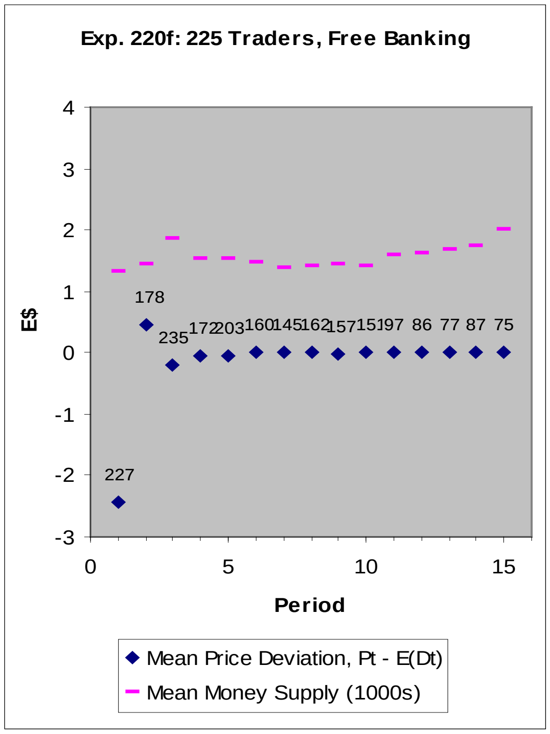


Figure 22: 10 of 20 experiments with 225 traders and free banking. It shows the money supply, the mean price deviation, and, above that, the volume in each period.

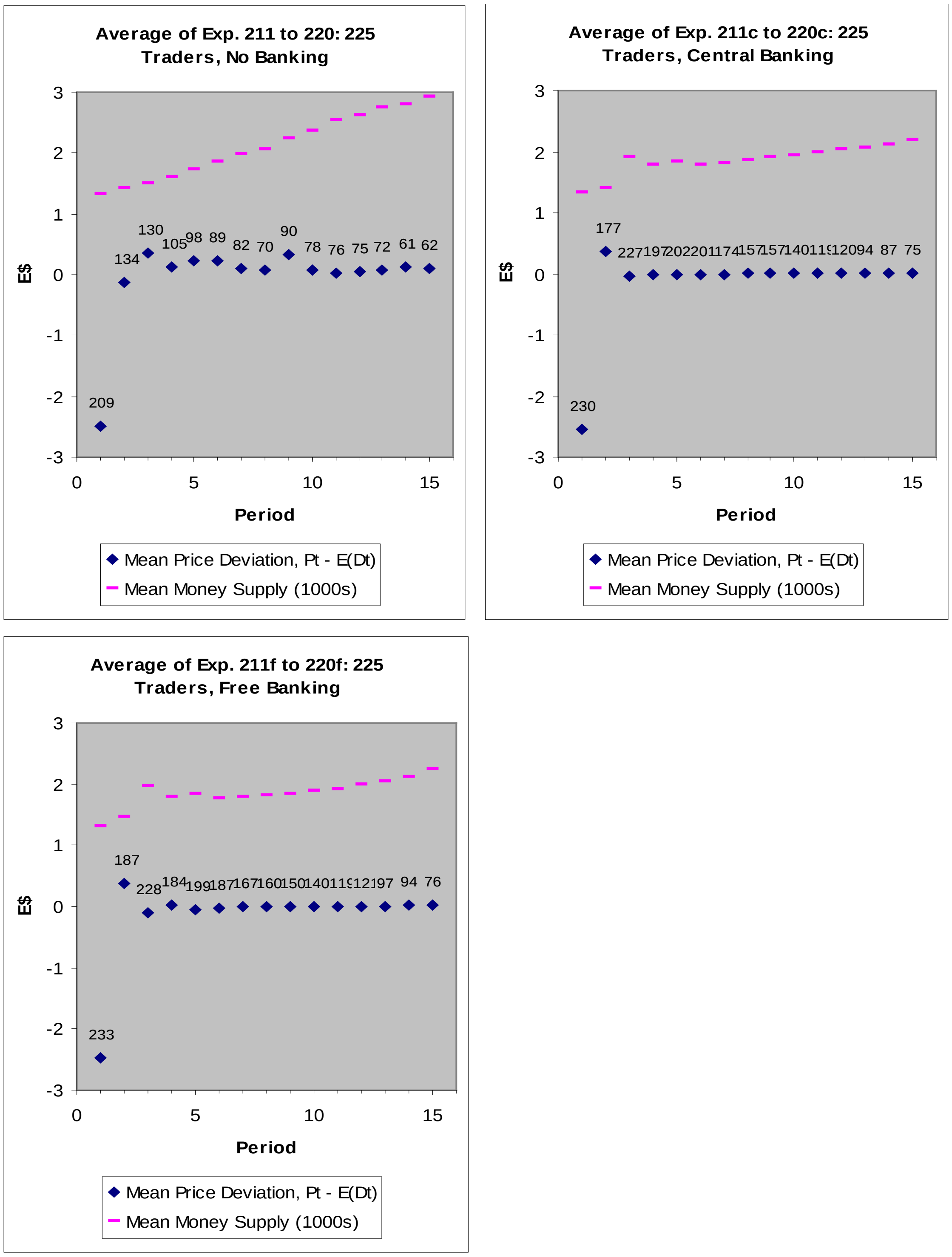


Figure 23: Treatment average of 10 of 20 experiments with 225 traders. It shows the money supply, the mean price deviation, and, above that, the volume in each period.

5.0 Conclusion

In summary, this chapter describes an agent-based model and simulation of banking which builds upon the agent-based model of asset trading developed in Chapter 1. The purpose is to test the macroeconomic price stability qualities of free banking. As compared to the canonical laboratory asset market as originally designed by Smith et al (1988), I find that both free and central banking reduce the tendency to bubble, and that free banking is most effective in this regard. Further, both kinds of banking accomplish this within a monetary environment of increasing money supply, as illustrated in Figure 20, and they do so by absorbing the influx of money, as illustrated in Figure 23. Thus, despite a challenging monetary environment, these results provide support for the theory of free banking, particularly the principle of adverse clearings as articulated by Selgin (1988, pp. 37 – 42), whereby free banks are limited in their note issue by the total stock of base money and the need to settle in base money with other banks any outstanding banknote balance.

However, I would stress that the external validity of these simulations is entirely dependent on the validity of my assumptions about the model and the parameters used. While I have gone to great lengths to justify my assumptions, these few simulations are still a poor substitute for the sort of data that would occur in the field, where banks would potentially have 1000s of customers and transactions per day under an array of money supply conditions. Unfortunately, we have no such field data beyond pre-Depression era historical records, since free banking remains illegal in every developed economy. As a result, agent-based simulations and laboratory experiments are a promising avenue of research, one which brings the scientific power of controlled experiments to long standing debates on the macroeconomic effects of banking.

CHAPTER 3: DESIGN FOR A FREE BANKING LABORATORY EXPERIMENT

1.0 Introduction

The ongoing financial crisis and resulting government intervention has called into question the wisdom of our current banking regime. Some reform proposals would expand the regulatory powers of the Federal Reserve, while others would curtail or even repeal those powers. It is worthwhile then to consider a world without central banking, a world which has not existed in practice since at least the Great Depression. Before this time and throughout much of the 18th and 19th centuries, many countries, such as Canada, Australia, and Scotland, practiced so called "free banking" or competitive note issue without control by a central bank (see White, 1995, and Dowd, 1992). Free banks enjoyed important freedoms, particularly the right to issue their own notes, contractually redeemable in a market determined monetary base (typically specie), and without any statutory reserve requirements.

Modern day proponents of free banking, including George Selgin (1988, pp. 52-85, 1994) and Lawrence H. White (1995, pp. 12-17, 107-115), argue it is capable of limiting fluctuations in the general price level by making for a more stable relationship between the volume of base money present in an economy and corresponding levels of credit and aggregate spending.

A separate literature, the experimental asset bubble research stemming from Vernon Smith et al. (1988), appears to support the notion that unsustainable booms are behavioral in origin - the result of Keynesian “animal spirits” and as such an unavoidable aspect of unfettered free markets. To be more precise, a bubble is an extended period of time during which an asset trades above fundamental value, which outside the lab typically can only be known with any degree of certainty ex post when the bubble has burst. Inside the lab, fundamental value can be induced and thus known with complete certainty ex ante (Smith, 1976). Smith et al. (1988) and numerous experimenters since have found that, even when subjects are informed ex ante of fundamental value, asset bubbles occur frequently and are robust to various treatment conditions. Subject experience tends to reduce bubbles, but this result is sensitive to changes in treatment conditions. Hussam et al. (2008) find that bubbles can be rekindled among experienced subjects by changing either the asset dividend structure or subject endowments.

The experimental asset bubble literature is largely orthogonal to the free banking literature. The free bankers’ theory of money supply does not allow for animal spirits and resulting asset bubbles. Vernon Smith’s experiments, on the other hand, do not include a well-defined monetary environment, and so do not indicate the extent to which the likelihood of asset bubbles may depend on the nature of banking and monetary arrangements.

In this chapter, I extend the experimental asset bubble research by incorporating specific banking arrangements into Smith's basic design. There are three main treatments:

1) Baseline, which is a replication of the double auction asset trading environment of Smith et al (1988), scaled up from 12 to 220 traders using simulated agents. That is, there are about 20 human subjects acting as traders, and each one has 10 simulated agents who mimic their behavior with some noise. The purpose of the scaling is to generate a sufficiently large number of transactions so that bankers have an approximately continuous distribution of clearings by which to judge their optimal precautionary reserves (see Selgin, 1988, pp. 73-78). The stock of money, which is in experimental dollars, is not constant but rather, as dictated by the original laboratory set up (Smith et al., 1988), consists of the cash endowment plus a random asset dividend payment in each period.

2) Central Banking, wherein two additional simulated agents are added as bankers, and they have all the significant rights and functionalities of bankers except the right to issue notes. Mainly, bankers accept deposits from traders and then lend some portion of that back out to traders at competitively determined interest rates. In other words, this is a system of free banking in deposits, where neither bank has the right to issue notes. There is no representation of the "central bank" beyond the restriction on note issue, but one can assume that the stock of *base* money, which is in experimental dollars, is created exogenously by a central bank. As in the Baseline Treatment, base money enters the system in the form of cash endowments and also per period dividend payments, and the base money is also the currency. The monetary environment is

different in two ways: a) each trader's money endowment consists of a deposit and a loan with one of the bankers, in such a way that the total money supply, i.e. the total of traders' cash holdings and deposits, is constant initially across treatments, and b) changes thereafter in the total money supply are determined partly by dividend payments but also by the amount of credit issued by the two bankers according to each banker's calculation of their demand for precautionary reserves.

3) Free Banking, which also allows the two bankers to issue notes redeemable in experimental dollars. This is a mature free banking economy, so that bank notes have come to fully replace experimental dollars as the currency, while the stock of experimental dollars represents the stock of bank reserves as well as base money. This is implemented by giving bankers a preference for issuing banknotes and traders a preference for using banknotes as currency. Each banker brings the other banker's notes to the clearinghouse on a regular basis, where they are cleared without cost or discounting. As will be discussed further in the agent-based model section, the settlement process is somewhat different between the two banking treatments. Under central banking, all transactions are in base money including deposits, withdrawals, and loans, in what might be called real-time gross settlement. Under free banking, all transactions, except those involving endowments and dividend payments, are in banknotes and traders have the right, though as mentioned not the preference, to redeem banknotes in base money. On a regular basis, the two free bankers settle each others banknote balances at the clearinghouse using base money, in what might be called end-

of-day net settlement.[7]

My hypothesis, based on the theory of free banking as reviewed in the following section, is that both banking treatments will tend to reduce the frequency and size of asset bubbles as compared to the baseline treatment, and the free banking treatment will be the most effective in this regard. Particularly, the principle of adverse clearings, as described by Selgin (1988, pp. 37 – 42), states that free banks are limited in their note issue by the total stock of base money and the need to settle in base money with other banks any outstanding banknote balance. As Selgin explains, the principle of adverse clearings, though controversial, is merely a generalization of the more commonly accepted rule of excess reserves, whereby in a system with monopolized currency supply, an individual bank is limited in the loans and investments it may issue by the fact that total reserves are finite and it must settle in base money all currency demands and clearing balances owed to other banks. Both the principle of adverse clearings and the rule of excess reserves can be tested with this experiment design. As of yet, however, this is only a proposed experiment design, so experiments have not yet been run. However, Chapters 1 and 2 provide some evidence in support of the theory using agent-based modeling and simulation.

Lastly, it should be noted that there is clearly more to central banking in practice than the stylized version used in this design, which represents only the aspect of monopoly of note issue. However, there is little agreement on what completely defines central banking. In an earlier version of this paper I proposed a more complicated

[7] This is how Lawrence H. White has described it.

experiment in which the central bank, in addition to monopolizing note issue, manipulates the interest rate and the money supply through open market operations. I may eventually return to that idea. In the meantime it is a fairly safe conjecture that, under reasonable assumptions, this more complete representation of central banking would perform far worse than the current one in terms of bubble reduction. The reason is, as Selgin (1988, p. 49) notes, "Unless some external short-run control is imposed on it, a monopoly bank of issue even when its issues are convertible into commodity money can for some time at least pursue any loan-pricing policy it desires, arbitrarily expanding or contracting the money supply and causing widespread changes in nominal income and prices." Since the current design does not allow for such central bank initiated fluctuations in the money supply, any forthcoming results should be interpreted as a comparison of free banking to a rather idealized version of central banking.

2.0 Theory of Free Banking

To briefly review the theory of free banking it is useful to consider the equation of exchange:

$$MV = Py,$$

where M is the money supply, V is the velocity of money, P is the (output) price level, and *y* is real income or output. For any given value of real income, the general price level will remain stable only if *MV* remains stable, i.e. only if changes in the money supply serve to just offset changes in money's velocity. A free banking system is supposed to tend automatically to stabilize *MV*, given some fixed stock of base money, owing to the tendency for any increase in *MV* to go hand-in-hand with an increase in the volume of interbank clearings, which in turn raises the demand for precautionary bank reserves by increasing the risk of a settlement default associated with any given aggregate reserve stock. As a result, the increase in aggregate spending generates an excess demand for reserves, forcing banks to reduce their loans and to thereby contract the outstanding supply of bank money until the quantity of reserves demanded is once again equal to the quantity on hand. In other words, under an assumption of fixed reserves, in this scenario of an increase in *MV*, the only way for free banks to bring about equilibrium in reserves is to reduce the demand for reserves to its original level, which requires that *MV* be reduced to its original level.

Further, Selgin (1994, p. 1453-57) shows that this result, where free banks tend to adjust the money supply so as to completely offset changes in velocity, depends critically on the freedom of banks to issue notes. When this freedom is revoked, money supply adjustments are less than complete (1994, p. 1457). The derived money supply equations are reproduced here. Equation 6 (p. 1453), representing the money supply under a system of free banking, is as follows:

$$M^* = R/(VQ)[(\varphi\, y)/(2(b-1))]^{1/2} \qquad (6)$$

Where R is the total stock of reserves, b is the number of banks, φ is the proportion of real income transactions to total transactions, and Q essentially represents banker conservatism but will be discussed in greater detail later. Equation 6 implies that the stock of money adjusts in inverse proportion to changes in velocity. Contrast this with the money supply under a system of monopolized note issue, shown as follows (p. 1457):

$$M^* = R/(VQ)[(\varphi\, y)/(2(1 - c)(b\text{-}1))]^{1/2} + R/c$$

Where c is the ratio of currency to deposits. Due to the second term, reflecting the reduction in reserves resulting from base money used as currency, the stock of money does not fully adjust so as to offset changes in velocity. Therefore, in the experiments I expect bubbles to be least prevalent in the free banking treatment, somewhat more prevalent in the central banking treatment, and most prevalent in the baseline treatment.

Finally, Selgin (1994, p. 1452) also discusses the meaning and origin of Q, or banker conservatism, stating:

> A well-known conclusion of the literature on precautionary reserve demand, beginning with Edgeworth, is that a bank's demand for reserves will be proportional to the standard deviation of its net reserve losses (Baltensperger, 1980, p. 9; Laidler, 1991, p. 186). If we denote a bank's demand for reserves as

σQ, then the factor of proportionality, Q, reflects the representative bank's desired level of security against the prospect of a settlement default, chosen to equate the marginal opportunity costs (forgone interest) and marginal benefits (reduced anticipated costs of reserve shortages) of reserve holding. The value of Q will in general be positively related to reserve shortage penalties and negatively related to the rate of interest (Olivera, 1971, pp. 1102-3).

Q is treated as a parameter in the model to be discussed in the next section, and the process of selecting the value of *Q* is discussed in the parameters section (3.1) of Chapter 2.

3.0 Agent-based Model of Banking and Simulation Software

Figure 24 is a flow chart of the banking simulation software. A significant challenge for the banking simulation was to begin the asset trading under conditions of monetary equilibrium, where money demand equals money supplied. Also, bankers should ideally be able to calculate and achieve their optimal precautionary reserve demand as quickly as possible, i.e. before the bubble action starts. The solution I settled on was to introduce a preliminary period in which each trader's cash endowment, given in experimental dollars (E$), is split into a portion which is deposited and a portion which is borrowed and held in cash. In this way, the initial money supply is held constant across all treatments. The portion which is deposited, α, is determined by the initial reserve ratio, a parameter, as follows:

$$\alpha = 1/(2 - \mathit{InitialReserveRatio})$$

The remaining portion of the endowment, 1- α, is borrowed and held in cash. Thus, for example, a 50% initial reserve ratio results in two thirds of the cash endowment deposited and one third borrowed and held in cash. The banks are then left with a reserve ratio of 50%.

I used an initial reserve ratio of 50% because I found through the simulations documented in Chapter 2 that this reserve ratio was close to where bankers settled in later periods; somewhat more at smaller trading scales and somewhat less at larger scales. The initial reserve ratio is in place until bankers have a sufficient clearing sample by which to judge their optimal reserve ratio and precautionary reserve demand. The clearing sample is used by bankers to estimate the clearings distribution, or, in other words, the net reserve loss distribution. As mentioned earlier, the banking literature finds that a bank's precautionary reserve demand is proportional to the standard deviation of its net reserve losses (Selgin, 1994, p. 1452). For the clearing sample length, I settled on roughly one period's worth of clearings: clearings under free banking and 500 under central banking. The disparity is due to the previously mentioned distinction between real-time gross settlement, in the case of central banking, and end-of-day net settlement, in the case of free banking. Particularly, under free banking, clearings only occur for free banks at the clearinghouse, at the end of each banking interval, and there are 10 intervals per period. At the clearinghouse, bankers submit each others banknotes, then the clearinghouse calculates each bank's balance, without charge or note discounting, and finally requests settlement in base money (E$). Under central banking, any sort of bank transaction counts as a clearing, since all currency is in base money (E$) and thus any transaction directly effects base money reserves. For example, a deposit or withdrawal counts as a clearing, so clearings are always much more frequent under central banking.

As mentioned earlier, precautionary reserve demand is proportional to the standard deviation of net reserve losses, σ, and the factor of proportionality is Q. Thus, precautionary reserve demand, *PRD*, is as follows:

$$PRD = \sigma Q,$$

Where,

$$\sigma = [(SSD_C)/(ClearingSampleSize - 1)]^{1/2},$$

And SSD_C is the sum of the squared deviations of clearings around the mean.

Precautionary reserve demand is calculated at the beginning of each trading interval. The optimal reserve ratio, *ORR*, is calculated on an as needed basis and is based on *PRD* as follows:

$$ORR = PRD/(Deposits + IssuedNotes)$$

Likewise, the actual reserve ratio, *ARR*, is calculated on an as needed basis as follows:

$$ARR = Reserves/(Deposits + IssuedNotes)$$

Base money (E$) counts as reserves under both free and central banking. Additionally, under free banking, other bank's notes count as reserves.

Under free banking, so as to quickly achieve a mature free banking economy where notes have fully replaced base money as currency, whenever a trader withdraws or borrows money the bank prefers to pay that money in newly issued bank notes. Thus, in the preliminary period when endowments are split into deposits and borrowed cash, the borrowed cash is in notes from the bank where the money was deposited. After that, traders are free to request money in any form. However, by design, redemptions in base money should be unlikely, and it is expected that traders will continue to use bank notes as currency, since they have no experience using anything else in these experiments. Therefore, assets will likely be purchased in bank notes. Furthermore, assuming no redemptions, after the preliminary period, the only time base money (E$) enters the hands of traders is when dividends are paid in base money at the end of each trading period. Since this is a small fraction of the typical trader's cash, traders will find it not useful as currency and soon deposit this with their depository bank.

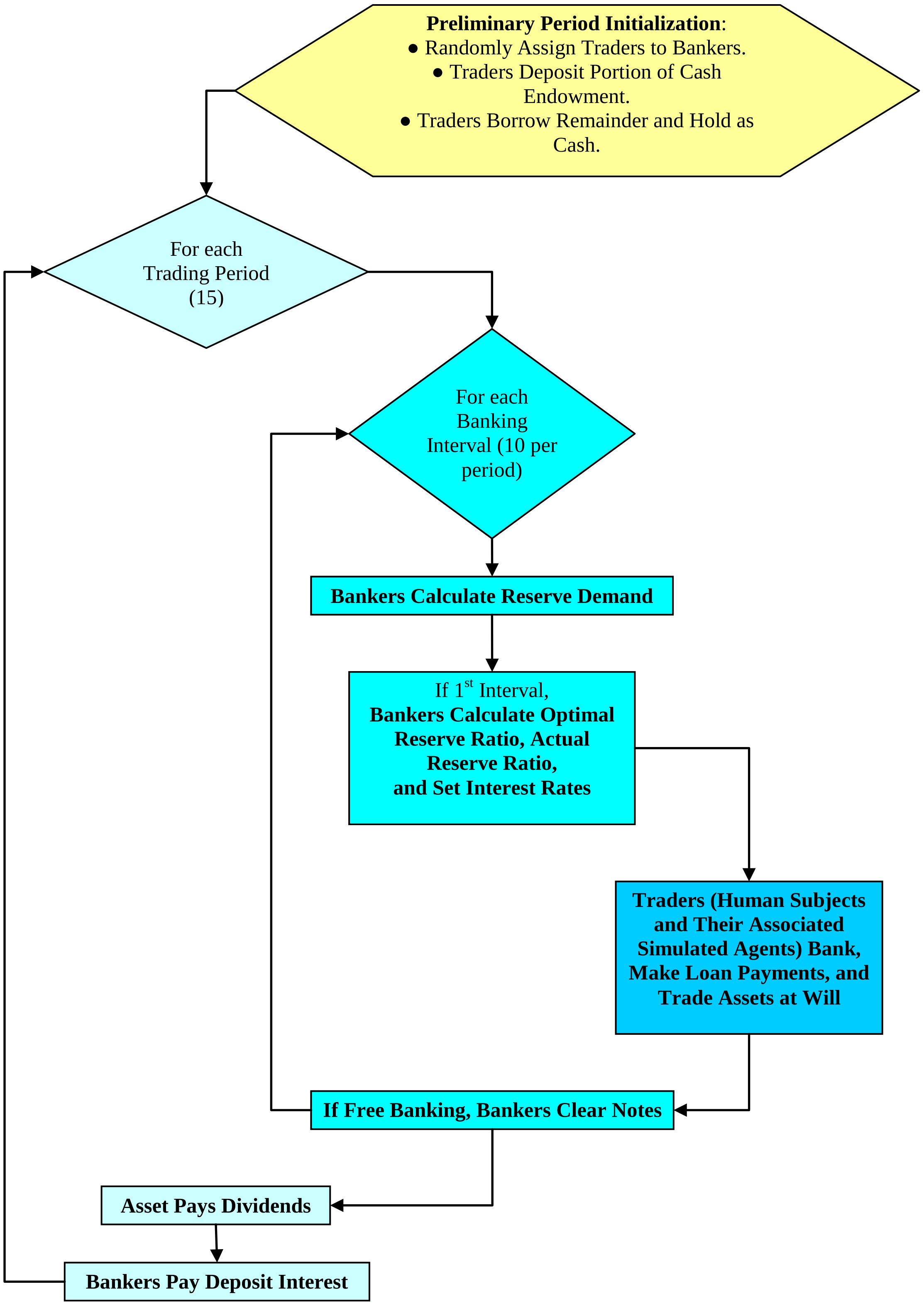

Preliminary Period Initialization:
● Randomly Assign Traders to Bankers.
● Traders Deposit Portion of Cash Endowment.
● Traders Borrow Remainder and Hold as Cash.
For each Trading Period (15)
For each Banking Interval (10 per period)
Bankers Calculate Reserve Demand
If 1st Interval, Bankers Calculate Optimal Reserve Ratio, Actual Reserve Ratio, and Set Interest Rates
Traders (Human Subjects and Their Associated Simulated Agents) Bank, Make Loan Payments, and Trade Assets at Will
If Free Banking, Bankers Clear Notes
Asset Pays Dividends
Bankers Pay Deposit Interest

Figure 24: Banking Simulation Software Flow Chart.

Under both free and central banking, in the preliminary period traders are randomly assigned to banks so that each bank has the same number of traders as well as the same distribution of types of traders. As mentioned earlier, a portion of the endowment is then deposited at the assigned bank and the remainder of the endowment is borrowed from the same bank as a short term loan. There are two types of loans available to traders, short and long term, where the duration of the short term loan is 2 periods and that of the long term loan is 4 periods. Interest is due at the beginning of each period that the loan is outstanding, and the principle is due in the final loan period. For example, a short term loan taken out in period 3, has interest due in period 4 and 5, and principle due in period 5. Loans may be taken out at anytime during the period, i.e. any banking interval, and the interest due is unaffected. Loans may not be taken out in later periods when the due date is beyond the end of the experiment. Traders are allowed a maximum of one short term loan and one long term loan at a time. Both short and long term loans are of size E$5. The preliminary short term loan is special as the size varies depending on the endowment class of the trader, but as can be seen from the parameters used (section 3.1) this loan ranges from E$0.75 for endowment class 1 to E$3.15 for endowment class 3.

The preliminary loan has interest due in the first trading period and principle and interest due in the second trading period. So as to equalize the initial money supply across treatments, I chose initial interest rates such that, given a 50% initial reserve ratio, depository interest earned by traders would be offset by interest paid on the preliminary short term loan. This also means bankers make no money in the preliminary period. Thus, the initial depository rate is 2% and the short term loan rate is 4%. The long term loan rate is 6%, simply a larger number meant to cover some of the cost of increased exposure to default.

After the preliminary period, bankers are free to adjust their interest rates at the beginning of each period. They do so by comparing their optimal and actual reserve ratios, as follows:

$$iRate_{t+1} = iRate_t + iRateCF * (ORR - ARR)$$

Where $iRate_t$ is the interest rate, either depository, short or long term loan, at time t, and *iRateCF* is the correction factor, which is a parameter set to 0.005. The correction factor was chosen so that over the course of a typical experiment interest rates fluctuate over a range of about plus or minus one percentage point.

The standard bankers use for issuing loans is first determined by calculation of the requesting trader's asset-to-liability ratio, where assets include the tradable assets at market value plus deposits and cash. Liabilities are other loans. The minimum asset-to-liability ratio is a parameter, *MinALR*, set to 0.9, which was chosen in conjunction with other parameters so as to be binding only a small percentage of the time. Secondly, and a much more frequently binding requirement, the banker's actual reserve ratio must exceed his optimal reserve ratio.

Finally, bankruptcy is possible for both traders and bankers, and merely results from an inability to pay claims. In the case of bankers, it is treated as a rare calamity and the experimental session is considered a bankrupt session and over, in which case human subjects are paid their cash endowments. In the case of a trader bankruptcy, it is treated as a fairly routine liquidation. First, it is made public, which means no banker will lend to the trader again. Then the immediate creditor is paid all remaining cash and savings. Over the next few trading periods, all remaining assets are liquidated using market orders, and the proceeds are given to the largest creditor. Thereafter, the trader is essentially out of the business of trading or banking.

3.1 Parameters

Many of the banking parameters have been discussed in the previous section, but Table 8 provides a formal list of them. As mentioned earlier, *Q* is a critical banking parameter used by bankers to calculate precautionary demand for reserves and the optimal reserve ratio. The values I use for *Q* come from the simulations discussed in Chapter 2, where I optimize *Q* based on bank profit maximization. Table 7 lists the parameters used in the asset trading portion of the experiment, which are largely taken from Smith et al. (1988).

Table 7: Trading Parameters.

Parameter	**Description**	**Value**
EndowClass1Cash	Cash endowment of Class 1 agents	E$2.25
EndowClass1Assets	Asset endowment of Class 1 agents	3
EndowClass2Cash	Cash endowment of Class 2 agents	E$5.85
EndowClass2Assets	Asset endowment of Class 2 Agents	2
EndowClass3Cash	Cash endowment of Class 3 agents	E$9.45
EndowClass3Assets	Asset endowment of Class 3 agents	1
Dividend1	First of four equally likely dividend possibilities.	E$0.0
Dividend2	Second of four equally likely dividend possibilities.	E$0.08
Dividend3	Third of four equally likely dividend possibilities.	E$0.28
Dividend4	Fourth of four equally likely dividend possibilities.	E$0.60
TP	Total number of trading periods.	15

Table 8: Banking Parameters.

Parameter	**Description**	**Value**
NumBankers	Total number of bankers (simulated agents)	2
FreeBanking	Switch that allows banks to issue notes.	True or False
LoanSize	Size of both short and long term loans	E$5
InitialReserveRatio	The reserve ratio used by bankers in the initial periods until a sufficient sample of clearings is established. Also the ratio used to determine the portion of cash endowment which is initially borrowed.	0.5
ClearingSampleSize	Number of clearings used to calculate precautionary reserve demand.	10 under free banking, which is exactly 1 period, or 500 under central banking, which simulations indicate is typically less than 1 period.
Q	Banker conservatism, used to calculate precautionary reserve demand.	1 to 150. As shown in Figures 2 and 3 of Chapter 3, Q of 40 maximizes profits for bankers under central banking, while Q of 2 does so under free banking.
InitialSavingsRate	Initial interest rate paid by bankers on deposits.	2%
InitialShortLoanRate	Initial interest rate charged by bankers on short term loans.	4%
InitialLongLoanRate	Initial interest rate charged by bankers on long term loans.	6%
ShortLoanDuration	Duration of short term loan, in periods.	2
LongLoanDuration	Duration of long term loan, in periods.	4
iRateCF	Interest rate correction factor.	0.005
MinALR	Minimum asset-to-liability ratio for bankers to issue loans.	0.9
TI	Total number of banking intervals per period.	10

4.0 Laboratory Experiment Design

4.1 Baseline Treatment

This treatment is based on the design of Smith et al (1988), in which about 20 human subjects receive an endowment of *X* experimental dollars and *Y* stock-like assets that pay a dividend from a known probability distribution at the end of each of $T = 15$ (or 30) periods of four minutes each.[8] *X* and *Y* are determined according to each subject's endowment class, as in Smith's original design (1988, p. 1127). In addition, each subject is matched with 10 simulated agents, who receive the same endowment as the human subject and mimic the subject's behavior with some noise. This makes the total population of asset traders 220, thus increasing the number of transactions. As Selgin (1988, pp. 73-78) illustrates, a large number of transactions provides bankers with an approximately continuous distribution of clearings, and this is necessary if appeal is to be had to the law of large numbers in drawing conclusions about the relation of the optimal ratio to various parameters. During each period, subjects, and their associated agents, buy or sell assets by submitting bids or offers, accepting bids or offers, and then confirming resulting contracts.

[8] It is helpful for this treatment and the banking treatments to imagine that each period roughly corresponds to a quarter, i.e. a 3 month period, in the financial world.

4.2 Central Banking Treatment (without note issue)

This is the same as the Baseline treatment, but with 2 additional simulated agents as bankers, so that there are on average about 113 traders, i.e. depositors, per bank. The bankers act according to the model described in the previous section. They have all the significant rights and functionalities of bankers except the right to issue notes. Mainly, bankers accept deposits from traders and then lend some portion of that back out to traders at competitively determined interest rates. In other words, this is a system of free banking in deposits, where neither bank has the right to issue notes. There is no representation of the "central bank" beyond the restriction on note issue, but one can assume that the stock of *base* money, which is in experimental dollars, is created exogenously by a central bank. As in the Baseline Treatment, base money enters the system in the form of cash endowments and also per period dividend payments, and the base money is also the currency. The monetary environment is different in two ways: a) each trader's money endowment consists of a deposit and a loan with one of the bankers, in such a way that the total money supply, i.e. the total of traders' cash holdings and deposits, is constant initially across treatments, and b) changes thereafter in the total money supply are determined partly by dividend payments but also by the amount of credit issued by the two bankers according to each banker's calculation of their demand for precautionary reserves.

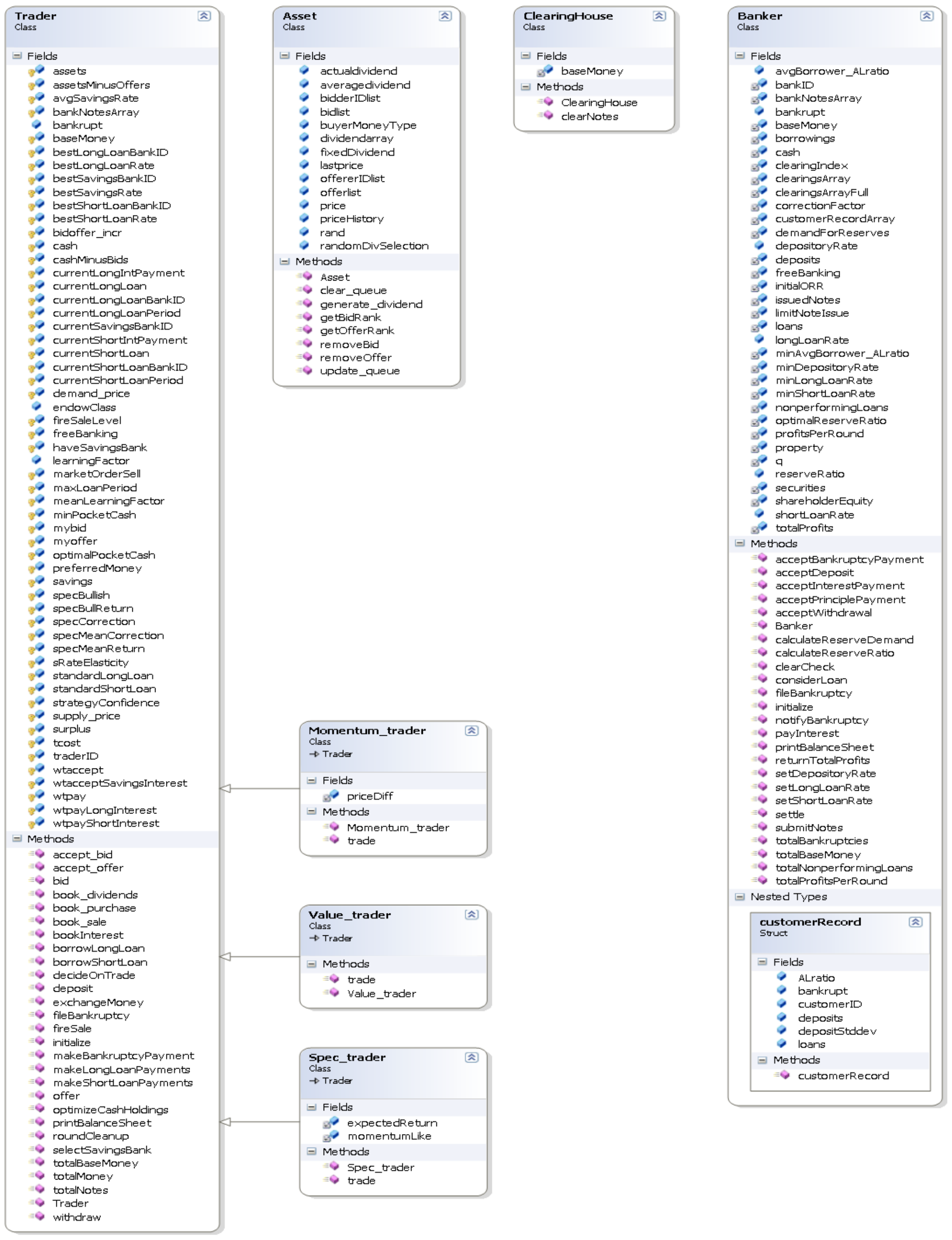
4.3 Free Banking Treatment (with note issue)

This is the same as the Central Banking treatment, but with bankers allowed to issue notes, which are fully redeemable in base money, i.e. experimental dollars. As the banking model described in the previous section indicates, this is a mature free banking economy, so that bank notes have come to fully replace experimental dollars as the currency, while the stock of experimental dollars represents the stock of bank reserves as well as base money. This is implemented by giving bankers a preference for issuing banknotes and traders a preference for using banknotes as currency. Each banker brings the other banker's notes to the clearinghouse on a regular basis, where they are cleared without cost or discounting. As was discussed in the agent-based model section, the settlement process is somewhat different between the two banking treatments. Under central banking, all transactions are in base money including deposits, withdrawals, and loans, in what might be called real-time gross settlement. Under free banking, all transactions, except those involving endowments and dividend payments, are in banknotes and traders have the right, though as mentioned not the preference, to redeem banknotes in base money. On a regular basis, the two free bankers settle each others banknote balances at the clearinghouse using base money, in what might be called end-of-day net settlement.

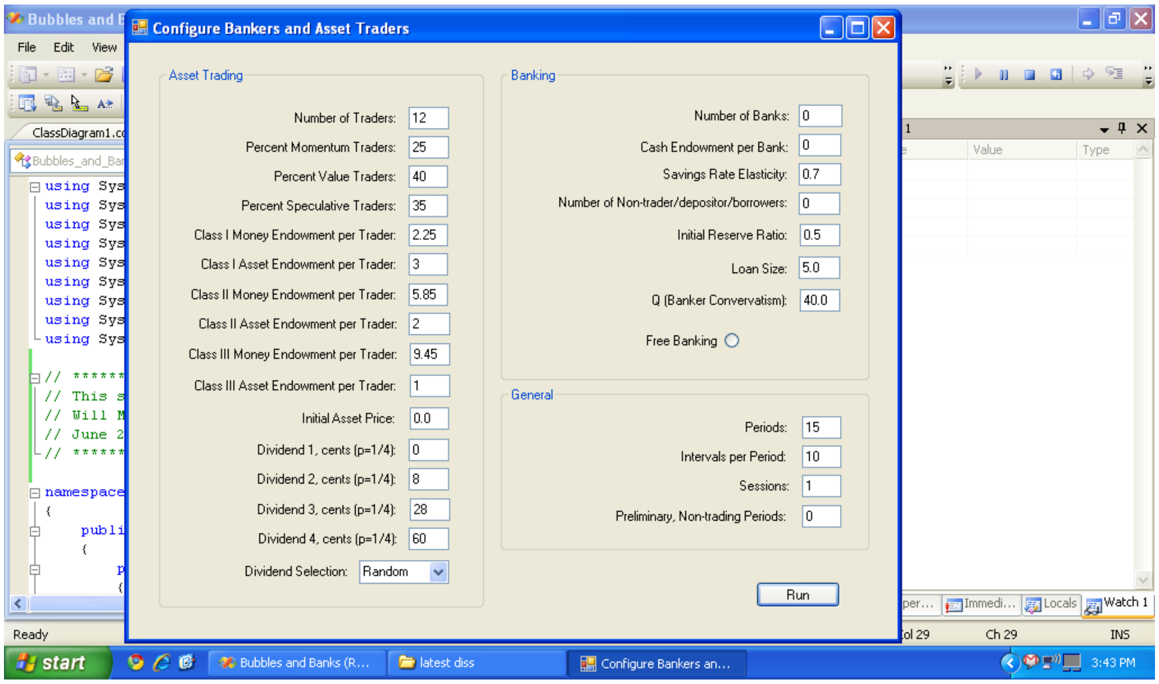
5.0 Conclusion

In summary, this chapter describes a laboratory experiment design which tests the macroeconomic price stability qualities of free banking. The particular details and parameters used in the banking model have been worked out using agent-based modeling and simulation techniques described in Chapter 2. However, there is still a fair amount of work that lies ahead in training human subjects in this environment and then altering certain parameters in response to that. Particularly, it is not clear how quickly human subjects will adjust and learn to both trade assets and bank. It is likely that the additional cognitive load of banking will affect asset prices, which may then require a control treatment. Nonetheless, this appears to be a promising avenue of research, one which brings the scientific power of controlled experiments to long standing debates on the macroeconomic effects of banking.

APPENDIX A: CLASS DIAGRAM OF THE SIMULATION SOFTWARE



APPENDIX B: SCREEN CAPTURE OF THE CONFIGURATION WINDOW OF THE SIMULATION SOFTWARE



Configure Bankers and Asset Traders

Asset Trading

Number of Traders:	12
Percent Momentum Traders:	25
Percent Value Traders:	40
Percent Speculative Traders:	35
Class I Money Endowment per Trader:	2.25
Class I Asset Endowment per Trader:	3
Class II Money Endowment per Trader:	5.85
Class II Asset Endowment per Trader:	2
Class III Money Endowment per Trader:	9.45
Class III Asset Endowment per Trader:	1
Initial Asset Price:	0.0
Dividend 1, cents (p=1/4):	0
Dividend 2, cents (p=1/4):	8
Dividend 3, cents (p=1/4):	28
Dividend 4, cents (p=1/4):	60
Dividend Selection:	Random

Banking

Number of Banks:	0
Cash Endowment per Bank:	0
Savings Rate Elasticity:	0.7
Number of Non-trader/depositor/borrowers:	0
Initial Reserve Ratio:	0.5
Loan Size:	5.0
Q (Banker Convervatism):	40.0
Free Banking	○

General

Periods:	15
Intervals per Period:	10
Sessions:	1
Preliminary, Non-trading Periods:	0

Run

APPENDIX C1: SOURCE CODE OF THE SIMULATION SOFTWARE (FORM1.CS)

```
// ****************************************************************
// This software was written and is owned and copyrighted by
// William McBride of George Mason University, for his dissertation.
// July 23, 2010.
// ****************************************************************

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace Bubbles_and_Banks
{
    public partial class FormMainConf : Form
    {
        public FormMainConf()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        //*****************************************************
        // This button makes the application run, and contains
        // much of the overall structure of the application.
        // First, parameters are taken from the GUI, then files
        // are initialized, then classes are instantiated and
        // initialized, then banking and trading begins,
        // and finally output files are written.
        //*****************************************************
        private void buttonRun_Click(object sender, EventArgs e)
        {
            double savingsInterest = 0.0;
            double dividends = 0.0;
            string tradeResponse;
            double moneySupply = 0.0;
            double baseMoneySupply = 0.0;
            int invID = 0;
            double price_deviation;
            double[] bankNoteArray;

            // Read in parameters from the GUI.
            Int16 numBanks = System.Convert.ToInt16(textBoxNumBanks.Text);
            double bankCash1 = System.Convert.ToDouble(textBoxBankCashEndow.Text);
            double sRateElasticity =
System.Convert.ToDouble(textBoxSavingsRateElasticity.Text);
            Int16 numNonTraders = System.Convert.ToInt16(textBoxNontraders.Text);
            double loanSize = System.Convert.ToDouble(textBoxLoanSize.Text);
            double q = System.Convert.ToDouble(textBoxQ.Text);
            bool freeBanking = System.Convert.ToBoolean(radioButtonFB.Checked);

            // Initialize banknote array.
            bankNoteArray = new double[numBanks];
            for (int i = 0; i < numBanks; i++)
            {
                bankNoteArray[i] = 0.0;
            }

            // Read in parameters from the GUI.
            Int16 numInv = System.Convert.ToInt16(textBoxNumInv.Text);
            double percentMomentum = System.Convert.ToDouble(textBoxPercentMom.Text);
            double percentValue = System.Convert.ToDouble(textBoxPercentValue.Text);
            double percentSpec = System.Convert.ToDouble(textBoxPercentSpec.Text);
            Int16 numMomentumInv = System.Convert.ToInt16(numInv * 0.01 *
percentMomentum);
            Int16 numValueInv = System.Convert.ToInt16(numInv * 0.01 * percentValue);
            Int16 numSpecInv = System.Convert.ToInt16(numInv * 0.01 * percentSpec);
            if ((numMomentumInv + numValueInv + numSpecInv) != numInv)
            {
                MessageBox.Show("Error: Choose trader percentages that produce no
remainder.");
                return;
            }

            // Set up output file names.
            string dateString = DateTime.Now.Year.ToString() +
DateTime.Now.Month.ToString() + DateTime.Now.Day.ToString();
            dateString += DateTime.Now.TimeOfDay.Hours.ToString() +
DateTime.Now.TimeOfDay.Minutes.ToString() + DateTime.Now.TimeOfDay.Seconds.ToString();
            string directory = @"C:\Documents and Settings\Will McBride\My
Documents\Visual Studio 2008\Projects\Bubbles and Banks\Bubbles and Banks\output\";
            directory += "Div" + comboBoxFixedDividend.Text + @"\";
            if (numBanks == 0) directory += @"No Banking\";
            else if (freeBanking) directory += @"Free Banking\";
            else directory += @"Central Banking\";

            // Write header for summary file.
            string filenameSummary = directory + "summary" + dateString + ".csv";
            StreamWriter fileS = new System.IO.StreamWriter(filenameSummary, true);
            string datalineS;
            string headerS =
"BubbleContracts,Volume,TraderBankruptcies,Bank0Bankrupt,Bank0Profits,Bank1Bankrupt,Bank1
Profits,Bank2Bankrupt,Bank2Profits,Bank3Bankrupt,Bank3Profits,Bank4Bankrupt,Bank4Profits"
;
            fileS.WriteLine(headerS);
            fileS.Flush();

            // Write header for meanPrice file.
            string filenameMeanPrice = directory + "meanPrice" + dateString + ".csv";
            StreamWriter fileMP = new System.IO.StreamWriter(filenameMeanPrice, true);
            string datalineMP;
            string headerMP = "Period,MeanPrice,MeanPriceDev,DivValue,Volume";
            fileMP.WriteLine(headerMP);
            fileMP.Flush();

            // Write header for traderEarnings file.
            string filenameEarnings = directory + "traderEarnings" + dateString + ".csv";
            StreamWriter fileE = new System.IO.StreamWriter(filenameEarnings, true);
            string datalineE;
            string headerE = "Trader,Cash";
            if (freeBanking)
            {
                for (int h = 0; h < numBanks; h++)
                {
                    headerE += ",Note" + h.ToString();
                }
            }
            headerE += 
",Savings,Assets,ShLoan,LoLoan,FireSale,SpecBull,SpecReturn,SpecCorr,StratConf,OptCash,Le
arningFactor,EndowClass,Strategy";
            fileE.WriteLine(headerE);
            fileE.Flush();

            // Open trading file.
            string filenameI = directory + "trading" + dateString + ".csv";
            StreamWriter fileI = new System.IO.StreamWriter(filenameI,true);

            // Open banking file.
            string filenameB = directory + "banking" + dateString + ".csv";
            StreamWriter fileB = new System.IO.StreamWriter(filenameB, true);
            string datalineB;
            string datalineBtemp;

            // Write header for config file.
            string filenameConfig = directory + "config" + dateString + ".csv";
            StreamWriter fileC = new System.IO.StreamWriter(filenameConfig, true);
            string datalineC;
            string headerC = 
"Periods,Intervals,NonTradingPeriods,Banks,BankEndow,ReserveRatio,SRateElasticity,LoanSiz
e,Q,FreeBanking,Traders,PercentMomentum,PercentValue,PercentSpec,NumNonTraders,C1MoneyEnd
ow,C1AssetEndow,C2MoneyEndow,C2AssetEndow,C3MoneyEndow,C3AssetEndow,Price,Div0,Div1,Div2,
Div4,DivSelection";
            fileC.WriteLine(headerC);
            fileC.Flush();

            // Write config file.
            datalineC = textBoxPeriods.Text + "," + textBoxIntPerPeriod.Text + "," + 
textBoxNonTradingPeriods.Text + "," + textBoxNumBanks.Text + ",";
            datalineC += textBoxBankCashEndow.Text + "," + textBoxReserveRatio.Text + "," 
+ textBoxSavingsRateElasticity.Text + "," + textBoxLoanSize.Text + "," + textBoxQ.Text + 
"," + radioButtonFB.Checked.ToString() + "," + textBoxNumInv.Text + "," + 
textBoxPercentMom.Text + ",";
            datalineC += textBoxPercentValue.Text + "," + textBoxPercentSpec.Text + "," + 
textBoxNontraders.Text + "," + textBoxC1money.Text + "," + textBoxC1assets.Text + "," + 
textBoxC2money.Text + "," + textBoxC2assets.Text + "," + textBoxC3money.Text + "," + 
textBoxC3assets.Text + ",";
            datalineC += textBoxAssetPrice.Text + "," + textBoxDiv1.Text + ",";
            datalineC += textBoxDiv2.Text + "," + textBoxDiv3.Text + "," + 
textBoxDiv4.Text + "," + comboBoxFixedDividend.Text;
            fileC.WriteLine(datalineC);
            fileC.Flush();

            // Read in parameters from GUI.
            int c1assets = System.Convert.ToInt32(textBoxC1assets.Text);
            int c2assets = System.Convert.ToInt32(textBoxC2assets.Text);
            int c3assets = System.Convert.ToInt32(textBoxC3assets.Text);

            double c1cash;
            double c2cash;
            double c3cash;
            double c1loans;
            double c2loans;
            double c3loans;
            double c1deposits;
            double c2deposits;
            double c3deposits;

            // Read in parameters from GUI.
            // Initial Reserve Ratio, a parameter.
            double initialRR = System.Convert.ToDouble(textBoxReserveRatio.Text);
            // alpha is a factor used to calculate initial loans
            // and deposits from the money endowment, and assumes
            // that all money is either deposited or borrowed and
            // then held in cash.
            // This allows the system to be set using a reserve
            // ratio, which can be chosen to be near optimal, and
            // thus the system can be initialized at near monetary
            // equilibrium.
            double alpha = 1.0 / (2.0 - initialRR);

            // Read in parameters from GUI.
            // Determine initial loans and deposits using alpha.
            if (numBanks == 0)
            {
                c1cash = System.Convert.ToDouble(textBoxC1money.Text);
                c2cash = System.Convert.ToDouble(textBoxC2money.Text);
                c3cash = System.Convert.ToDouble(textBoxC3money.Text);
                c1loans = 0.0;
                c2loans = 0.0;
                c3loans = 0.0;
                c1deposits = 0.0;
                c2deposits = 0.0;
                c3deposits = 0.0;
            }
            else
            {
                c1cash = 0.0;
                c2cash = 0.0;
                c3cash = 0.0;
                c1loans = (1.0 - alpha) * System.Convert.ToDouble(textBoxC1money.Text);
                c2loans = (1.0 - alpha) * System.Convert.ToDouble(textBoxC2money.Text);
                c3loans = (1.0 - alpha) * System.Convert.ToDouble(textBoxC3money.Text);
                c1deposits = alpha * System.Convert.ToDouble(textBoxC1money.Text);
                c2deposits = alpha * System.Convert.ToDouble(textBoxC2money.Text);
                c3deposits = alpha * System.Convert.ToDouble(textBoxC3money.Text);
            }

            int initialBank = 0;
            int endowClass = 1;
            double iCash = 0.0;
            Int32 iAssets = 0;
            double iDeposits = 0.0;
            double iLoans = 0.0;

            // Read in parameters from GUI.
            double price1 = System.Convert.ToDouble(textBoxAssetPrice.Text);
            double div0 = System.Convert.ToDouble(textBoxDiv1.Text) / 100.0;
            double div1 = System.Convert.ToDouble(textBoxDiv2.Text) / 100.0;
            double div2 = System.Convert.ToDouble(textBoxDiv3.Text) / 100.0;
            double div3 = System.Convert.ToDouble(textBoxDiv4.Text) / 100.0;
            string divSelection = comboBoxFixedDividend.Text;

            // Initialize the asset object.
            Asset TheAsset = new Asset(price1, div0, div1, div2, div3, divSelection);

            // Read in parameters from GUI.
            int numPeriods = System.Convert.ToInt32(textBoxPeriods.Text);
            int numIntervalsPerPeriod = System.Convert.ToInt32(textBoxIntPerPeriod.Text);
            int numNonTradingPeriods =
System.Convert.ToInt32(textBoxNonTradingPeriods.Text);

            // Initialize output variables.
            int bubbleContracts = 0;
            int[] bubbleContractsArray = new int[numPeriods];
            int[] volumeArray = new int[numPeriods];
            int totalVolume = 0;
            double[] meanPriceArray = new double[numPeriods];
            double[] meanPriceDevArray = new double[numPeriods];
            double[] divValueArray = new double[numPeriods];
            for (int i = 0; i < numPeriods; i++)
            {
                bubbleContractsArray[i] = 0;
                volumeArray[i] = 0;
                meanPriceArray[i] = 0.0;
                meanPriceDevArray[i] = 0.0;
                divValueArray[i] = 0.0;
            }
            string bubbleSummary;

            // Write header for trading file.
            string datalineI;
            string headerI = "Period,Int,Price,PriceDev,Buyer,BuCash";
            if (freeBanking)
            {
                for (int h = 0; h < numBanks; h++)
                {
                    headerI += ",BuNote" + h.ToString();
                }
            }
            headerI += 
",BuSavings,BuAssets,BuShLoan,BuLoLoan,BuFireSale,BuSpecBull,BuSpecReturn,BuSpecCorr,BuSt
ratConf,BuOptCash,Seller,SeCash";
            if (freeBanking)
            {
                for (int h = 0; h < numBanks; h++)
                {
                    headerI += ",SeNote" + h.ToString();
                }
            }
            headerI += 
",SeSavings,SeAssets,SeShLoan,SeLoLoan,SeFireSale,SeSpecBull,SeSpecReturn,SeSpecCorr,SeSt
ratConf,SeOptCash";
            fileI.WriteLine(headerI);
            fileI.Flush();

            // Write header for banking file.
            string headerB = "Period,Int,TranType,Amount,Comment,Banker,BaCash";
            if (freeBanking)
            {
                for (int h = 0; h < numBanks; h++)
                {
                    headerB += ",BaNote" + h.ToString();
                }
            }
            headerB += 
",BaLoans,BaInotes,BaDeposits,ReserveR,OptRR,ReserveD,ChRate,ShLoanRate,LoLoanRate,Trader
,TrCash";
            if (freeBanking)
            {
                for (int h = 0; h < numBanks; h++)
                {
                    headerB += ",TrNote" + h.ToString();
                }
            }
            headerB += 
",TrSavings,TrAssets,TrShLoan,TrLoLoan,FireSale,SpecBull,SpecReturn,SpecCorr,StratConf,Tr
OptCash";
            fileB.WriteLine(headerB);
            fileB.Flush();

            // Establish clearing sample size.
            int clearingSampleSize;
            if (freeBanking)
            {
                // One period worth of clearings.
                clearingSampleSize = 10;
            }
            else
            {
                // Pick a nice big number that's not likely to exceed
                // the typical number of clearings per period.
                clearingSampleSize = 500;
            }

            // Instantiate and initialize banker array.
            Banker[] BankerArray = new Banker[numBanks];
            for (int b = 0; b < numBanks; b++)
            {
                BankerArray[b] = new Banker();
                BankerArray[b].initialize(bankCash1, numInv + numNonTraders, initialRR,
b, numBanks, freeBanking, clearingSampleSize, q);
            }

            // Instantiate clearing house.
            ClearingHouse TheClearingHouse = new ClearingHouse();

            // Instantiate trader arrays.
            Momentum_trader[] MomentumTraderArray = new Momentum_trader[numMomentumInv];
            Value_trader[] ValueTraderArray = new Value_trader[numValueInv];
            Spec_trader[] SpecTraderArray = new Spec_trader[numSpecInv];
            NonTrader[] NonTraderArray = new NonTrader[numNonTraders];

            // Assign endowment values and initialize trader arrays.
            for (int i = 0; i < numMomentumInv; i++)
            {
                if (endowClass == 1)
                {
                    iCash = c1cash;
                    iAssets = c1assets;
                    iLoans = c1loans;
                    iDeposits = c1deposits;
                }
                else if (endowClass == 2)
                {
                    iCash = c2cash;
                    iAssets = c2assets;
                    iLoans = c2loans;
                    iDeposits = c2deposits;
                }
                else if (endowClass == 3)
                {
                    iCash = c3cash;
                    iAssets = c3assets;
                    iLoans = c3loans;
                    iDeposits = c3deposits;
                }

                MomentumTraderArray[i] = new Momentum_trader();
                // Print each bank operation to a file.
                datalineB = "-1,-1,";
                datalineB += MomentumTraderArray[i].initialize(BankerArray, TheAsset,
iCash, iDeposits, iLoans, iAssets, i, sRateElasticity, freeBanking, endowClass,
initialBank, loanSize);
                fileB.WriteLine(datalineB);
                fileB.Flush();

                endowClass++;
                if (endowClass == 4)
                {
                    endowClass = 1;
                    initialBank++;
                    if (initialBank == numBanks)
                    {
                        initialBank = 0;
                    }
                }
            }
            for (int j = 0; j < numValueInv; j++)
            {
                if (endowClass == 1)
                {
                    iCash = c1cash;
                    iAssets = c1assets;
                    iLoans = c1loans;
                    iDeposits = c1deposits;
                }
                else if (endowClass == 2)
                {
                    iCash = c2cash;
                    iAssets = c2assets;
                    iLoans = c2loans;
                    iDeposits = c2deposits;
                }
                else if (endowClass == 3)
                {
                    iCash = c3cash;
                    iAssets = c3assets;
                    iLoans = c3loans;
                    iDeposits = c3deposits;
                }

                ValueTraderArray[j] = new Value_trader();
                // Print each bank operation to a file.
                datalineB = "-1,-1,";
                datalineB += ValueTraderArray[j].initialize(BankerArray, TheAsset, iCash, 
iDeposits, iLoans, iAssets, (numMomentumInv + j), sRateElasticity, freeBanking, 
endowClass, initialBank, loanSize);
                fileB.WriteLine(datalineB);
                fileB.Flush();

                endowClass++;
                if (endowClass == 4)
                {
                    endowClass = 1;
                    initialBank++;
                    if (initialBank == numBanks)
                    {
                        initialBank = 0;
                    }
                }
            }
            for (int k = 0; k < numSpecInv; k++)
            {
                if (endowClass == 1)
                {
                    iCash = c1cash;
                    iAssets = c1assets;
                    iLoans = c1loans;
                    iDeposits = c1deposits;
                }
                else if (endowClass == 2)
                {
                    iCash = c2cash;
                    iAssets = c2assets;
                    iLoans = c2loans;
                    iDeposits = c2deposits;
                }
                else if (endowClass == 3)
                {
                    iCash = c3cash;
                    iAssets = c3assets;
                    iLoans = c3loans;
                    iDeposits = c3deposits;
                }

                SpecTraderArray[k] = new Spec_trader();
                // Print each bank operation to a file.
                datalineB = "-1,-1,";
                datalineB += SpecTraderArray[k].initialize(BankerArray, TheAsset, iCash,
iDeposits, iLoans, iAssets, (numMomentumInv + numValueInv + k), sRateElasticity,
freeBanking, endowClass, initialBank, loanSize);
                fileB.WriteLine(datalineB);
                fileB.Flush();

                endowClass++;
                if (endowClass == 4)
                {
                    endowClass = 1;
                    initialBank++;
                    if (initialBank == numBanks)
                    {
                        initialBank = 0;
                    }
                }
            }
            for (int l = 0; l < numNonTraders; l++)
            {
                // Give nontraders class three money, the most money,
                // but no assets.
                endowClass = 3;
                iCash = c3cash;
                iAssets = 0;
                iLoans = c3loans;
                iDeposits = c3deposits;
                NonTraderArray[l] = new NonTrader();
                // Print each bank operation to a file.
                datalineB = "-1,-1,";
                datalineB += NonTraderArray[l].initialize(BankerArray, TheAsset, iCash,
iDeposits, iLoans, iAssets, (numMomentumInv + numValueInv + numSpecInv + l),
sRateElasticity, freeBanking, endowClass, initialBank, loanSize);
                fileB.WriteLine(datalineB);
                fileB.Flush();

                initialBank++;
                if (initialBank == numBanks)
                {
                    initialBank = 0;
                }
            }

            int iMom = 0;
            int iVal = 0;
            int iSpec = 0;
            int iNonTrader = 0;

            int randomBankingAgent = 0;

            // Start the cycle of periods.
            for (int r = 0; r < numPeriods; r++)
            {
                // Start the cycle of intervals.
                for (int s = 0; s < numIntervalsPerPeriod; s++)
                {
                    // Banks calculate reserve demand and set rates.
                    for (int b = 0; b < numBanks; b++)
                    {
                        // Print number of clearings.
                        datalineB = r.ToString() + ", ,MeanClearings,";
                        datalineB += BankerArray[b].calculateReserveDemand() + ",Bank" +
b.ToString();

                        fileB.WriteLine(datalineB);
                        fileB.Flush();

                        // Banks set interest rates only at the
                        // beginning of each period.
                        if (s == 0)
                        {
                            BankerArray[b].setDepositoryRate();
                            BankerArray[b].setLongLoanRate();
                            BankerArray[b].setShortLoanRate();
                        }
                    }

                    // Total up money supply each interval, for
                    // printing.
                    moneySupply = 0.0;
                    baseMoneySupply = 0.0;
                    for (int i = 0; i < numMomentumInv; i++)
                    {
                        moneySupply += MomentumTraderArray[i].totalMoney();
                        baseMoneySupply += MomentumTraderArray[i].totalBaseMoney();
                    }
                    for (int j = 0; j < numValueInv; j++)
                    {
                        moneySupply += ValueTraderArray[j].totalMoney();
                        baseMoneySupply += ValueTraderArray[j].totalBaseMoney();
                    }
                    for (int k = 0; k < numSpecInv; k++)
                    {
                        moneySupply += SpecTraderArray[k].totalMoney();
                        baseMoneySupply += SpecTraderArray[k].totalBaseMoney();
                    }
                    for (int i = 0; i < BankerArray.Length; i++)
                    {
                        baseMoneySupply += BankerArray[i].totalBaseMoney();
                    }

                    // Print money supply.
                    datalineB = r.ToString() + "," + s.ToString() + ",MoneySupply,";
                    datalineB += moneySupply.ToString() + ",Base=" +
baseMoneySupply.ToString();
                    fileB.WriteLine(datalineB);
                    fileB.Flush();

                    iMom = 0;
                    iVal = 0;
                    iSpec = 0;
                    iNonTrader = 0;

                    // Cycle through each trader, and each type so
                    // that the series
                    // is random but proportional to trader type.
                    while (((iMom + iVal + iSpec) < numInv) | (iNonTrader <
numNonTraders))
                    {
                        randomBankingAgent = TheAsset.rand.Next(numInv + numNonTraders);
                        if ((randomBankingAgent < numMomentumInv) & (iMom <
numMomentumInv))
                        {
                            // Momentum traders bank.
                            if (numBanks > 0)
                            {
                                if (s == 0)
                                {
                                    // Only search for best rates at
                                    // the beginning of each period.
                                    // Print each bank operation to a
                                    // file.
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB +=
MomentumTraderArray[iMom].selectSavingsBank(BankerArray, TheAsset, -1);
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();

                                    if (freeBanking)
                                    {
                                        // Switch all money to preferred
                                        // money, to make transactions
                                        // easier.
                                        datalineBtemp =
MomentumTraderArray[iMom].exchangeMoney(BankerArray);
                                        if (datalineBtemp != null)
                                        {
                                            datalineB = r.ToString() + "," + s.ToString()
+ ",";

                                            datalineB += datalineBtemp;
                                            fileB.WriteLine(datalineB);
                                            fileB.Flush();
                                        }
                                    }

                                    // Make short loan payments if any due
                                    // at the beginning of each period.
                                    // Print each bank operation to a file.
                                    datalineBtemp =
MomentumTraderArray[iMom].makeShortLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                    // Make long loan payments if any
                                    // due at the beginning of each
                                    // period.
                                    // Print each bank operation to a
                                    // file.
                                    datalineBtemp =
MomentumTraderArray[iMom].makeLongLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                if (freeBanking)
                                {
                                    // Switch all money to preferred money, to make
transactions easier.
                                    datalineBtemp =
MomentumTraderArray[iMom].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                // Print each bank operation to a file.
                                datalineB = r.ToString() + "," + s.ToString() + ",";
                                // Traders optimize cash holdings,
                                // by depositing cash in
                                // excess of the optimal cash level
                                // or otherwise withdrawing
                                // or borrowing cash to achieve the
                                // optimal cash level.
                                datalineB +=
MomentumTraderArray[iMom].optimizeCashHoldings(BankerArray, TheAsset, numPeriods - r);
                                fileB.WriteLine(datalineB);
                                fileB.Flush();
                                if (freeBanking)
                                {
                                    // Switch all money to preferred money,
                                    // to make transactions easier.
                                    datalineBtemp =
MomentumTraderArray[iMom].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                            }
                            // Momentum traders trade.
                            if (r >= numNonTradingPeriods)
                            {
                                tradeResponse =
MomentumTraderArray[iMom].trade(BankerArray, TheAsset, (numIntervalsPerPeriod - s), r);
                                if (tradeResponse == "Contract")
                                {
                                    // Print each trade to a file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    price_deviation = TheAsset.bidlist[0] -
(TheAsset.averagedividend * (numPeriods - r));
                                    if (price_deviation >= 1.0)
                                    {
                                        bubbleContracts++;
                                        bubbleContractsArray[r]++;
                                    }
                                    volumeArray[r]++;
                                    meanPriceArray[r] += TheAsset.bidlist[0];
                                    datalineI += "," + TheAsset.bidlist[0].ToString() +
"," + price_deviation.ToString();
                                    if (TheAsset.bidderIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.bidderIDlist[0];
MomentumTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.bidderIDlist[0] < (numMomentumInv +
numValueInv))
                                    {
                                        invID = TheAsset.bidderIDlist[0] -
numMomentumInv;
                                        ValueTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.bidderIDlist[0] - numMomentumInv
- numValueInv;
                                        SpecTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
                                    }
                                    if (TheAsset.offererIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.offererIDlist[0];
                                        MomentumTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.offererIDlist[0] < (numMomentumInv
+ numValueInv))
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv;
                                        ValueTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv - numValueInv;
                                        SpecTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
                                    }
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();

                                    TheAsset.update_queue();
                                }
                                else if (tradeResponse != "Nothing")
                                {
                                    // Print each bid or offer to a file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    datalineI += "," + tradeResponse;
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();
                                }
                            }
                            iMom++;
                        }
                        else if ((randomBankingAgent < (numMomentumInv + numValueInv)) &
(iVal < numValueInv))
                        {
                            // Value traders bank.
                            if (numBanks > 0)
                            {
                                if (s == 0)
                                {
                                    // Only search for best rates at the
                                    // beginning of each period.
                                    // Print each bank operation to a
                                    // file.
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB +=
ValueTraderArray[iVal].selectSavingsBank(BankerArray, TheAsset, -1);
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();

                                    if (freeBanking)
                                    {
                                        // Switch all money to
                                        // preferred money, to make
                                        // transactions easier.
                                        datalineBtemp =
ValueTraderArray[iVal].exchangeMoney(BankerArray);
                                        if (datalineBtemp != null)
                                        {
                                            datalineB = r.ToString() + "," + s.ToString()
+ ",";

                                            datalineB += datalineBtemp;
                                            fileB.WriteLine(datalineB);
                                            fileB.Flush();
                                        }
                                    }

                                    // Make short loan payments if any
                                    //
                                    // due at the beginning of each period.
                                    // Print each bank operation to
                                    // a file.
                                    datalineBtemp =
ValueTraderArray[iVal].makeShortLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                    // Make long loan payments if any
                                    // due at the beginning of each
                                    // period.
                                    // Print each bank operation to
                                    // a file.
                                    datalineBtemp =
ValueTraderArray[iVal].makeLongLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";
                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                if (freeBanking)
                                {
                                    // Switch all money to preferred money, to make
transactions easier.
                                    datalineBtemp =
ValueTraderArray[iVal].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";
                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                // Print each bank operation to a file.
                                datalineB = r.ToString() + "," + s.ToString() + ",";
                                // Traders optimize cash holdings,
                                // by depositing cash in
                                // excess of the optimal cash
                                // level or otherwise withdrawing
                                // or borrowing cash to achieve the
                                // optimal cash level.
                                datalineB +=
ValueTraderArray[iVal].optimizeCashHoldings(BankerArray, TheAsset, numPeriods - r);
                                fileB.WriteLine(datalineB);
                                fileB.Flush();
                                if (freeBanking)
                                {
                                    // Switch all money to preferred
                                    // money, to make transactions
                                    // easier.
                                    datalineBtemp =
ValueTraderArray[iVal].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";
                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                            }
                            // Value traders trade.
                            if (r >= numNonTradingPeriods)
                            {
                                tradeResponse = ValueTraderArray[iVal].trade(BankerArray,
TheAsset, (numPeriods - r), (numIntervalsPerPeriod - s), r);
                                if (tradeResponse == "Contract")
                                {
                                    // Print each trade to a file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    price_deviation = TheAsset.bidlist[0] -
(TheAsset.averagedividend * (numPeriods - r));
                                    if (price_deviation >= 1.0)
                                    {
                                        bubbleContracts++;
                                        bubbleContractsArray[r]++;
                                    }
                                    volumeArray[r]++;
                                    meanPriceArray[r] += TheAsset.bidlist[0];
                                    datalineI += "," + TheAsset.bidlist[0].ToString() +
"," + price_deviation.ToString();
                                    if (TheAsset.bidderIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.bidderIDlist[0];

MomentumTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.bidderIDlist[0] < (numMomentumInv +
numValueInv))
                                    {
                                        invID = TheAsset.bidderIDlist[0] -
numMomentumInv;

                                        ValueTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.bidderIDlist[0] - numMomentumInv
- numValueInv;

                                        SpecTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
                                    }
                                    if (TheAsset.offererIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.offererIDlist[0];
                                        MomentumTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.offererIDlist[0] < (numMomentumInv
+ numValueInv))
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv;

                                        ValueTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv - numValueInv;

                                        SpecTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
```

```
                                    }
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();

                                    TheAsset.update_queue();
                                }
                                else if (tradeResponse != "Nothing")
                                {
                                    // Print each bid or offer to a
                                    // file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    datalineI += "," + tradeResponse;
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();
                                }
                            }
                            iVal++;
                        }
                        else if ((randomBankingAgent < (numMomentumInv + numValueInv +
numSpecInv)) & (iSpec < numSpecInv))
                        {
                            // Speculative traders bank.
                            if (numBanks > 0)
                            {
                                if (s == 0)
                                {
                                    // Only search for best rates at
                                    // the beginning of each period.
                                    // Print each bank operation to
                                    // a file.
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB +=
SpecTraderArray[iSpec].selectSavingsBank(BankerArray, TheAsset, -1);
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();

                                    if (freeBanking)
                                    {
                                        // Switch all money to preferred
                                        // money, to make transactions
                                        // easier.
                                        datalineBtemp =
SpecTraderArray[iSpec].exchangeMoney(BankerArray);
                                        if (datalineBtemp != null)
                                        {
                                            datalineB = r.ToString() + "," + s.ToString()
+ ",";

                                            datalineB += datalineBtemp;
                                            fileB.WriteLine(datalineB);
                                            fileB.Flush();
                                        }
                                    }

                                    // Make short loan payments if any
                                    // due at the beginning of each
                                    // period.
                                    // Print each bank operation to a
                                    // file.
                                    datalineBtemp =
SpecTraderArray[iSpec].makeShortLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                    // Make long loan payments if any
                                    // due at the beginning of each
                                    // period.
                                    // Print each bank operation to a
                                    // file.
                                    datalineBtemp =
SpecTraderArray[iSpec].makeLongLoanPayments(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                if (freeBanking)
                                {
                                    // Switch all money to preferred
                                    // money, to make transactions
                                    // easier.
                                    datalineBtemp =
SpecTraderArray[iSpec].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                                // Print each bank operation to a file.
                                datalineB = r.ToString() + "," + s.ToString() + ",";
                                // Traders optimize cash holdings,
                                // by depositing cash in
                                // excess of the optimal cash level
                                // or otherwise withdrawing
                                // or borrowing cash to achieve the
                                // optimal cash level.
                                datalineB +=
SpecTraderArray[iSpec].optimizeCashHoldings(BankerArray, TheAsset, numPeriods - r);
                                fileB.WriteLine(datalineB);
                                fileB.Flush();
                                if (freeBanking)
                                {
                                    // Switch all money to preferred money,
                                    // to make transactions easier.
                                    datalineBtemp =
SpecTraderArray[iSpec].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";

                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }
                            }
                            // Speculative traders trade.
                            if (r >= numNonTradingPeriods)
                            {
                                tradeResponse = SpecTraderArray[iSpec].trade(BankerArray,
TheAsset, (numPeriods - r), numPeriods, (numIntervalsPerPeriod - s), r);
                                if (tradeResponse == "Contract")
                                {
                                    // Print each trade to a file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    price_deviation = TheAsset.bidlist[0] -
(TheAsset.averagedividend * (numPeriods - r));
                                    if (price_deviation >= 1.0)
                                    {
                                        bubbleContracts++;
                                        bubbleContractsArray[r]++;
                                    }
                                    volumeArray[r]++;
                                    meanPriceArray[r] += TheAsset.bidlist[0];
                                    datalineI += "," + TheAsset.bidlist[0].ToString() +
"," + price_deviation.ToString();
                                    if (TheAsset.bidderIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.bidderIDlist[0];

MomentumTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.bidderIDlist[0] < (numMomentumInv +
numValueInv))
                                    {
                                        invID = TheAsset.bidderIDlist[0] -
numMomentumInv;
                                        ValueTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.bidderIDlist[0] - numMomentumInv
- numValueInv;
                                        SpecTraderArray[invID].book_purchase(TheAsset);
                                        datalineI += "," +
TheAsset.bidderIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
                                    }
                                    if (TheAsset.offererIDlist[0] < numMomentumInv)
                                    {
                                        invID = TheAsset.offererIDlist[0];
                                        MomentumTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
MomentumTraderArray[invID].printBalanceSheet();
                                    }
                                    else if (TheAsset.offererIDlist[0] < (numMomentumInv
+ numValueInv))
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv;
                                        ValueTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
```

```
                                        datalineI += "," +
ValueTraderArray[invID].printBalanceSheet();
                                    }
                                    else
                                    {
                                        invID = TheAsset.offererIDlist[0] -
numMomentumInv - numValueInv;
                                        SpecTraderArray[invID].book_sale(TheAsset);
                                        datalineI += "," +
TheAsset.offererIDlist[0].ToString();
                                        datalineI += "," +
SpecTraderArray[invID].printBalanceSheet();
                                    }
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();

                                    TheAsset.update_queue();
                                }
                                else if (tradeResponse != "Nothing")
                                {
                                    // Print each bid or offer to a file.
                                    datalineI = r.ToString() + "," + s.ToString();
                                    datalineI += "," + tradeResponse;
                                    fileI.WriteLine(datalineI);
                                    fileI.Flush();
                                }
                            }
                            iSpec++;
                        }
                        // Non-traders bank and then trade.
                        // Not used to generate dissertation data.
                        else if ((randomBankingAgent < (numInv + numNonTraders)) &
(iNonTrader < numNonTraders))
                        {
                            // Nontraders bank.
                            if (s == 0)
                            {
                                // Only search for best rates at the beginning of each
period.
                                // Print each bank operation to a file.
                                datalineB = r.ToString() + "," + s.ToString() + ",";
                                datalineB +=
NonTraderArray[iNonTrader].selectSavingsBank(BankerArray, TheAsset, -1);
                                fileB.WriteLine(datalineB);
                                fileB.Flush();

                                if (freeBanking)
                                {
                                    // Switch all money to preferred money,
                                    // to make transactions easier.
                                    datalineBtemp =
NonTraderArray[iNonTrader].exchangeMoney(BankerArray);
                                    if (datalineBtemp != null)
                                    {
                                        datalineB = r.ToString() + "," + s.ToString() +
",";
                                        datalineB += datalineBtemp;
                                        fileB.WriteLine(datalineB);
                                        fileB.Flush();
                                    }
                                }

                                // Make short loan payments if any due
                                // at the beginning of each period.
                                // Print each bank operation to a file.
                                datalineBtemp =
NonTraderArray[iNonTrader].makeShortLoanPayments(BankerArray);
                                if (datalineBtemp != null)
                                {
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB += datalineBtemp;
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();
                                }
                                // Make long loan payments if any due
                                // at the beginning of each period.
                                // Print each bank operation to a file.
                                datalineBtemp =
NonTraderArray[iNonTrader].makeLongLoanPayments(BankerArray);
                                if (datalineBtemp != null)
                                {
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB += datalineBtemp;
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();
                                }
                            }
                            if (freeBanking)
                            {
                                // Switch all money to preferred money,
                                // to make transactions easier.
                                datalineBtemp =
NonTraderArray[iNonTrader].exchangeMoney(BankerArray);
                                if (datalineBtemp != null)
                                {
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB += datalineBtemp;
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();
                                }
                            }
                            // Print each bank operation to a file.
                            datalineB = r.ToString() + "," + s.ToString() + ",";
                            // Traders optimize cash holdings,
                            // by depositing cash in excess of
                            // the optimal cash level or otherwise
                            // withdrawing or borrowing
                            // cash to achieve the optimal cash level.
                            datalineB +=
NonTraderArray[iNonTrader].optimizeCashHoldings(BankerArray, TheAsset, numPeriods - r);
                            fileB.WriteLine(datalineB);
                            fileB.Flush();
                            if (freeBanking)
                            {
                                // Switch all money to preferred money,
                                // to make transactions easier.
                                datalineBtemp =
NonTraderArray[iNonTrader].exchangeMoney(BankerArray);
                                if (datalineBtemp != null)
                                {
                                    datalineB = r.ToString() + "," + s.ToString() + ",";
                                    datalineB += datalineBtemp;
                                    fileB.WriteLine(datalineB);
                                    fileB.Flush();
                                }
                            }

                            // Instead of trading, non-traders work and consume.
                            NonTraderArray[iNonTrader].work(TheAsset);
                            NonTraderArray[iNonTrader].consume(TheAsset);
```

```
                                iNonTrader++;
                                //iTraderRatio = System.Convert.ToDouble(iNonTrader) /
System.Convert.ToDouble(iMom + iVal + iSpec);
                            }
                        }

                        // Clear notes at the end of each interval.
                        // Perhaps too frequent, but every
                        // period is too infrequent.
                        if (freeBanking)
                        {
                            // Print status to file.
                            datalineB = r.ToString() + "," + s.ToString() + ",";
                            datalineB += TheClearingHouse.clearNotes(BankerArray);
                            fileB.WriteLine(datalineB);
                            fileB.Flush();
                        }

                        // Total up money supply each interval, for printing.
                        moneySupply = 0.0;
                        baseMoneySupply = 0.0;
                        for (int x = 0; x < numBanks; x++)
                        {
                            bankNoteArray[x] = 0.0;
                        }
                        for (int i = 0; i < numMomentumInv; i++)
                        {
                            for (int x = 0; x < numBanks; x++)
                            {
                                bankNoteArray[x] += MomentumTraderArray[i].totalNotes(x);
                            }
                            moneySupply += MomentumTraderArray[i].totalMoney();
                            baseMoneySupply += MomentumTraderArray[i].totalBaseMoney();
                        }
                        for (int j = 0; j < numValueInv; j++)
                        {
                            for (int x = 0; x < numBanks; x++)
                            {
                                bankNoteArray[x] += ValueTraderArray[j].totalNotes(x);
                            }
                            moneySupply += ValueTraderArray[j].totalMoney();
                            baseMoneySupply += ValueTraderArray[j].totalBaseMoney();
                        }
                        for (int k = 0; k < numSpecInv; k++)
                        {
                            for (int x = 0; x < numBanks; x++)
                            {
                                bankNoteArray[x] += SpecTraderArray[k].totalNotes(x);
                            }
                            moneySupply += SpecTraderArray[k].totalMoney();
                            baseMoneySupply += SpecTraderArray[k].totalBaseMoney();
                        }
                        for (int i = 0; i < BankerArray.Length; i++)
                        {
                            baseMoneySupply += BankerArray[i].totalBaseMoney();
                        }

                        // Print money supply.
                        datalineB = r.ToString() + "," + s.ToString() + ",MoneySupply,";
                        datalineB += moneySupply.ToString() + ",Base=" +
baseMoneySupply.ToString();
                        fileB.WriteLine(datalineB);
                        fileB.Flush();
                        // Print notes held by traders.
                        if (freeBanking)
                        {
                            for (int x = 0; x < numBanks; x++)
                            {
                                datalineB = r.ToString() + "," + s.ToString() + ",Bank" +
x.ToString() + "NotesHeldByTraders,";
                                datalineB += bankNoteArray[x].ToString();
                                fileB.WriteLine(datalineB);
                                fileB.Flush();
                            }
                        }
                    }
                    // Asset pays dividends at the end of each trading
                    // period.
                    if (r >= numNonTradingPeriods)
                    {
                        TheAsset.generate_dividend();
                    }
                    // Banks pay interest at the end of each period.
                    // Ideally, interest would be paid based on the
                    // number of intervals each deposit is held, but
                    // this should suffice for now.
                    for (int c = 0; c < numBanks; c++)
                    {
                        BankerArray[c].payInterest();
                    }
                    // Book dividends and interest, and clean up.
                    savingsInterest = 0.0;
                    dividends = 0.0;
                    for (int i = 0; i < numMomentumInv; i++)
                    {
                        if (numBanks > 0) savingsInterest +=
MomentumTraderArray[i].bookInterest(BankerArray);
                        if (r >= numNonTradingPeriods) dividends +=
MomentumTraderArray[i].book_dividends(TheAsset);
                        MomentumTraderArray[i].periodCleanup();
                    }
                    for (int j = 0; j < numValueInv; j++)
                    {
                        if (numBanks > 0) savingsInterest +=
ValueTraderArray[j].bookInterest(BankerArray);
                        if (r >= numNonTradingPeriods) dividends +=
ValueTraderArray[j].book_dividends(TheAsset);
                        ValueTraderArray[j].periodCleanup();
                    }
                    for (int k = 0; k < numSpecInv; k++)
                    {
                        if (numBanks > 0) savingsInterest +=
SpecTraderArray[k].bookInterest(BankerArray);
                        if (r >= numNonTradingPeriods) dividends +=
SpecTraderArray[k].book_dividends(TheAsset);
                        SpecTraderArray[k].periodCleanup();
                    }
                    for (int l = 0; l < numNonTraders; l++)
                    {
                        savingsInterest += NonTraderArray[l].bookInterest(BankerArray);
                        NonTraderArray[l].periodCleanup();
                    }
                    // Print savings interest.
                    datalineB = r.ToString() + ", ,SavingsInterestThisPeriod,";
                    datalineB += savingsInterest.ToString();
                    fileB.WriteLine(datalineB);
                    fileB.Flush();
                    // Print dividends.
                    datalineB = r.ToString() + ", ,DividendsThisPeriod,";
                    datalineB += dividends.ToString();
                    fileB.WriteLine(datalineB);
                    fileB.Flush();
                    // Print bank profits and nonperforming loans.
                    for (int c = 0; c < numBanks; c++)
                    {
                        datalineB = r.ToString() + ", ,BankProfitsThisPeriod,";
                        datalineB += BankerArray[c].totalProfitsPerPeriod().ToString();
                        datalineB += ",Bank" + c.ToString();
                        fileB.WriteLine(datalineB);
                        fileB.Flush();

                        datalineB = r.ToString() + ", ,NonPerformingLoans,";
                        datalineB += BankerArray[c].totalNonperformingLoans().ToString();
                        datalineB += ",Bank" + c.ToString();
                        fileB.WriteLine(datalineB);
                        fileB.Flush();
                    }

                    // Clear bids and offers.
                    TheAsset.clear_queue();
                }

                // All done. Print out trader earnings, summary,
                // and mean price files.
                for (int i = 0; i < numMomentumInv; i++)
                {
                    datalineE = i.ToString() + ",";
                    datalineE += MomentumTraderArray[i].printBalanceSheet();
                    datalineE += "," + MomentumTraderArray[i].learningFactor.ToString();
                    datalineE += "," + MomentumTraderArray[i].endowClass.ToString();
                    datalineE += ",Momentum";
                    fileE.WriteLine(datalineE);
                    fileE.Flush();
                }
                for (int j = 0; j < numValueInv; j++)
                {
                    datalineE = (numMomentumInv + j).ToString() + ",";
                    datalineE += ValueTraderArray[j].printBalanceSheet();
                    datalineE += "," + (ValueTraderArray[j].learningFactor *
0.05).ToString();
                    datalineE += "," + ValueTraderArray[j].endowClass.ToString();
                    datalineE += ",Value";
                    fileE.WriteLine(datalineE);
                    fileE.Flush();
                }
                for (int k = 0; k < numSpecInv; k++)
                {
                    datalineE = (numValueInv + numMomentumInv + k).ToString() + ",";
                    datalineE += SpecTraderArray[k].printBalanceSheet();
                    datalineE += "," + SpecTraderArray[k].learningFactor.ToString();
                    datalineE += "," + SpecTraderArray[k].endowClass.ToString();
                    datalineE += ",Speculator";
                    fileE.WriteLine(datalineE);
                    fileE.Flush();
                }

                bubbleSummary = "All done! ";
                bubbleSummary += "\r\nTotal Bubble contracts: " + bubbleContracts.ToString();
                for (int i = 0; i < numPeriods; i++)
                {
                    bubbleSummary += "\r\nPeriod " + i.ToString() + " Bubble contracts: " +
bubbleContractsArray[i].ToString() + " out of " + volumeArray[i].ToString();
                    if (volumeArray[i] > 0)
                    {
                        meanPriceArray[i] = meanPriceArray[i] /
System.Convert.ToDouble(volumeArray[i]);
                    }
                    // If volume equals zero, go in and manually
                    // use closing bid and offer.
                    // Should be rare.
                    datalineMP = i.ToString() + "," + meanPriceArray[i].ToString();
                    meanPriceDevArray[i] = meanPriceArray[i] - (TheAsset.averagedividend *
    (numPeriods - i));
                    divValueArray[i] = TheAsset.averagedividend * (numPeriods - i);
                    datalineMP += "," + meanPriceDevArray[i].ToString() + "," +
    divValueArray[i].ToString();
                    datalineMP += "," + volumeArray[i];
                    fileMP.WriteLine(datalineMP);
                    fileMP.Flush();

                    totalVolume += volumeArray[i];
                }
                datalineS = bubbleContracts.ToString() + "," + totalVolume.ToString();
                if (numBanks > 0)
                {
                    double profits = 0.0;
                    int traderBankruptcies = BankerArray[0].totalBankruptcies();
                    bubbleSummary += "\r\nTraderBankruptcies: " +
    traderBankruptcies.ToString();
                    datalineS += "," + traderBankruptcies.ToString();
                    for (int i = 0; i < numBanks; i++)
                    {
                        bubbleSummary += "\r\nBank" + i.ToString() + " Bankrupt: " +
    BankerArray[i].bankrupt.ToString();
                        datalineS += "," + BankerArray[i].bankrupt.ToString();
                        profits = BankerArray[i].returnTotalProfits();
                        bubbleSummary += "\r\nBank" + i.ToString() + " Profits: " +
    profits.ToString();
                        datalineS += "," + profits.ToString();
                    }
                }
                fileS.WriteLine(datalineS);
                fileS.Flush();
                MessageBox.Show(bubbleSummary);

            }
        }
    }
```

APPENDIX C2: SOURCE CODE OF THE SIMULATION SOFTWARE (TRADER.CS)

```
// *****************************************************************
// This software was written and is owned and copyrighted by
// William McBride of George Mason University, for his dissertation.
// July 23, 2010.
// *****************************************************************

//****************************************************
// This file contains two classes: Trader and Asset.
//****************************************************

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Bubbles_and_Banks
{
    //****************************************************
    // Class: Trader
    // This describes all the properties and functions
    // of an asset trader.
    //****************************************************
    public class Trader
    {
        protected double cash;
        protected double cashMinusBids;
        protected Int32 assets;
        protected Int32 assetsMinusOffers;
        protected double tcost;
        protected double bidoffer_incr;
        protected double surplus;
        protected Int32 traderID;
        protected double wtpay;
        protected double wtaccept;
        protected double mybid;
        protected double myoffer;
        protected double supply_price;
        protected double demand_price;
        protected double bestSavingsRate;
        protected double avgSavingsRate;
        protected Int32 bestSavingsBankID;
        protected bool haveSavingsBank;
        protected double optimalPocketCash;
        protected double bestShortLoanRate;
        protected Int32 bestShortLoanBankID;
        protected double bestLongLoanRate;
        protected Int32 bestLongLoanBankID;
        protected Int32 currentSavingsBankID;
        protected Int32 currentShortLoanBankID;
        protected Int32 currentLongLoanBankID;
        protected double currentLongIntPayment;
        protected double currentShortLoan;
        protected Int32 currentShortLoanPeriod;
        protected double currentShortIntPayment;
        protected double currentLongLoan;
        protected Int32 currentLongLoanPeriod;
        protected Int32 maxLoanPeriod;
        protected double savings;
        protected double wtpayShortInterest;
        protected double wtpayLongInterest;
        protected double wtacceptSavingsInterest;
        protected bool marketOrderSell;
        protected string fireSaleLevel;
        protected double sRateElasticity;
        protected double standardShortLoan;
        protected double standardLongLoan;
        protected double[] bankNotesArray;
        protected double baseMoney;
        protected bool freeBanking;
        protected int preferredMoney;
        protected double minPocketCash;
        public int endowClass;
        public bool bankrupt;
        protected bool specBullish;
        protected double specBullReturn;
        protected double specCorrection;
        protected double specMeanReturn;
        protected double specMeanCorrection;
        public double learningFactor;
        protected double meanLearningFactor;
        protected double strategyConfidence;

        public Trader()
        {
            cash = 0.0;
            cashMinusBids = 0.0;
            assets = 0;
            assetsMinusOffers = 0;
            tcost = 0.0;  // May capture risk aversion.
            surplus = 0.0;
            traderID = 0;
            wtpay = 0.0;
            wtaccept = 0.0;
            mybid = 0.0;
            myoffer = 0.0;
            bestSavingsRate = 0.0;
            avgSavingsRate = 0.0;
            bestSavingsBankID = 0;
            haveSavingsBank = false;
            bestShortLoanRate = 0.0;
            bestShortLoanBankID = 0;
            bestLongLoanRate = 0.0;
            bestLongLoanBankID = 0;
            currentSavingsBankID = 100;
            currentShortLoanBankID = 100;
            currentLongLoanBankID = 100;
            currentShortLoan = 0.0;
            currentLongLoan = 0.0;
            currentShortLoanPeriod = 3;
            currentLongLoanPeriod = 5;
            maxLoanPeriod = 4;
            currentShortIntPayment = 0.0;
            currentLongIntPayment = 0.0;
            savings = 0.0;
            // WTP interest should ideally depend on some
            // measure of demand for funds,
            // such as expected return.  Since there is
            // nothing in the model representing
            // expected return over the term of the loan,
            // 2 or 4 periods, it is best to
```

```
            // assume a constant liquidity preference.
            wtpayShortInterest = 0.06;
            wtpayLongInterest = 0.08;
            wtacceptSavingsInterest = 0.0001;
            marketOrderSell = false;
            fireSaleLevel = "";
            preferredMoney = -1; //  by default it is base money.
            bankrupt = false;
            specBullish = true;
            specBullReturn = 0.0;
            specCorrection = 0.0;
            // Mean return that works - 5 or 6% for 25,40,35
            // trader ratio.
            specMeanReturn = 0.06;
            // Likewise, mean correction that works.
            specMeanCorrection = 0.05;
            learningFactor = 0.0;
            meanLearningFactor = 0.5;
            strategyConfidence = 1.0;
        }
        //*******************************************************
        // Trader function: initialize()
        // Initializes trader object.
        //*******************************************************
        public string initialize(Banker[] bankArray, Asset the_asset, double initialCash,
double initialDeposit, double initialLoan, Int32 a1, Int32 ID, double CRE, bool FB, int
endowC, int initialBank, double loanSize)
        {
            string dataline = "";

            this.endowClass = endowC;
            cash = initialCash + initialDeposit;
            baseMoney = cash;
            assets = a1;
            assetsMinusOffers = assets;
            traderID = ID;
            minPocketCash = 0.0;
            optimalPocketCash = minPocketCash;
            sRateElasticity = CRE;
            this.standardLongLoan = loanSize;
            this.standardShortLoan = loanSize;
            freeBanking = FB;
            bankNotesArray = new double[bankArray.Length];
            for (int i = 0; i < bankArray.Length; i++)
            {
                bankNotesArray[i] = 0.0;
            }

            if (bankArray.Length > 0)
            {
                // Disperse traders across banks evenly.
                this.selectSavingsBank(bankArray, the_asset, initialBank);
                // Deposit initial endowment, which is always
                // in base money.
                this.deposit(bankArray, initialDeposit, -1,
this.wtacceptSavingsInterest);
                dataline = "Initialize+OpenSavings+Deposit+";
                // Take out a short term loan from savings bank.
                dataline += borrowShortLoan(bankArray, initialLoan, the_asset, true);
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
            }
            cashMinusBids = cash;

            // Randomize speculator expected return and
            // technical correction.
            specBullReturn = the_asset.rand.NextDouble() * 2.0 * specMeanReturn;
            specCorrection = the_asset.rand.NextDouble() * 2.0 * specMeanCorrection;

            // Randomize learning.
            learningFactor = the_asset.rand.NextDouble() * 2.0 * meanLearningFactor;

            return dataline;
        }
        //****************************************************
        // Trader function: bid()
        // Inserts bid_price into bid queue.
        // If high bid and accepted, then returns
        // "contract."
        //****************************************************
        public string bid (double bid_price, Asset the_asset, bool fromAcceptOffer)
        {
            string dataline;
            int debugVar = 0;

            // First remove all bids.
            the_asset.removeBid(this.traderID);
            this.cashMinusBids = this.cash;

            int bid_index = the_asset.bidlist.Count;

            // Update bid list.  Insert bid in queue even if
            // not the high bid (seems to be
            // what Vernon does).
            if (the_asset.bidlist.Count == 0)
            {
                the_asset.bidlist.Insert(0, bid_price);
                the_asset.bidderIDlist.Insert(0, this.traderID);
                cashMinusBids = cashMinusBids - bid_price;
                if (cashMinusBids < 0.0)
                {
                    // Shouldn't happen.
                    debugVar = 1;
                }
            }
            else if (bid_price <= the_asset.bidlist[the_asset.bidlist.Count - 1])
            {
                the_asset.bidlist.Insert(the_asset.bidlist.Count, bid_price);
                the_asset.bidderIDlist.Insert(the_asset.bidderIDlist.Count,
this.traderID);
                cashMinusBids = cashMinusBids - bid_price;
                if (cashMinusBids < 0.0)
                {
                    // Shouldn't happen.
                    debugVar = 1;
                }
            }
            else for (int i = 0; i < bid_index; i++)
                {
                    if (bid_price > the_asset.bidlist[i])
                    {
                        the_asset.bidlist.Insert(i, bid_price);
                        the_asset.bidderIDlist.Insert(i, this.traderID);
                        cashMinusBids = cashMinusBids - bid_price;
                        if (cashMinusBids < -0.0000001)
                        {
                            // Shouldn't happen.
                            debugVar = 1;
                        }
                        // New as of 5-23-2010: Full implementation
                        // of SSW (1988) requires cancelling
                        // all standing best bids that are bested,
                        // i.e. they don't go into the queue.
                        if ((i == 0) & !fromAcceptOffer)
                        {
                            the_asset.removeBid(the_asset.bidderIDlist[1]);
                        }
                        bid_index = i;
                    }
                }
            // Check if it's a new high bid.
            if (bid_price == the_asset.bidlist[0])
            {
                // Check if there's any offers
                if (the_asset.offerlist.Count > 0)
                {
                    // Check if bid accepted
                    if (the_asset.bidlist[0] >= the_asset.offerlist[0])
                    {
                        dataline = "Contract";
                        // Return Contract, meaning bid accepted,
                        // so book trade.
                        return dataline;
                    }
                }
            }
            else
            {
                // Might happen if a tie.
                dataline = bid_price.ToString() + ",Bid:NotHighest," +
this.traderID.ToString() + "," + printBalanceSheet();
                return dataline;
            }
            dataline = bid_price.ToString() + ",Bid," + this.traderID.ToString() + "," +
printBalanceSheet();
            return dataline;
        }
        //*********************************************
        // Trader function: offer()
        // Inserts offer into queue.
        // If new low offer and accepted,
        // then returns "contract."
        //*********************************************
        public string offer (double offer_price, Asset the_asset, bool fromAcceptBid)
        {
            string dataline;

            // First remove all offers.
            the_asset.removeOffer(this.traderID);
            this.assetsMinusOffers = this.assets;

            int offer_index = the_asset.offerlist.Count;

            // Update offer list.
            if (the_asset.offerlist.Count == 0)
            {
                the_asset.offerlist.Insert(0, offer_price);
                the_asset.offererIDlist.Insert(0, this.traderID);
                assetsMinusOffers = assetsMinusOffers - 1;
            }
            else if (offer_price >= the_asset.offerlist[the_asset.offerlist.Count - 1])
            {
                the_asset.offerlist.Insert(the_asset.offerlist.Count, offer_price);
                the_asset.offererIDlist.Insert(the_asset.offererIDlist.Count,
this.traderID);
                assetsMinusOffers = assetsMinusOffers - 1;
            }
            else for (int i = 0; i < offer_index; i++)
            {
                if (offer_price < the_asset.offerlist[i])
                {
                    the_asset.offerlist.Insert(i,offer_price);
                    the_asset.offererIDlist.Insert(i,this.traderID);
                    assetsMinusOffers = assetsMinusOffers - 1;
                    // New as of 5-23-2010: Full implementation of
                    // SSW (1988) requires cancelling
                    // all standing best offers that are bested,
                    // i.e. they don't go into the queue.
                    if ((i == 0) & !fromAcceptBid)
                    {
                        the_asset.removeOffer(the_asset.offererIDlist[1]);
                    }
                    offer_index = i;
                }
            }
            // Check if it's a new low offer.
            if (offer_price == the_asset.offerlist[0])
            {
                // Check if any bids
                if (the_asset.bidlist.Count > 0)
                {
                    // Check if offer accepted
                    if (the_asset.offerlist[0] <= the_asset.bidlist[0])
                    {
                        dataline = "Contract";
                        // Return Contract, meaning offer accepted,
                        // so book trade.
                        return dataline;
                    }
                }
            }
            else
            {
                // Might happen if tie.
                dataline = offer_price.ToString() + ",Offer:NotLowest," +
this.traderID.ToString() + "," + printBalanceSheet();
                return dataline;
            }
            dataline = offer_price.ToString() + ",Offer," + this.traderID.ToString() +
"," + printBalanceSheet();
            return dataline;
        }
        //*************************************
        // Trader function: accept_bid()
        // Merely offers at the highest bid.
        //*************************************
        public string accept_bid (Asset the_asset)
        {
            return offer(the_asset.bidlist[0], the_asset, true);
        }
        //**************************************
        // Trader function: accept_offer()
        // Merely bids at the highest offer.
        //**************************************
        public string accept_offer (Asset the_asset)
        {
            return bid(the_asset.offerlist[0], the_asset, true);
        }
        //****************************************************
        // Trader function: decideOnTrade()
        // Forms bid/offer based on willing to pay or
        // accept, then submits bid/offer if sufficient
        // cash/assets, otherwise adjusts optimal cash level.
        //*****************************************************
        public string decideOnTrade(Banker[] bankArray, Asset a1)
        {
            string dataline;
            double highBid = 0.0;
            double lowOffer = 0.0;

            // To keep bids from starting around zero, put a
            // floor on bids at half the current
            // price unless wtpay is less.
            // Should make prices less eratic and also match
            // better SSW (1988). 5-23-2010
            double minBid = Math.Min(0.5 * a1.price, wtpay);
            if (minBid < 0.0) minBid = 0.0;

            // Was set to 2, meaning offer is upto 3 times
            // willingness to accept.
            // Currently, set to 1, meaning offer is upto 2
            // times willingess to accept.
            // OK, back to 2. 5-25-2010
            double offerOptimism = 2.0;

            if (a1.offerlist.Count == 0) lowOffer = 1000.0;
            else lowOffer = a1.offerlist[0];

            if (a1.bidlist.Count == 0) highBid = 0.0;
            else highBid = a1.bidlist[0];

            if (wtpay > highBid)
            {
                //Buy if you can

                // If no bids then construct a bid and submit it.
                if (a1.bidlist.Count == 0)
                {
                    if (a1.offerlist.Count == 0)
                    {
                        mybid = minBid + (a1.rand.NextDouble() * (wtpay - minBid));
                    }
                    else
                    {
                        mybid = minBid + (a1.rand.NextDouble() * (Math.Min(wtpay,
a1.offerlist[0]) - minBid));
                    }
                    // Increase optimal cash so as to maintain or
                    // borrow enough to bid next time.
                    this.optimalPocketCash = mybid + this.minPocketCash;
                    if (cash >= mybid) return bid(mybid, a1, false);
                    else
                    {
                        // Withdraw money.
                        if (this.haveSavingsBank & (this.savings >= (mybid - this.cash)))
                        {
                            if (withdraw(bankArray, (mybid - cash), this.preferredMoney))
                            {
                                return bid(mybid, a1, false);
                            }
                        }
                        dataline = "Nothing";
                        return dataline;
                    }
                }
                else if (a1.offerlist.Count == 0)
                {
                    mybid = a1.bidlist[0] + (a1.rand.NextDouble() * (wtpay -
a1.bidlist[0]));
                    // Increase optimal cash so as to maintain or
                    // borrow enough to bid next time.
                    this.optimalPocketCash = mybid + this.minPocketCash;
                    if (cash >= mybid) return bid(mybid, a1, false);
                    else
                    {
                        // Withdraw money.
                        if (this.haveSavingsBank & (this.savings >= (mybid - this.cash)))
                        {
                            if (withdraw(bankArray, (mybid - cash), this.preferredMoney))
                            {
                                return bid(mybid, a1, false);
                            }
                        }
                        dataline = "Nothing";
                        return dataline;
                    }
                }
                //Accept lowest offer if willingness to pay exceeds
                // the lowest offer by more
                //than the bid-offer spread. This should serve to
                // quickly narrow the spread.
                else if ((wtpay - a1.offerlist[0]) > (a1.offerlist[0] - a1.bidlist[0]))
                {
                    // Increase optimal cash so as to maintain or
                    // borrow enough to bid next time.
                    this.optimalPocketCash = a1.offerlist[0] + this.minPocketCash;
                    if (cash >= a1.offerlist[0]) return accept_offer(a1);
                    else
                    {
                        // Withdraw money.
                        if (this.haveSavingsBank & (this.savings >= (a1.offerlist[0] -
this.cash)))
                        {
                            if (withdraw(bankArray, (a1.offerlist[0] - cash),
this.preferredMoney))
                            {
                                return accept_offer(a1);
                            }
                        }
                        dataline = "Nothing";
                        return dataline;
                    }
                }
                // Otherwise contruct a bid and submit it, always
                // higher than the highest bid.
                else if (wtpay > a1.bidlist[0])
                {
                    mybid = a1.bidlist[0] + (a1.rand.NextDouble() * (Math.Min(wtpay,
a1.offerlist[0]) - a1.bidlist[0]));
                    // Increase optimal cash so as to maintain or
                    // borrow enough to bid next time.
                    this.optimalPocketCash = mybid + this.minPocketCash;
                    if (cash >= mybid) return bid(mybid, a1, false);
                    else
                    {
                        // Withdraw money.
                        if (this.haveSavingsBank & (this.savings >= (mybid - this.cash)))
                        {
                            if (withdraw(bankArray, (mybid - cash), this.preferredMoney))
                            {
                                return bid(mybid, a1, false);
                            }
                        }
                        dataline = "Nothing";
                        return dataline;
                    }
                }
            }
            else if (((wtaccept > 0.0) | (a1.bidlist.Count > 0)) & (wtaccept < lowOffer))
            {
                // Sell if you can

                // If no offers then construct a offer and submit it.
                if (a1.offerlist.Count == 0)
                {
                    if (a1.bidlist.Count == 0)
                    {
                        myoffer = wtaccept + a1.rand.NextDouble() * offerOptimism *
wtaccept;
                    }
                    else
                    {
                        myoffer = Math.Max(wtaccept, a1.bidlist[0]) +
a1.rand.NextDouble() * offerOptimism * Math.Max(wtaccept, a1.bidlist[0]);
                    }
                    if (assets > 0) return offer(myoffer, a1, false);
                    else
                    {
                        dataline = "Nothing";
                        return dataline;
                        // can't borrow assets.
                    }
                }
                else if (a1.bidlist.Count == 0)
                {
                    myoffer = a1.offerlist[0] + (a1.rand.NextDouble() * (wtaccept -
a1.offerlist[0]));
                    if (assets > 0) return offer(myoffer, a1, false);
                    else
                    {
                        dataline = "Nothing";
                        return dataline;
                        // can't borrow assets.
                    }
                }
                // Accept highest bid if it exceeds willingness to
                // accept by more
                // than the bid-offer spread. This should serve to
                // quickly narrow the spread.
                // Also accept if market order.
                if (((a1.bidlist[0] - wtaccept) > (a1.offerlist[0] - a1.bidlist[0])) |
marketOrderSell)
                {
                    if (assets > 0) return accept_bid(a1);
                    else
                    {
                        dataline = "Nothing";
                        return dataline;
                        // can't borrow assets.
                    }
                }
                // Otherwise contruct an offer and submit it, always
                // lower than the lowest offer.
                if (a1.offerlist[0] > wtaccept)
                {
                    myoffer = a1.offerlist[0] + (a1.rand.NextDouble() *
(Math.Max(wtaccept, a1.bidlist[0]) - a1.offerlist[0]));
                    if (assets > 0) return offer(myoffer, a1, false);
                    else
                    {
                        dataline = "Nothing";
                        return dataline;
                        // can't borrow assets.
                    }
                }
            }
            dataline = "Nothing";
            return dataline;
        }
        //*********************************************
        // Trader function: book_purchase()
        // Updates buyers cash and assets after a
        // purchase.
        //*********************************************
        public void book_purchase(Asset the_asset)
        {
            double mostBankNotes = 0.0;
            int mostBankNoteIndex = -1;

            // Debug: Surplus should be zero, if not problem.
            this.surplus = the_asset.bidlist[0] - the_asset.offerlist[0];
            if (this.surplus != 0.0)
            {
                this.surplus = 0.0;
            }

            // Update buyer's cash and assets
            this.assets = this.assets + 1;
            this.cash = this.cash - the_asset.bidlist[0];
            // Debug
            if (this.cash < -0.0000001)
            {
                // Shouldn't happen.
                mostBankNotes = 0.0;
            }
            this.optimalPocketCash -= the_asset.bidlist[0];
            if (this.optimalPocketCash < this.minPocketCash)
            {
                this.optimalPocketCash = this.minPocketCash;
            }
            // Assume no distinction between bank notes in trading,
            // pay with the most plentiful.
            // Otherwise, it's just not practical to keep track of
            // money type used in bids and asks.
            // Perhaps this is why there are typically trading
            // accounts, to confirm the quality
            // of money before trading occurs.
            // Also, in terms of discipline, perhaps it's not ideal,
            // but traders still can discipline
            // bankers by withdrawing in base money.

            // Debug: shouldn't need to do all this, just use
            // preferred money.
            if (freeBanking)
            {
                for (int i = 0; i < bankNotesArray.Length; i++)
                {
                    if (this.bankNotesArray[i] > mostBankNotes)
                    {
                        mostBankNotes = this.bankNotesArray[i];
                        mostBankNoteIndex = i;
                    }
                }
                if ((mostBankNotes - the_asset.bidlist[0]) > -0.000001)
                {
                    this.bankNotesArray[mostBankNoteIndex] -= the_asset.bidlist[0];
                }
                else if (mostBankNoteIndex != preferredMoney)
```

```
            {
                // Shouldn't happen if in optimizeCashHoldings()
                // each trader exchanges notes
                // to accumulate his preferred notes.
                this.bankNotesArray[mostBankNoteIndex] -= the_asset.bidlist[0];
            }
            else
            {
                // Also shouldn't happen if in optimizeCashHoldings()
                // each trader exchanges notes
                // to accumulate his preferred notes.
                this.bankNotesArray[mostBankNoteIndex] -= the_asset.bidlist[0];
            }
        }
        else
        {
            this.baseMoney -= the_asset.bidlist[0];
        }
        // Assume all trades are done in preferred money.
        the_asset.buyerMoneyType = this.preferredMoney;

        this.assetsMinusOffers = this.assetsMinusOffers + 1;
    }
    //*********************************************
    // Trader function: book_sale()
    // Update seller's cash and assets upon
    // sale.
    //*********************************************
    public void book_sale(Asset the_asset)
    {
        // Debug: Surplus should be zero, if not problem.
        this.surplus = the_asset.bidlist[0] - the_asset.offerlist[0];
        if (this.surplus != 0.0)
        {
            this.surplus = 0.0;
        }

        // Update seller's cash and assets
        this.assets = this.assets - 1;
        this.cash = this.cash + the_asset.offerlist[0];
        // Assume no distinction between bank notes in trading,
        // pay with the most plentiful.
        // Otherwise, it's just not practical to keep track of
        // money type used in bids and asks.
        // Perhaps this is why there are typically trading
        // accounts, to confirm the quality
        // of money before trading occurs.
        // Also, in terms of discipline, perhaps it's not ideal,
        // but traders still can discipline
        // bankers by withdrawing in base money.
        if (the_asset.buyerMoneyType > -1)
        {
            this.bankNotesArray[the_asset.buyerMoneyType] += the_asset.offerlist[0];
        }
        else
        {
            this.baseMoney += the_asset.offerlist[0];
        }

        this.cashMinusBids = this.cashMinusBids + the_asset.offerlist[0];
    }
    //***************************************
    // Trader function: book_dividends()
    // Update cash to reflect dividend
    // payments.
    //***************************************
```

```
        public double book_dividends(Asset the_asset)
        {
            double dividends = 0.0;

            dividends = this.assets * the_asset.actualdividend;
            this.cash = this.cash + dividends;
            this.cashMinusBids = this.cash;
            // Assume dividends are paid in base money.
            this.baseMoney += dividends;
            this.assetsMinusOffers = this.assets;

            return dividends;
        }
        //************************************************
        // Trader function: selectSavingsBank()
        // Finds the best savings rate, then opens or
        // transfers account to the bank with the best
        // rate.  Transfers only occur with some
        // elasticity, otherwise all traders would
        // switch at once resulting in a bank run of
        // sorts.
        //************************************************
        public string selectSavingsBank(Banker[] bankArray, Asset the_asset, int
assignedBank)
        {
            double transfer = 0.0;
            string dataline;
            int traderBankRatio;
            double percentChangeInP;
            double percentChangeInQ;
            double doubleodds;
            int intodds;
            int randNum;
            int typeOfMoney;

            this.bestSavingsRate = 0.0;
            this.avgSavingsRate = 0.0;

            // Search for best rate.
            for (int i = 0; i < bankArray.Length; i++)
            {
                if (bankArray[i].depositoryRate > this.bestSavingsRate)
                {
                    this.bestSavingsRate = bankArray[i].depositoryRate;
                    this.bestSavingsBankID = i;
                }
                avgSavingsRate += bankArray[i].depositoryRate;
            }
            avgSavingsRate = avgSavingsRate / bankArray.Length;

            if (this.haveSavingsBank == false)
            {
                this.currentSavingsBankID = assignedBank;

                // Assume the preferred bank note is where each agent
                // has a savings account.
                if (freeBanking) this.preferredMoney = this.currentSavingsBankID;
                this.haveSavingsBank = true;

                dataline = "OpenSavings," + transfer.ToString() + ",Success,";
                dataline += currentSavingsBankID.ToString();
                dataline += "," + bankArray[currentSavingsBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                return dataline;
            }
            else if (this.bestSavingsRate !=
bankArray[currentSavingsBankID].depositoryRate)
            {
                // Transfer entire savings balance, but only
                // according to elasticity.
                // Without some degree of inelasticity, the bank
                // looses all customers
                // at once and bankruptcy ensues.
                if (bankArray[currentSavingsBankID].depositoryRate <= 0.0)
                {
                    percentChangeInQ = 0.5;  // limit exodus to 50%
                }
                else
                {
                    percentChangeInP = (this.bestSavingsRate -
bankArray[currentSavingsBankID].depositoryRate) /
bankArray[currentSavingsBankID].depositoryRate;
                    percentChangeInQ = percentChangeInP * this.sRateElasticity;
                    // limit exodus to 50%.
                    if (percentChangeInQ > 0.5) percentChangeInQ = 0.5;
                }
                if (percentChangeInQ < 0.000001) percentChangeInQ = 0.000001;
                doubleodds = 1.0 / percentChangeInQ;
                intodds = (int)Math.Round(doubleodds);
                randNum = the_asset.rand.Next(0, intodds);

                if (randNum == (intodds - 1))
                {
                    transfer = savings;
                    // Assume that in free banking agents withdraw
                    // banknotes, not base money.
                    if (freeBanking)
                    {
                        typeOfMoney = this.currentSavingsBankID;
                    }
                    else
                    {
                        typeOfMoney = -1;
                    }
                    if (withdraw(bankArray, savings, typeOfMoney))
                    {
                        dataline = "TransferSavings," + transfer.ToString() +
",Success,";
                    }
                    else
                    {
                        dataline = "TransferSavings," + transfer.ToString() +
",BankruptBank,";
                    }
                    this.currentSavingsBankID = this.bestSavingsBankID;
                    // Assume the preferred bank note is where each
                    // agent has a savings account.
                    if (freeBanking) this.preferredMoney = this.currentSavingsBankID;
                    deposit(bankArray, transfer, typeOfMoney,
this.wtacceptSavingsInterest);
                    dataline += currentSavingsBankID.ToString();
                    dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                    dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    return dataline;
                }
                else
                {
                    dataline = "MaintainSavings," + transfer.ToString() + ",Success,";
                    dataline += currentSavingsBankID.ToString();
                    dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                    dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    return dataline;
                }
            }
            else
            {
                dataline = "MaintainSavings," + transfer.ToString() + ",Success,";
                dataline += currentSavingsBankID.ToString();
                dataline += "," + bankArray[currentSavingsBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                return dataline;
            }
        }
        //************************************
        // Trader function: totalMoney()
        // Merely totals cash and savings.
        //************************************
        public double totalMoney()
        {
            return (this.cash + this.savings);
        }
        //***************************************
        // Trader function: totalBaseMoney()
        // Merely returns baseMoney.
        //***************************************
        public double totalBaseMoney()
        {
            return (this.baseMoney);
        }
        //***************************************
        // Trader function: totalNotes()
        // Merely returns each kind of note.
        //***************************************
        public double totalNotes(int bankI)
        {
            return this.bankNotesArray[bankI];
        }
        //***************************************
        // Trader function: exchangeMoney()
        // At current savings bank, exchanges
        // all money for preferred money,
        // where preferred money is, in the
        // case of free banking, banknotes from
        // the saving bank, or in the case of
        // central banking, experimental
        // dollars, i.e. base money.  This is
        // only called under free banking.
        //***************************************
        public string exchangeMoney(Banker[] bankArray)
        {
            string dataline = null;
            double amount = 0.0;

            // Assume each agent prefers banknotes from his savings
            // bank.
            if (freeBanking)
            {
                for (int i = -1; i < bankArray.Length; i++)
                {
                    if (i == -1)
                    {
                        amount = this.baseMoney;
                    }
                    else
                    {
                        amount = this.bankNotesArray[i];
                    }
                    if ((amount > 0.0) & (i != this.preferredMoney))
                    {
                        dataline = "Exchange," + amount.ToString();
                        if (!deposit(bankArray, amount, i, -1.0))
                        {
                            // Should not happen
                            dataline += ",DepositFailure,";
                            dataline += this.currentSavingsBankID.ToString();
                            dataline += "," +
bankArray[this.currentSavingsBankID].printBalanceSheet();
                            dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                            return dataline;
                        }
                        if (!withdraw(bankArray, amount, this.preferredMoney))
                        {
                            // Should not happen
                            dataline += ",WithdrawalFailure,";
                            dataline += this.currentSavingsBankID.ToString();
                            dataline += "," +
bankArray[this.currentSavingsBankID].printBalanceSheet();
                            dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                            return dataline;
                        }
                        dataline += ",Success,";
                        dataline += this.currentSavingsBankID.ToString();
                        dataline += "," +
bankArray[this.currentSavingsBankID].printBalanceSheet();
                        dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    }
                }
            }
            return dataline;
        }
        //************************************************************
        // Trader function: optimizeCashHoldings()
        // Optimizes cash holdings, by depositing cash in
        // excess of the optimal cash level or otherwise withdrawing
        // or borrowing cash to achieve the optimal cash level.
        // The optimal level is determined by trading activity, so
        // that if a trader has insufficient cash to support his bid,
        // then he increases his optimal cash level so as to
        // potentially have the cash in the next interval, either
        // from additional borrowing or dividend payments.
        //************************************************************
        public string optimizeCashHoldings(Banker[] bankArray, Asset theAsset, int
periodsLeft)
        {
            double diff;
            string dataline;
            double prospectiveALratio = 0.0;

            // Should reset optimalpocketcash after bid is removed,
            // but deal with it later.
            // this.optimalPocketCash = this.minPocketCash;
            //for (int i; i < theAsset.bidderIDlist.Count; i++)
            //{
            //    if (theAsset.bidderIDlist[i] == this.traderID)
            //    {
```

```
            //        this.optimalPocketCash = theAsset.bidlist[i] + this.minPocketCash;
            //    }
            //}

            // Institute a saving rule.
            if (this.bankrupt)
            {
                diff = this.cash + this.savings;
                if (diff > 0.0)
                {
                    dataline = "BankruptcyPayment," + diff.ToString() +
this.makeBankruptcyPayment(bankArray);

                }
                else
                {
                    diff = 0.0;
                    dataline = "Nothing," + diff.ToString() + ",BankruptTrader,";
                    dataline += currentSavingsBankID.ToString();
                    dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                    dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                }
            }
            else if (this.cash > this.optimalPocketCash)
            {
                // Deposit the difference.
                diff = (this.cash - this.optimalPocketCash);
                dataline = "Deposit," + diff.ToString();
                if (deposit(bankArray, diff, this.preferredMoney,
this.wtacceptSavingsInterest)) dataline += ",Success,";
                else dataline += ",Failure:InterestTooLow,";
                dataline += currentSavingsBankID.ToString();
                dataline += "," + bankArray[currentSavingsBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
            }
            else if (this.cash < this.optimalPocketCash)
            {
                // Withdraw the difference or if insufficient funds
                // withdraw the balance
                // and borrow short and then long.
                if (this.savings > (this.optimalPocketCash - this.cash))
                {
                    diff = this.optimalPocketCash - this.cash;
                    dataline = "Withdrawal," + diff.ToString();
                    if (withdraw(bankArray, diff, this.preferredMoney)) dataline +=
",Success,";
                    else dataline += ",BankruptBank,";
                    dataline += currentSavingsBankID.ToString();
                    dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                    dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                }
                else
                {
                    if (withdraw(bankArray, this.savings, this.preferredMoney)) dataline
= "Withdrawal+";
                    else dataline = "Withdrawal+BankruptBank+";
                    if (periodsLeft <= maxLoanPeriod)
                    {
                        dataline += "LoanAttempt," + this.cash.ToString() + ",TooLate,";
                        dataline += currentSavingsBankID.ToString();
                        dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                        dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    }
                    else if (this.currentShortLoanPeriod > 2)
                    {
                        dataline += borrowShortLoan(bankArray, this.standardShortLoan,
theAsset, false);
                        dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    }
                    else if (this.currentLongLoanPeriod > 4)
                    {
                        dataline += borrowLongLoan(bankArray, this.standardLongLoan,
theAsset, false);
                        dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    }
                    else
                    {
                        prospectiveALratio = (this.cash + this.savings + this.assets *
theAsset.price) / (3.0* this.standardLongLoan);
                        dataline += "LoanAttempt," + this.cash.ToString() +
",2LoansAlready:ProspectiveALratio=" + prospectiveALratio.ToString() + ",";
                        dataline += currentSavingsBankID.ToString();
                        dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
                        dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    }
                }
            }
            else
            {
                diff = 0.0;
                dataline = "Nothing," + diff.ToString() + ",Optimal,";
                dataline += currentSavingsBankID.ToString();
                dataline += "," + bankArray[currentSavingsBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
            }
            return dataline;
        }
        //*************************************************
        // Trader function: deposit()
        // Deposits money in current savings bank.
        //*************************************************
        public bool deposit(Banker[] bankArray, double amount, int moneyType, double wta)
        {
            // Transfer the cash.
            if (this.bestSavingsRate > wta)
            {
                bankArray[this.currentSavingsBankID].acceptDeposit(amount, this.traderID,
moneyType);
                this.cash -= amount;
                this.cashMinusBids -= amount;
                this.savings += amount;
                if (moneyType == -1)
                {
                    baseMoney -= amount;
                }
                else
                {
                    bankNotesArray[moneyType] -= amount;
                }
                return true;
            }
            else
            {
                // Not worth the deposit.
                return false;
            }
        }
        //************************************************
        // Trader function: withdraw()
        // Withdraws money from current savings bank.
        //************************************************
        public bool withdraw(Banker[] bankArray, double amount, int moneyType)
        {
            // Transfer the cash if it's there.
            if (this.savings >= amount)
            {
                if (bankArray[this.currentSavingsBankID].acceptWithdrawal(amount,
this.traderID, moneyType))
                {
                    this.cash += amount;
                    this.cashMinusBids += amount;
                    this.savings -= amount;
                    if (moneyType == -1)
                    {
                        baseMoney += amount;
                    }
                    else
                    {
                        bankNotesArray[moneyType] += amount;
                    }
                    return true;
                }
                else
                {
                    // Bank is bankrupt.
                    return false;
                }
            }
            else
            {
                // Trader is broke.  Should never happen,
                // so long as no
                // request to withdraw occurs
                // that exceeds savings balance.
                return false;
            }
        }
        //************************************************
        // Trader function: borrowShortLoan()
        // Sorts the banks from lowest to highest short
        // loan rate, then requests a short loan from
        // each until loan is granted or there are no
        // more banks.
        //************************************************
        public string borrowShortLoan(Banker[] bankArray, double amount, Asset the_asset,
bool initial)
        {
            this.bestShortLoanRate = 1.0;
            Int32[] bestShortLoanBankIDarray = new Int32[bankArray.Length];
            double lastRate = 0.0;
            int i = 0;
            int j = 0;
            int l = 0;
            int bbIndex = 0;
            string dataline;
            string bankResponse = " ";
            string allResponses = "";
            bool newBank = false;

            // Sort for the best rate.
            for (j = 0; j < bankArray.Length; j++)
            {
                for (i = 0; i < bankArray.Length; i++)
                {
                    if ((bankArray[i].shortLoanRate > lastRate) &
(bankArray[i].shortLoanRate < this.bestShortLoanRate))
                    {

                        this.bestShortLoanRate = bankArray[i].shortLoanRate;
                        bestShortLoanBankIDarray[j] = i;
                    }
                    else if ((bankArray[i].shortLoanRate == lastRate) &
(bankArray[i].shortLoanRate < this.bestShortLoanRate))
                    {
                        if (j > 0)
                        {
                            newBank = true;
                            for (l = 0; l < j; l++)
                            {
                                if (i == bestShortLoanBankIDarray[l])
                                {
                                    newBank = false;
                                }
                            }
                            if (newBank)
                            {
                                this.bestShortLoanRate = bankArray[i].shortLoanRate;
                                bestShortLoanBankIDarray[j] = i;
                            }
                        }
                    }
                }
                lastRate = bestShortLoanRate;
                bestShortLoanRate = 1.0;
            }

            ///debug
            lastRate = 0.0;

            for (int k = 0;k < bankArray.Length;k++)
            {
                bbIndex = bestShortLoanBankIDarray[k];
                // Initially borrow from savings bank.
                if (initial) bbIndex = this.currentSavingsBankID;
                bestShortLoanRate = bankArray[bbIndex].shortLoanRate;

                //// debug
                if (bestShortLoanRate < lastRate)
                {
                    //// MAJOR PROB, shouldn't happen.
                    k = k;
                }
                else
                {
                    lastRate = bestShortLoanRate;
                }

                if (this.bestShortLoanRate < wtpayShortInterest)
                {
                    bankResponse = bankArray[bbIndex].considerLoan(amount, this.cash,
this.savings, this.assets, this.currentLongLoan, this.traderID, the_asset, initial);
```

```
                    allResponses += bankResponse;
                    if (bankResponse == "Yes")
                    {
                        this.currentShortLoanBankID = bbIndex;
                        this.cash += amount;
                        this.cashMinusBids += amount;
                        if (freeBanking) this.bankNotesArray[bbIndex] += amount;
                        else this.baseMoney += amount;
                        this.currentShortLoan = amount;
                        this.currentShortIntPayment = amount * this.bestShortLoanRate;
                        this.currentShortLoanPeriod = 0;
                        dataline = "ShortLoan," + amount.ToString() + ",(Yes)" +
allResponses + "," + bbIndex.ToString();
                        dataline += "," + bankArray[bbIndex].printBalanceSheet();
                        return dataline;
                    }
                }
                else
                {
                    dataline = "ShortLoan," + amount.ToString() + "," + allResponses +
"No:InterestTooHigh," + bbIndex.ToString();
                    dataline += "," + bankArray[bbIndex].printBalanceSheet();
                    return dataline;
                }
            }
            // Print all banks' responses.
            dataline = "ShortLoan," + amount.ToString() + "," + allResponses + "," +
bbIndex.ToString();
            dataline += "," + bankArray[bbIndex].printBalanceSheet();
            return dataline;
        }
        //**************************************************
        // Trader function: borrowLongLoan()
        // Sorts the banks from lowest to highest long
        // loan rate, then requests a long loan from
        // each until loan is granted or there are no
        // more banks.
        //**************************************************
        public string borrowLongLoan(Banker[] bankArray, double amount, Asset the_asset,
bool initial)
        {
            this.bestLongLoanRate = 1.0;
            Int32[] bestLongLoanBankIDarray = new Int32[bankArray.Length];
            double lastRate = 0.0;
            int i = 0;
            int j = 0;
            int l = 0;
            int bbIndex = 0;
            string dataline;
            string bankResponse = " ";
            string allResponses = "";
            bool newBank = false;

            // Sort for the best rate.
            for (j = 0; j < bankArray.Length; j++)
            {
                for (i = 0; i < bankArray.Length; i++)
                {
                    if ((bankArray[i].longLoanRate > lastRate) &
(bankArray[i].longLoanRate < this.bestLongLoanRate))
                    {
                        this.bestLongLoanRate = bankArray[i].longLoanRate;
                        bestLongLoanBankIDarray[j] = i;
                    }
                    else if ((bankArray[i].longLoanRate == lastRate) &
(bankArray[i].longLoanRate < this.bestLongLoanRate))
```

```
                    {
                        if (j > 0)
                        {
                            newBank = true;
                            for (l = 0; l < j; l++)
                            {
                                if (i == bestLongLoanBankIDarray[l])
                                {
                                    newBank = false;
                                }
                            }
                            if (newBank)
                            {
                                this.bestLongLoanRate = bankArray[i].longLoanRate;
                                bestLongLoanBankIDarray[j] = i;
                            }
                        }
                    }
                }
                lastRate = bestLongLoanRate;
                bestLongLoanRate = 1.0;
            }

            ///debug
            lastRate = 0.0;

            for (int k = 0; k < bankArray.Length; k++)
            {
                bbIndex = bestLongLoanBankIDarray[k];
                bestLongLoanRate = bankArray[bbIndex].longLoanRate;

                //// debug
                if (bestLongLoanRate < lastRate)
                {
                    //// MAJOR PROB, shouldn't happen.
                    k = k;
                }
                else
                {
                    lastRate = bestLongLoanRate;
                }

                if (this.bestLongLoanRate < wtpayLongInterest)
                {
                    bankResponse = bankArray[bbIndex].considerLoan(amount, this.cash,
this.savings, this.assets, this.currentShortLoan, this.traderID, the_asset, initial);
                    allResponses += bankResponse;
                    if (bankResponse == "Yes")
                    {
                        this.currentLongLoanBankID = bbIndex;
                        this.cash += amount;
                        this.cashMinusBids += amount;
                        if (freeBanking) this.bankNotesArray[bbIndex] += amount;
                        else this.baseMoney += amount;
                        this.currentLongLoan = amount;
                        this.currentLongIntPayment = amount * this.bestLongLoanRate;
                        this.currentLongLoanPeriod = 0;
                        dataline = "LongLoan," + amount.ToString() + ",(Yes)" +
allResponses + "," + bbIndex.ToString();
                        dataline += "," + bankArray[bbIndex].printBalanceSheet();
                        return dataline;
                    }
                }
                else
                {
                    dataline = "LongLoan," + amount.ToString() + "," + allResponses +
"No:InterestTooHigh," + bbIndex.ToString();
                    dataline += "," + bankArray[bbIndex].printBalanceSheet();
                    return dataline;
                }
            }
            // Print all banks' responses.
            dataline = "LongLoan," + amount.ToString() + "," + allResponses + "," +
bbIndex.ToString();
            dataline += "," + bankArray[bbIndex].printBalanceSheet();
            return dataline;
        }
        //************************************************
        // Trader function: makeShortLoanPayments()
        // Determines the short loan period and then pays
        // interest or principle plus interest
        // accordingly.
        //************************************************
        public string makeShortLoanPayments(Banker[] bankArray)
        {
            string dataline = null;
            double payment;
            int printedBankID = this.currentShortLoanBankID;

            if (this.currentShortLoanPeriod == 0)
            {
                // Actually should be in period 1, since
                // optimizeCashHoldings
                // is called after this.
                this.currentShortLoanPeriod++;
            }
            if (this.currentShortLoanPeriod == 1)
            {
                // Pay interest only.  For simplicity, no
                // proration.  Also,
                // can be considered a fixed cost to the bank.
                payment = this.currentShortIntPayment;
                dataline = "PayShortLoanInterest," + payment.ToString();
                if (this.cash >= payment)
                {
                    bankArray[this.currentShortLoanBankID].acceptInterestPayment(payment,
this.traderID, this.preferredMoney);
                    this.cash -= payment;
                    this.cashMinusBids -= payment;
                    // Assume payments are made in preferred money.
                    if (freeBanking) this.bankNotesArray[this.preferredMoney] -= payment;
                    else this.baseMoney -= payment;
                    dataline += ",1stPaymentPaid,";
                }
                else if (this.savings >= (payment - this.cash))
                {
                    if (withdraw(bankArray, payment - cash, this.preferredMoney))
                    {

bankArray[this.currentShortLoanBankID].acceptInterestPayment(payment, this.traderID,
this.preferredMoney);
                        this.cash -= payment;
                        this.cashMinusBids -= payment;
                        // Assume payments are made in preferred
                        // money.
                        if (freeBanking) this.bankNotesArray[this.preferredMoney] -=
payment;
                        else this.baseMoney -= payment;
                        dataline += ",1stPaymentPaidAfterWithdrawing,";
                    }
                    else
                    {
                        dataline += ",BankruptBank,";
                    }
                }
                else
                {
                    dataline += this.fileBankruptcy(bankArray, true);
                }
                dataline += printedBankID.ToString();
                dataline += "," + bankArray[printedBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();

                this.currentShortLoanPeriod++;
            }
            else if (this.currentShortLoanPeriod == 2)
            {
                // Payoff interest and principle.
                payment = this.currentShortIntPayment + this.currentShortLoan;
                dataline = "PayoffShortLoan," + payment.ToString();
                if (this.cash >= payment)
                {

bankArray[this.currentShortLoanBankID].acceptInterestPayment(this.currentShortIntPayment,
this.traderID, this.preferredMoney);

bankArray[this.currentShortLoanBankID].acceptPrinciplePayment(this.currentShortLoan,
this.traderID, this.preferredMoney);
                    // Assuming 1 loan at a time and paid off in full.
                    this.currentShortLoan = 0.0;
                    // Return to default value.
                    this.currentShortLoanBankID = 100;
                    this.cash -= payment;
                    this.cashMinusBids -= payment;
                    // Assume payments are made in preferred money.
                    if (freeBanking) this.bankNotesArray[this.preferredMoney] -= payment;
                    else this.baseMoney -= payment;
                    dataline += ",LoanPaidOff,";
                }
                else if (this.savings >= (payment - this.cash))
                {
                    if (withdraw(bankArray, payment - cash, this.preferredMoney))
                    {

bankArray[this.currentShortLoanBankID].acceptInterestPayment(this.currentShortIntPayment,
this.traderID, this.preferredMoney);

bankArray[this.currentShortLoanBankID].acceptPrinciplePayment(this.currentShortLoan,
this.traderID, this.preferredMoney);
                        // Assuming 1 loan at a time and paid off in full.
                        this.currentShortLoan = 0.0;
                        // Return to default value.
                        this.currentShortLoanBankID = 100;
                        this.cash -= payment;
                        this.cashMinusBids -= payment;
                        // Assume payments are made in preferred
                        // money.
                        if (freeBanking) this.bankNotesArray[this.preferredMoney] -=
payment;
                        else this.baseMoney -= payment;
                        dataline += ",LoanPaidOffAfterWithdrawing,";
                    }
                    else
                    {
                        dataline += ",BankruptBank,";
                    }
                }
                else
                {
                     dataline += this.fileBankruptcy(bankArray, true);
                }
                dataline += printedBankID.ToString();
                dataline += "," + bankArray[printedBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();

                this.currentShortLoanPeriod++;
            }
            return dataline;
        }
        //***********************************************
        // Trader function: makeLongLoanPayments()
        // Determines the long loan period and then pays
        // interest or principle plus interest
        // accordingly.
        //***********************************************
        public string makeLongLoanPayments(Banker[] bankArray)
        {
            string dataline = null;
            double payment;
            int printedBankID = this.currentLongLoanBankID;

            if (this.currentLongLoanPeriod == 0)
            {
                // Actually should be in period 1,
                // since optimizeCashHoldings
                // is called after this.
                this.currentLongLoanPeriod++;
            }
            if ((this.currentLongLoanPeriod > 0) & (this.currentLongLoanPeriod < 4))
            {
                // Pay interest only.  For simplicity,
                // no proration.  Also,
                // can be considered a fixed cost to the bank.
                payment = this.currentLongIntPayment;
                dataline = "PayLongLoanInterest," + payment.ToString();
                if (this.cash >= payment)
                {
                    bankArray[this.currentLongLoanBankID].acceptInterestPayment(payment,
this.traderID, this.preferredMoney);
                    this.cash -= payment;
                    this.cashMinusBids -= payment;
                    // Assume payments are made in preferred money.
                    if (freeBanking) this.bankNotesArray[this.preferredMoney] -= payment;
                    else this.baseMoney -= payment;
                    if (this.currentLongLoanPeriod == 1) dataline += ",1stPaymentPaid,";
                    else if (this.currentLongLoanPeriod == 2) dataline +=
",2ndPaymentPaid,";
                    else if (this.currentLongLoanPeriod == 3) dataline +=
",3rdPaymentPaid,";
                }
                else if (this.savings >= (payment - this.cash))
                {
                    if (withdraw(bankArray, payment - cash, this.preferredMoney))
                    {

bankArray[this.currentLongLoanBankID].acceptInterestPayment(payment, this.traderID,
this.preferredMoney);
                        this.cash -= payment;
                        this.cashMinusBids -= payment;
                        // Assume payments are made in preferred
                        // money.
                        if (freeBanking) this.bankNotesArray[this.preferredMoney] -= 
payment;
                        else this.baseMoney -= payment;
                        if (this.currentLongLoanPeriod == 1) dataline += 
",1stPaymentPaidAfterWithdrawing,";
                        else if (this.currentLongLoanPeriod == 2) dataline += 
",2ndPaymentPaidAfterWithdrawing,";
                        else if (this.currentLongLoanPeriod == 3) dataline += 
",3rdPaymentPaidAfterWithdrawing,";
                    }
                    else
                    {
                        dataline += ",BankruptBank,";
                    }
                }
                else
                {
                     dataline += this.fileBankruptcy(bankArray, false);
                }
                dataline += printedBankID.ToString();
                dataline += "," + bankArray[printedBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," + 
this.printBalanceSheet();

                this.currentLongLoanPeriod++;
            }
            else if (this.currentLongLoanPeriod == 4)
            {
                // Payoff interest and principle.
                payment = this.currentLongIntPayment + this.currentLongLoan;
                dataline = "PayoffLongLoan," + payment.ToString();
                if (this.cash >= payment)
                {

bankArray[this.currentLongLoanBankID].acceptInterestPayment(this.currentLongIntPayment, 
this.traderID, this.preferredMoney);

bankArray[this.currentLongLoanBankID].acceptPrinciplePayment(this.currentLongLoan, 
this.traderID, this.preferredMoney);
                    // Assuming 1 loan at a time and paid off
                    // in full.
                    this.currentLongLoan = 0.0;
                    // Return to default value.
                    this.currentLongLoanBankID = 100;
                    this.cash -= payment;
                    this.cashMinusBids -= payment;
                    // Assume payments are made in preferred money.
                    if (freeBanking) this.bankNotesArray[this.preferredMoney] -= payment;
                    else this.baseMoney -= payment;
                    dataline += ",LoanPaidOff,";
                }
                else if (this.savings >= (payment - this.cash))
                {
                    if (withdraw(bankArray, payment - cash, this.preferredMoney))
                    {

bankArray[this.currentLongLoanBankID].acceptInterestPayment(this.currentLongIntPayment, 
this.traderID, this.preferredMoney);

bankArray[this.currentLongLoanBankID].acceptPrinciplePayment(this.currentLongLoan, 
this.traderID, this.preferredMoney);
                        // Assuming 1 loan at a time and paid off
                        // in full.
                        this.currentLongLoan = 0.0;
                        // Return to default value.
                        this.currentLongLoanBankID = 100;
                        this.cash -= payment;
                        this.cashMinusBids -= payment;
                        // Assume payments are made in preferred
                        // money.
                        if (freeBanking) this.bankNotesArray[this.preferredMoney] -=
payment;
                        else this.baseMoney -= payment;
                        dataline += ",LoanPaidOffAfterWithdrawing,";
                    }
                    else
                    {
                        dataline += ",BankruptBank,";
                    }
                }
                else
                {
                    dataline += this.fileBankruptcy(bankArray, false);
                }
                dataline += printedBankID.ToString();
                dataline += "," + bankArray[printedBankID].printBalanceSheet();
                dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();

                this.currentLongLoanPeriod++;
            }
            return dataline;
        }
        //**************************************************
        // Trader function: fileBankruptcy()
        // First notifies all banks of bankruptcy, then
        // pays immediate creditor all cash and
        // savings, where the immediate creditor is the
        // bank owed the most.
        //**************************************************
        public string fileBankruptcy(Banker[] bankArray, bool shortLoan)
        {
            string dataline = "";
            int immediateCreditor;
            double liquidity = 0.0;

            this.bankrupt = true;
            for (int i = 0; i < bankArray.Length; i++)
            {
                bankArray[i].notifyBankruptcy(this.traderID);
            }
            currentShortLoanPeriod = 3;
            currentLongLoanPeriod = 5;

            // First pay the immediate creditor what remains
            // in savings,
            // should be less than owed.
            if (withdraw(bankArray, this.savings, this.preferredMoney))
            {
                liquidity = this.cash;
                if (shortLoan)
                {
                    immediateCreditor = this.currentShortLoanBankID;

bankArray[this.currentShortLoanBankID].acceptBankruptcyPayment(this.cash, this.traderID,
this.preferredMoney);
                    this.currentShortLoan -= this.cash;
                    if (this.currentShortLoan <= 0.0)
                    {
                        // Should only happen due to interest
                        // payment.
                        this.currentShortLoan = 0.0;
                    }
                }
                else
                {
                    immediateCreditor = this.currentLongLoanBankID;

bankArray[this.currentLongLoanBankID].acceptBankruptcyPayment(this.cash, this.traderID,
this.preferredMoney);
                    this.currentLongLoan -= this.cash;
                    if (this.currentLongLoan <= 0.0)
                    {
                        // Should only happen due to interest payment.
                        this.currentLongLoan = 0.0;
                    }
                }
                this.cash = 0.0;
                this.cashMinusBids = 0.0;
                // Assume payments are made in preferred money.
                if (freeBanking) this.bankNotesArray[this.preferredMoney] = 0.0;
                else this.baseMoney = 0.0;
                dataline = ",BankruptTrader+" + liquidity.ToString() + "GivenToBanker" +
immediateCreditor.ToString() + ",";
                return dataline;
            }
            else
            {
                dataline = ",BankruptTrader+BankruptBank,";
                return dataline;
            }
        }
        //*************************************************
        // Trader function: makeBankruptcyPayment()
        // Pays immediate creditor all cash and savings,
        // where the immediate creditor is the bank owed
        // the most.
        //*************************************************
        public string makeBankruptcyPayment(Banker[] bankArray)
        {
            string dataline = "";
            int immediateCreditor;
            double liquidity = 0.0;
            bool shortLoan;

            if ((this.currentShortLoan >= this.currentLongLoan) &
(this.currentShortLoanBankID != 100))
            {
                shortLoan = true;
            }
            else if (this.currentLongLoanBankID != 100)
            {
                shortLoan = false;
            }
            else
            {
                // Shouldn't happen.
                shortLoan = true;
            }

            if (this.savings > 0.0)
            {
                if (!withdraw(bankArray, this.savings, this.preferredMoney))
                {
                    dataline = ",BankruptTrader+BankruptBank,";
                    dataline += currentSavingsBankID.ToString();
                    dataline += "," +
bankArray[currentSavingsBankID].printBalanceSheet();
```

```
                    dataline += "," + this.traderID.ToString() + "," +
this.printBalanceSheet();
                    return dataline;
                }
            }
            liquidity = this.cash;
            if (shortLoan)
            {
                immediateCreditor = this.currentShortLoanBankID;
                bankArray[this.currentShortLoanBankID].acceptBankruptcyPayment(this.cash,
this.traderID, this.preferredMoney);
                this.currentShortLoan -= this.cash;
                if (this.currentShortLoan < 0.0) this.currentShortLoan = 0.0;
            }
            else
            {
                immediateCreditor = this.currentLongLoanBankID;
                bankArray[this.currentLongLoanBankID].acceptBankruptcyPayment(this.cash,
this.traderID, this.preferredMoney);
                this.currentLongLoan -= this.cash;
                if (this.currentLongLoan < 0.0) this.currentLongLoan = 0.0;
            }
            this.cash = 0.0;
            this.cashMinusBids = 0.0;
            // Assume payments are made in preferred money.
            if (freeBanking) this.bankNotesArray[this.preferredMoney] = 0.0;
            else this.baseMoney = 0.0;
            dataline = ",BankruptTrader+" + liquidity.ToString() + "GivenToBanker" +
immediateCreditor.ToString() + ",";
            dataline += immediateCreditor.ToString();
            dataline += "," + bankArray[immediateCreditor].printBalanceSheet();
            dataline += "," + this.traderID.ToString() + "," + this.printBalanceSheet();
            return dataline;
        }
        //************************************************
        // Trader function: fireSale()
        // Determines liabilities and bankruptcy status,
        // then accordingly discounts WTP and WTA and/or
        // issues a market order sell.
        //************************************************
        public void fireSale(Asset a1, int intervalsLeftInPeriod)
        {
            marketOrderSell = false;
            fireSaleLevel = "";
            double discount;
            double cashCushion = a1.price + 0.5;
            double liabilities = 0.0;
            double money = this.cashMinusBids + this.savings;

            // debug
            if ((this.cashMinusBids - this.cash) > 0.0000001)
            {
                // MAJOR PROBLEM
                discount = 0.0;
            }

            if (this.currentLongLoanPeriod <= 3)
            {
                liabilities += this.currentLongIntPayment;
            }
            else if (this.currentLongLoanPeriod == 4)
            {
                liabilities += this.currentLongLoan + this.currentLongIntPayment;
            }
            if (this.currentShortLoanPeriod <= 1)
            {
```

```
            liabilities += this.currentShortIntPayment;
        }
        else if (this.currentShortLoanPeriod == 2)
        {
            liabilities += this.currentShortLoan + this.currentShortIntPayment;
        }

        if (this.bankrupt)
        {
            // Essentially, the borrower acts as an agent of
            // the bank now.
            // First remove all bids.
            a1.removeBid(this.traderID);
            this.cashMinusBids = this.cash;

            // Don't try to borrow money next interval, since
            // the banks won't let you anyway.
            this.optimalPocketCash = 0.0;

            discount = 0.8;
            wtpay = 0.0;
            wtaccept = discount * a1.price;
            this.marketOrderSell = true;
            this.fireSaleLevel = "BankruptcyLiquidation";
        }
        else if (liabilities > 0.0)
        {
            if (money < liabilities)
            {
                // First remove all bids.
                a1.removeBid(this.traderID);
                this.cashMinusBids = this.cash;

                // Try to borrow money next interval.
                this.optimalPocketCash = liabilities;

                // Minimum discount factor is 0.4, or 60% off.
                discount = 0.3 + intervalsLeftInPeriod * 0.1;
                if (discount > 1.0) discount = 1.0;

                wtpay = wtpay * discount;
                wtaccept = wtaccept * discount;

                fireSaleLevel = "(" + discount.ToString() + ")Dis+";

                if (intervalsLeftInPeriod < 3)
                {
                    marketOrderSell = true;
                    fireSaleLevel += "MOS+";
                }
            }
            if ((money < (liabilities + cashCushion)) & (intervalsLeftInPeriod < 3))
            {
                // Don't bid
                wtpay = 0.0;
                fireSaleLevel += "Nobid";
            }
        }
    }
    //************************************************
    // Trader function: bookInterest()
    // Adds depository interest to savings.
    //************************************************
    public double bookInterest(Banker[] bankArray)
    {
        double interest = 0.0;
```

```
            if (this.haveSavingsBank)
            {
                interest = this.savings *
bankArray[this.currentSavingsBankID].depositoryRate;
                this.savings += interest;
            }
            return interest;
        }
        //************************************************
        // Trader function: periodCleanup()
        // Resets cashMinusBids and optimalPocketCash.
        //************************************************
        public void periodCleanup()
        {
            this.cashMinusBids = this.cash;
            this.optimalPocketCash = this.minPocketCash;
        }
        //************************************************
        // Trader function: printBalanceSheet()
        // Prints key values.
        //************************************************
        public string printBalanceSheet()
        {
            string dataline;
            dataline = this.cash.ToString();
            if (freeBanking)
            {
                for (int i = 0; i < this.bankNotesArray.Length; i++)
                {
                    dataline += "," + this.bankNotesArray[i].ToString();
                }
            }
            dataline += "," + this.savings.ToString() + ",";
            dataline += this.assets.ToString() + "," + this.currentShortLoan.ToString() +
",";

            dataline += this.currentLongLoan.ToString() + "," + this.fireSaleLevel;
            dataline += "," + this.specBullish.ToString() + "," +
this.specBullReturn.ToString() + "," + this.specCorrection.ToString();
            dataline += "," + this.strategyConfidence.ToString();
            dataline += "," + this.optimalPocketCash.ToString();

            return dataline;
        }
    }

    //************************************************
    // Derived Class: Momentum_trader
    // Derived from Trader class.
    //************************************************
    public class Momentum_trader : Trader
    {
        private double priceDiff;

        public Momentum_trader()
        {
            priceDiff = 0.0;
        }
        //************************************************
        // Momentum Trader function: trade()
        // Forms willingness to pay (WTP) for assets
        // and accept (WTA) based on strategy confidence,
        // fire sale status, and the momentum strategy.
        // The momentum strategy basically linearly
        // extrapolates prices from the last and current
        // prices.
        //************************************************
        public string trade(Banker[] bankArray, Asset a1, int intervalsLeft, int period)
        {
            // Take the first few periods to learn or trust one's
            // own strategy.
            // Likewise, it makes sense to assume others will
            // do the same.

            this.strategyConfidence = this.learningFactor + this.learningFactor * period
* period;
            if (strategyConfidence > 1.0) strategyConfidence = 1.0;

            // Opening price diff doesn't matter.
            if (a1.lastprice <= 0.0) priceDiff = 0.0;
            else priceDiff = a1.price - a1.lastprice;

            // FINAL VERSION 6-17-2010
            wtpay = a1.price + strategyConfidence * priceDiff;
            // TRY DAN'S LEARNING EXP
            // wtpay = a1.price + strategyConfidence;

            if (wtpay <= 0.0) wtpay = 0.0;
            wtaccept = wtpay + tcost;

            // Fire sale if loans are due in the next period.
            fireSale(a1, intervalsLeft);

            return decideOnTrade(bankArray, a1);
        }
    }

    //************************************************
    // Derived Class: Value_trader
    // Derived from Trader class.
    //************************************************
    public class Value_trader : Trader
    {

        public Value_trader()
        {

        }
        //************************************************
        // Value Trader function: trade()
        // Forms willingness to pay (WTP) for assets
        // and accept (WTA) based on strategy confidence,
        // fire sale status, and the value strategy.
        // Under the value strategy, prices are expected
        // to converge to fundamental value.
        //************************************************
        public string trade(Banker[] bankArray, Asset a1, Int32 periods_left, int
intervalsLeft, int period)
        {
            // Take the first few periods to learn or trust
            // one's own strategy.
            // Likewise, it makes sense to assume others will
            // do the same.

            // Extra cognitive load to compute dividend value.
            this.strategyConfidence = 0.05 * this.learningFactor + 0.05 *
this.learningFactor * period * period;
            if (strategyConfidence > 1.0) strategyConfidence = 1.0;

            double dividendValue = a1.averagedividend * periods_left;

            // FINAL VERSION 6-17-2010
            wtpay = a1.price + (dividendValue - a1.price) * strategyConfidence;
            // TRY DAN'S LEARNING EXP
            // wtpay = a1.price + strategyConfidence;

            if (wtpay <= 0.0) wtpay = 0.0;
            wtaccept = wtpay + tcost;

            // Fire sale if loans are due in the next period.
            fireSale(a1, intervalsLeft);

            return decideOnTrade(bankArray, a1);
        }

    }

    //*************************************************
    // Derived Class: Spec_trader
    // Derived from Trader class.
    //*************************************************
    public class Spec_trader : Trader
    {
        private double expectedReturn;
        private bool momentumLike;

        public Spec_trader()
        {
            expectedReturn = 0.1;
            momentumLike = false;
        }
        //*************************************************
        // Speculative Trader function: trade()
        // Forms willingness to pay (WTP) for assets
        // and accept (WTA) based on strategy confidence,
        // fire sale status, and the speculative
        // strategy.
        // Under the speculative strategy, prices are
        // expected to appreciate at the speculative
        // bullish rate of return (specReturn), then
        // when there is a price correction
        // (specCorrection) the strategy switches to
        // a value strategy.
        //*************************************************
        public string trade(Banker[] bankArray, Asset a1, Int32 periods_left, Int32
total_periods, int intervalsLeft, int period)
        {
            double priceAvg = 0.0;

            // Take the first few periods to learn or trust
            // one's own strategy.
            // Likewise, it makes sense to assume others will
            // do the same.

            this.strategyConfidence = this.learningFactor + this.learningFactor * period
* period;
            if (strategyConfidence > 1.0) strategyConfidence = 1.0;

            // Calculate price average.
            int priceAvgLength = 10;
            if (a1.priceHistory.Count < priceAvgLength) priceAvgLength =
a1.priceHistory.Count;
            for (int i = 0; i < priceAvgLength; i++)
            {
                priceAvg += a1.priceHistory[i];
            }
            if (priceAvgLength > 0)
            {
                priceAvg = priceAvg / System.Convert.ToDouble(priceAvgLength);
            }
            else
            {
                priceAvg = 0.0;
            }

            // Strategy A3
            // 1st stage: bubble up, start out bullish.
            if (specBullish)
            {
                if (period > 13)
                {
                    // should be rare.
                    specBullish = false;
                }
                // Switch to bearish on a randomized X% correction,
                // otherwise remain bullish.
                if (a1.price < ((1.0 - this.specCorrection) * priceAvg))
                {
                    specBullish = false;
                    // 2nd stage: Become a value trader.
                    expectedReturn = ((a1.averagedividend * periods_left) - a1.price) /
a1.price;
                }
                else
                {
                    if (momentumLike)
                    {
                        // Act as a less eratic momentum trader.
                        // Not used.
                        if (priceAvg == 0.0)
                        {
                            expectedReturn = 0.0;
                        }
                        else
                        {
                            expectedReturn = (a1.price - priceAvg) / a1.price;
                        }
                    }
                    else
                    {
                        // Randomized positive return.
                        expectedReturn = this.specBullReturn;
                    }
                }
            }
            // 2nd stage: Become a value trader.
            else
            {
                expectedReturn = ((a1.averagedividend * periods_left) - a1.price) /
a1.price;
            }

            // FINAL VERSION 6-17-2010
            wtpay = a1.price + (a1.price * expectedReturn * strategyConfidence);
            // TRY DAN'S LEARNING EXP
            // wtpay = a1.price + strategyConfidence;

            if (wtpay <= 0.0) wtpay = 0.0;
            wtaccept = wtpay + tcost;

            // Fire sale if loans are due in the next period.
            fireSale(a1, intervalsLeft);

            return decideOnTrade(bankArray, a1);
```

```
        }
    }

    //************************************************
    // Derived Class: Nontrader
    // Derived from Trader class.
    // Not used to generate data for dissertation.
    //************************************************
    public class NonTrader : Trader
    {
        private double baseSalary = 2.5;
        private double stochasticSalary = 0.0;
        private double preferredConsumption = 0.0;

        public NonTrader()
        {

        }
        //************************************************
        // Speculative Trader function: work()
        // No trading, but working and consuming instead,
        // on average consuming all earnings.
        //************************************************
        public void work(Asset the_asset)
        {
            stochasticSalary = the_asset.rand.NextDouble() * baseSalary;
            this.cash += baseSalary + stochasticSalary;
            if (freeBanking)
            {
                // For expedience, assume nontraders are paid in
                // banknotes which come from outside the
                // system, say because of old and forgotten notes
                // circulating in another country.
                // If nontraders turn out to be more useful, then
                // should probably introduce an
                // employer who banks and pays in bank notes.
                this.bankNotesArray[this.currentSavingsBankID] += baseSalary +
stochasticSalary;
            }
            else
            {
                this.baseMoney += baseSalary + stochasticSalary;
            }
            this.cashMinusBids = this.cash;
        }
        //************************************************
        // Speculative Trader function: consume()
        // No trading, but working and consuming instead,
        // on average consuming all earnings.
        //************************************************
        public void consume(Asset the_asset)
        {
            preferredConsumption = baseSalary + the_asset.rand.NextDouble() * baseSalary;
            this.cash -= preferredConsumption;
            if (this.cash < 0.0) this.cash = 0.0;
            if (freeBanking)
            {
                this.bankNotesArray[this.currentSavingsBankID] -= preferredConsumption;
                if (this.bankNotesArray[this.currentSavingsBankID] < 0.0)
                {
                    this.bankNotesArray[this.currentSavingsBankID] = 0.0;
                }
            }
            else
            {
                this.baseMoney -= preferredConsumption;
                if (this.baseMoney < 0.0) this.baseMoney = 0.0;
            }
            this.cashMinusBids = this.cash;
        }
    }

    //*************************************************
    // Class: Asset
    // Describes the properties and functions of
    // the tradable asset.  Also, maintains the bid
    // and offer queues and price histories, so in
    // in this sense also represents the asset
    // exchange.
    //*************************************************
    public class Asset
    {
        public double price, actualdividend, averagedividend, lastprice;
        public double[] dividendarray = new double[4];
        // Use this random object in various trader functions.
        public Random rand = new Random();
        public int buyerMoneyType;
        public bool randomDivSelection;
        public int fixedDividend;

        public List<double> bidlist = new List<double>();
        public List<double> offerlist = new List<double>();
        public List<Int32> bidderIDlist = new List<Int32>();
        public List<Int32> offererIDlist = new List<Int32>();
        public List<double> priceHistory = new List<double>();

        public Asset(double p1, double d0, double d1, double d2, double d3, string
dSelection)
        {
            price = p1;
            lastprice = -1.0; // to signify opening price.
            dividendarray[0] = d0;
            dividendarray[1] = d1;
            dividendarray[2] = d2;
            dividendarray[3] = d3;
            averagedividend = (d0 + d1 + d2 + d3)/4.0;
            actualdividend = 0.0;
            buyerMoneyType = -1;
            if (dSelection == "Random")
            {
                randomDivSelection = true;
            }
            else
            {
                randomDivSelection = false;
                fixedDividend = System.Convert.ToInt16(dSelection);
                fixedDividend -= 1;
            }
        }
        //*************************************************
        // Asset function: generate_dividend()
        // Generates either a random or pre-selected
        // dividend.
        //*************************************************
        public void generate_dividend()
        {
            if (randomDivSelection)
            {
                actualdividend = dividendarray[rand.Next(0, 4)];
            }
            else
            {
```

```
        actualdividend = dividendarray[fixedDividend];
    }
}
//************************************************
// Asset function: update_queue()
// Updates price variables and bid and offer
// queues.  Called after a contract is made.
//************************************************
public void update_queue()
{
    this.lastprice = this.price;
    this.price = this.bidlist[0];
    this.priceHistory.Insert(0, this.price);
    if (this.bidlist[0] != this.offerlist[0])
    {
        // SHOULD NOT HAPPEN
        this.price = this.offerlist[0];
    }
    this.bidlist.RemoveAt(0);
    this.bidderIDlist.RemoveAt(0);
    this.offerlist.RemoveAt(0);
    this.offererIDlist.RemoveAt(0);
}
//************************************************
// Asset function: removeBid()
// Removes a trader's bid from the bid queue.
//************************************************
public void removeBid(int traderID)
{
    // Removes all bids by this trader.
    for (int i = 0; i < this.bidderIDlist.Count; i++)
    {
        if (this.bidderIDlist[i] == traderID)
        {
            this.bidderIDlist.RemoveAt(i);
            this.bidlist.RemoveAt(i);
        }
    }
}
//************************************************
// Asset function: removeOffer()
// Removes a trader's offer from the offer queue.
//************************************************
public void removeOffer(int traderID)
{
    // Removes all offers by this trader.
    for (int i = 0; i < this.offererIDlist.Count; i++)
    {
        if (this.offererIDlist[i] == traderID)
        {
            this.offererIDlist.RemoveAt(i);
            this.offerlist.RemoveAt(i);
        }
    }
}
//************************************************
// Asset function: getBidRank()
// Returns a trader's bid rank in the queue.
//************************************************
public int getBidRank(int traderID)
{
    for (int i = 0; i < this.bidderIDlist.Count; i++)
    {
        if (this.bidderIDlist[i] == traderID)
        {
            return i;
```

```
                }
            }
            return -1;
        }
        //**********************************************
        // Asset function: getOfferRank()
        // Returns a trader's offer rank in the queue.
        //**********************************************
        public int getOfferRank(int traderID)
        {
            for (int i = 0; i < this.offererIDlist.Count; i++)
            {
                if (this.offererIDlist[i] == traderID)
                {
                    return i;
                }
            }
            return -1;
        }
        //**********************************************
        // Asset function: clear_queue()
        // Clears bid and offer queues.
        //**********************************************
        public void clear_queue()
        {
            // Keep prices.
            this.bidlist.Clear();
            this.bidderIDlist.Clear();
            this.offerlist.Clear();
            this.offererIDlist.Clear();
        }
    }
}
```

APPENDIX C3: SOURCE CODE OF THE SIMULATION SOFTWARE (BANKER.CS)

```
// *****************************************************************
// This software was written and is owned and copyrighted by
// William McBride of George Mason University, for his dissertation.
// July 23, 2010.
// *****************************************************************

//**********************************************************
// This file contains two classes: Banker and ClearingHouse.
//**********************************************************

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Bubbles_and_Banks
{
    //**********************************************
    // Class: Banker
    // Describes the properties and functions of
    // a banker agent.
    //**********************************************
    public class Banker
    {
        // liabilities
        private double deposits;
        private double issuedNotes;
        private double borrowings;
        private double shareholderEquity;

        // assets
        private double cash;
        private double baseMoney;
        private double[] bankNotesArray;
        private double loans;
        private double nonperformingLoans;
        private double property;
        private double securities;

        public double reserveRatio;
        private double optimalReserveRatio;
        public double avgBorrower_ALratio;
        private double minAvgBorrower_ALratio;
        private double correctionFactor;
        public double depositoryRate;
        public double longLoanRate;
        public double shortLoanRate;
        private double minDepositoryRate;
        private double minLongLoanRate;
        private double minShortLoanRate;
        private bool freeBanking;
        private int bankID;
        private bool limitNoteIssue;
        private double[] clearingsArray;
        private int clearingIndex;
        private bool clearingsArrayFull;
        private double demandForReserves;
        private double initialORR;
        private double q;
        private double profitsPerPeriod;
        private double totalProfits;
        public bool bankrupt;

        public struct customerRecord
        {
            public Int32 customerID;
            public double deposits;
            public double loans;
            public double ALratio;
            public double depositStddev;
            public bool bankrupt;

            public customerRecord(double d, double l)
            {
                customerID = 0;
                deposits = d;
                loans = l;
                ALratio = 0.0;
                depositStddev = 0.0;
                bankrupt = false;
            }
        }

        private customerRecord[] customerRecordArray;

        public Banker()
        {
            deposits = 0.0;
            issuedNotes = 0.0;
            borrowings = 0.0;
            shareholderEquity = 0.0;
            cash = 0.0;
            loans = 0.0;
            nonperformingLoans = 0.0;
            property = 0.0;
            securities = 0.0;
            reserveRatio = 0.0;
            optimalReserveRatio = 0.5;
            // Changed initial rates to be closer together,
            // should result in less profits,
            // but the impetus is to reduce the initial drain
            // on the money supply resulting
            // from banking.  6-6-2010
            depositoryRate = 0.02;
            shortLoanRate = depositoryRate + 0.02;
            longLoanRate = depositoryRate + 0.04;
            minDepositoryRate = 0.0;
            minShortLoanRate = 0.0001;
            minLongLoanRate = 0.0001;
            correctionFactor = 0.005;
            avgBorrower_ALratio = 0.0;
            minAvgBorrower_ALratio = 0.9;
            // Limit note issue to amount of deposits.
            // Not needed for sufficiently high trader/banker
            // ratio, i.e. more than 100.
            limitNoteIssue = false;
            demandForReserves = 0.0;
            profitsPerPeriod = 0.0;
            totalProfits = 0.0;
            bankrupt = false;
        }
        //************************************************
        // Banker function: initialize()
        // Initializes banking parameters.
```

```
        //************************************************
        public void initialize(double c1, Int32 numInv, double iORR, int ID, int
numBanks, bool FB, int clearingSample, double bankQ)
        {
            cash = c1; // usually zero.
            baseMoney = c1;
            bankID = ID;
            // assume max possible customers = all traders.
            customerRecordArray = new customerRecord[numInv];
            for (int j = 0; j < numInv; j++)
            {
                customerRecordArray[j].ALratio = 0.0;
                customerRecordArray[j].customerID = 0;
                customerRecordArray[j].deposits = 0.0;
                customerRecordArray[j].depositStddev = 0.0;
                customerRecordArray[j].loans = 0.0;
            }
            this.initialORR = iORR;
            this.optimalReserveRatio = iORR;
            freeBanking = FB;
            this.q = bankQ;
            bankNotesArray = new double[numBanks];
            for (int i = 0; i < numBanks; i++)
            {
                bankNotesArray[i] = 0.0;
            }
            this.clearingsArray = new double[clearingSample];
            for (int k = 0; k < clearingSample; k++)
            {
                this.clearingsArray[k] = 0.0;
            }
            this.clearingIndex = 0;
            this.clearingsArrayFull = false;
        }
        //************************************************
        // Banker function: setDepositoryRate()
        // Calculates reserve ratio then sets depository
        // interest rate based on difference between
        // optimal reserve ratio and actual reserve
        // ratio.
        //************************************************
        public void setDepositoryRate()
        {
            calculateReserveRatio();
            depositoryRate = depositoryRate + correctionFactor*(optimalReserveRatio -
reserveRatio);
            if (depositoryRate < minDepositoryRate) depositoryRate = minDepositoryRate;
        }
        //************************************************
        // Banker function: setLongLoanRate()
        // Calculates reserve ratio then sets long loan
        // interest rate based on difference between
        // optimal reserve ratio and actual reserve
        // ratio.
        //************************************************
        public void setLongLoanRate()
        {
            calculateReserveRatio();
            longLoanRate = longLoanRate + correctionFactor * (optimalReserveRatio -
reserveRatio);
            if (longLoanRate < minLongLoanRate) longLoanRate = minLongLoanRate;
        }
        //************************************************
        // Banker function: setShortLoanRate()
        // Calculates reserve ratio then sets short loan
        // interest rate based on difference between
        // optimal reserve ratio and actual reserve
        // ratio.
        //**************************************************
        public void setShortLoanRate()
        {
            calculateReserveRatio();
            shortLoanRate = shortLoanRate + correctionFactor * (optimalReserveRatio -
reserveRatio);
            if (shortLoanRate < minShortLoanRate) shortLoanRate = minShortLoanRate;
        }
        //**************************************************
        // Banker function: calculateReserveDemand()
        // Calculates precautionary reserve demand by
        // first checking that the clearing sample is
        // sufficiently large and then calculating the
        // standard deviation of clearings and finally
        // multiplying that by q, where q is a parameter
        // representing banker conservatism.  The value
        // of Q is chosen by maximizing bank profits
        // over a series of experiments.
        //**************************************************
        public string calculateReserveDemand()
        {
            string dataline;
            double var = 0.0;
            double stddev = 0.0;
            double sumSqDev = 0.0;
            double mean = 0.0;
            double sumClearings = 0.0;

            if (!this.clearingsArrayFull | ((this.deposits + this.issuedNotes) <= 0.0))
            {
                this.demandForReserves = 0.0;
            }
            else
            {
                for (int i = 0; i < this.clearingsArray.Length; i++)
                {
                    sumClearings += this.clearingsArray[i];
                }
                mean = sumClearings / this.clearingsArray.Length;
                for (int j = 0; j < this.clearingsArray.Length; j++)
                {
                    sumSqDev += Math.Pow(this.clearingsArray[j] - mean, 2.0);
                }
                var = sumSqDev / (this.clearingsArray.Length - 1);
                stddev = Math.Sqrt(var);
                this.demandForReserves = q * stddev;
            }

            dataline = mean.ToString();
            return dataline;
        }
        //**************************************************
        // Banker function: calculateReserveRatio()
        // Calculates the actual reserve ratio and the
        // optimal reserve ratio.
        //**************************************************
        public void calculateReserveRatio()
        {
            if ((deposits + issuedNotes) <= 0)
            {
                this.reserveRatio = 1.0;
                this.optimalReserveRatio = 1.0;
            }
            else if (this.demandForReserves == 0.0)
            {
                this.reserveRatio = (cash + securities) / (deposits + issuedNotes);
                // Don't calculate optimal reserve ratio, but
                // keep at initial default level.
                this.optimalReserveRatio = this.initialORR;
            }
            else
            {
                this.reserveRatio = (cash + securities) / (deposits + issuedNotes);
                this.optimalReserveRatio = demandForReserves / (this.deposits +
this.issuedNotes);
            }
        }
        //************************************************
        // Banker function: considerLoan()
        // Accepts or denies a loan request based on the
        // trader's asset-to-liability ratio, bankruptcy
        // status, and the banker's reserve ratio.
        //************************************************
        public string considerLoan(double amount, double custCash, double custDeposits,
Int32 custAssets, double custOtherLoans, Int32 custID, Asset the_asset, bool initial)
        {
            double prospectiveALratio;
            double prospectiveReserveRatio;
            string dataline;

            // Establish credit rating.
            prospectiveALratio = (custCash + custDeposits + custAssets * the_asset.price)
/ (custOtherLoans + amount);

            if (this.customerRecordArray[custID].bankrupt)
            {
                dataline = "No:BankruptTrader";
                return dataline;
            }
            else if (initial | (prospectiveALratio > minAvgBorrower_ALratio))
            {
                calculateReserveRatio();
                if (freeBanking)
                {
                    prospectiveReserveRatio = (cash + securities) / (deposits +
issuedNotes + amount);
                }
                else
                {
                    prospectiveReserveRatio = (cash + securities - amount) / (deposits +
issuedNotes);
                }
                if (initial | (prospectiveReserveRatio >= this.optimalReserveRatio))
                {
                    // Not used for sufficiently high trader/banker
                    // ratios, i.e. more than 100.
                    if (freeBanking)
                    {
                        if (limitNoteIssue & ((this.issuedNotes + amount) > (1.0 *
this.deposits)))
                        {
                            dataline = "No:NotesExceed1xDeposits";
                            return dataline;
                        }
                    }

                    customerRecordArray[custID].ALratio = prospectiveALratio;
                    customerRecordArray[custID].loans += amount;
                    // Calculate new overall asset to liability ratio.
                    avgBorrower_ALratio = (avgBorrower_ALratio * loans + 
prospectiveALratio * amount) / (loans + amount);
                    loans += amount;
                    if (freeBanking)
                    {
                        this.issuedNotes += amount;
                    }
                    else
                    {
                        this.cash -= amount;
                        this.baseMoney -= amount;
                        this.clearingsArray[clearingIndex] = amount;
                        this.clearingIndex += 1;
                        if (this.clearingIndex >= this.clearingsArray.Length)
                        {
                            this.clearingsArrayFull = true;
                            this.clearingIndex = 0;
                        }
                    }
                    calculateReserveRatio();
                    dataline = "Yes";
                    return dataline;
                }
                else
                {
                    dataline = "No:LowOnCash";
                    return dataline;
                }
            }
            else
            {
                dataline = "No:BadCredit";
                return dataline;
            }
        }
        //**************************************************
        // Banker function: acceptDeposit()
        // Updates banker records and clearing sample to
        // reflect deposit.
        //**************************************************
        public void acceptDeposit(double customerMoney, Int32 ID, int moneyType)
        {
            customerRecordArray[ID].deposits += customerMoney;
            this.deposits += customerMoney;
            if (moneyType == -1)
            {
                this.baseMoney += customerMoney;
                // Do not count this in free banking,
                // only count reserve clearings.
                if (!freeBanking)
                {
                    this.clearingsArray[clearingIndex] = customerMoney;
                    this.clearingIndex += 1;
                    if (this.clearingIndex >= this.clearingsArray.Length)
                    {
                        this.clearingsArrayFull = true;
                        this.clearingIndex = 0;
                    }
                }
                this.cash += customerMoney;
            }
            else if (moneyType == this.bankID)
            {
                this.issuedNotes -= customerMoney;
            }
            else
            {
                this.bankNotesArray[moneyType] += customerMoney;
                this.cash += customerMoney;
            }

            // Calculate std dev of deposit balance,
            // assuming uniform distribution.
            // Not used.
            customerRecordArray[ID].depositStddev = 2 * customerRecordArray[ID].deposits
* Math.Sqrt(1.0/12.0);
        }
        //************************************************
        // Banker function: acceptWithdrawal()
        // Updates banker records and clearing sample to
        // reflect a withdrawal.
        //************************************************
        public bool acceptWithdrawal(double customerMoney, Int32 ID, int moneyType)
        {
            if (moneyType == -1)
            {
                if (this.baseMoney >= customerMoney)
                {
                    customerRecordArray[ID].deposits -= customerMoney;
                    this.deposits -= customerMoney;
                    this.baseMoney -= customerMoney;
                    this.clearingsArray[clearingIndex] = customerMoney;
                    this.clearingIndex += 1;
                    if (this.clearingIndex >= this.clearingsArray.Length)
                    {
                        this.clearingsArrayFull = true;
                        this.clearingIndex = 0;
                    }
                    this.cash -= customerMoney;
                    // Calculate std dev of deposit balance,
                    // assuming uniform distribution.
                    // Not used.
                    customerRecordArray[ID].depositStddev = 2 *
customerRecordArray[ID].deposits * Math.Sqrt(1.0 / 12.0);
                    return true;
                }
                else
                {
                    fileBankruptcy();
                    return false;
                }
            }
            else if (moneyType == this.bankID)
            {
                customerRecordArray[ID].deposits -= customerMoney;
                this.deposits -= customerMoney;
                this.issuedNotes += customerMoney;
                // Calculate std dev of deposit balance,
                // assuming uniform distribution.
                // Not used.
                customerRecordArray[ID].depositStddev = 2 *
customerRecordArray[ID].deposits * Math.Sqrt(1.0 / 12.0);
                return true;
            }
            else
            {
                // Should never happen
                return false;
            }
        }
        //************************************************
        // Banker function: clearCheck()
        // Implemented before note functions, and is no
        // longer used.
        //************************************************
        public bool clearCheck(double checkAmount, Int32 ID)
        {
            // Doesn't deal with note issue.
            if (this.cash >= checkAmount)
            {
                customerRecordArray[ID].deposits -= checkAmount;
                this.deposits -= checkAmount;
                this.cash -= checkAmount;
                // Calculate std dev of deposit balance,
                // assuming uniform distribution.
                // Not used.
                customerRecordArray[ID].depositStddev = 2 *
customerRecordArray[ID].deposits * Math.Sqrt(1.0 / 12.0);
                return true;
            }
            else
            {
                fileBankruptcy();
                return false;
            }
        }
        //************************************************
        // Banker function: acceptInterestPayment()
        // Updates banker records and clearing sample to
        // reflect an loan interest payment.
        //************************************************
        public void acceptInterestPayment(double amount, Int32 ID, int moneyType)
        {
            if (moneyType == -1)
            {
                this.baseMoney += amount;
                this.clearingsArray[clearingIndex] = amount;
                this.clearingIndex += 1;
                if (this.clearingIndex >= this.clearingsArray.Length)
                {
                    this.clearingsArrayFull = true;
                    this.clearingIndex = 0;
                }
                this.cash += amount;
            }
            else if (moneyType == this.bankID)
            {
                this.issuedNotes -= amount;
            }
            else
            {
                this.bankNotesArray[moneyType] += amount;
                this.cash += amount;
            }
            profitsPerPeriod += amount;
        }
        //************************************************
        // Banker function: acceptPrinciplePayment()
        // Updates banker records and clearing sample to
        // reflect a loan principle repayment.
        //************************************************
        public void acceptPrinciplePayment(double amount, Int32 ID, int moneyType)
        {
            customerRecordArray[ID].loans -= amount;
            this.loans -= amount;
            if (moneyType == -1)
            {
                this.baseMoney += amount;
                this.clearingsArray[clearingIndex] = amount;
                this.clearingIndex += 1;
                if (this.clearingIndex >= this.clearingsArray.Length)
                {
                    this.clearingsArrayFull = true;
                    this.clearingIndex = 0;
                }
                this.cash += amount;
            }
            else if (moneyType == this.bankID)
            {
                this.issuedNotes -= amount;
            }
            else
            {
                this.bankNotesArray[moneyType] += amount;
                this.cash += amount;
            }
        }
        //************************************************
        // Banker function: acceptBankruptcyPayment()
        // Updates banker records and clearing sample to
        // reflect a trader's bankruptcy payment.
        //************************************************
        public void acceptBankruptcyPayment(double amount, int ID, int moneyType)
        {
            if (customerRecordArray[ID].loans > 0.0)
            {
                if (customerRecordArray[ID].loans > amount)
                {
                    customerRecordArray[ID].loans -= amount;
                    this.nonperformingLoans -= amount;
                    this.loans -= amount;
                }
                else
                {
                    this.nonperformingLoans -= customerRecordArray[ID].loans;
                    this.loans -= customerRecordArray[ID].loans;
                    customerRecordArray[ID].loans = 0.0;
                }
            }
            if (moneyType == -1)
            {
                this.baseMoney += amount;
                this.clearingsArray[clearingIndex] = amount;
                this.clearingIndex += 1;
                if (this.clearingIndex >= this.clearingsArray.Length)
                {
                    this.clearingsArrayFull = true;
                    this.clearingIndex = 0;
                }
                this.cash += amount;
            }
            else if (moneyType == this.bankID)
            {
                this.issuedNotes -= amount;
            }
            else
            {
                this.bankNotesArray[moneyType] += amount;
                this.cash += amount;
            }
            profitsPerPeriod += amount;
        }
        //************************************************
        // Banker function: notifyBankruptcy()
        // Updates banker records to reflect a trader's
        // bankruptcy.
        //*************************************************
        public void notifyBankruptcy(int ID)
        {
            customerRecordArray[ID].bankrupt = true;
            this.nonperformingLoans += customerRecordArray[ID].loans;
            profitsPerPeriod -= customerRecordArray[ID].loans;
        }
        //*************************************************
        // Banker function: setDepositoryRate()
        // Updates banker records to reflect periodic
        // depository interest payments to traders.
        //*************************************************
        public void payInterest()
        {
            int i = 0;
            double payment = 0.0;

            for (i = 0; i < customerRecordArray.Length; i++)
            {
                if (customerRecordArray[i].deposits > 0.0)
                {
                    payment = customerRecordArray[i].deposits * this.depositoryRate;
                    customerRecordArray[i].deposits += payment;
                    this.deposits += payment;
                    profitsPerPeriod -= payment;
                }
            }
        }
        //*************************************************
        // Banker function: submitNotes()
        // Updates banker records to reflect sending all
        // other bank's notes to the clearing house.
        //*************************************************
        public double submitNotes(int bankIndex)
        {
            double tempNotes;

            // Make sure not to send your own notes.
            if (this.bankID != bankIndex)
            {
                tempNotes = bankNotesArray[bankIndex];
                bankNotesArray[bankIndex] = 0.0;
                this.cash -= tempNotes;
                return tempNotes;
            }
            else return -1.0;
        }
        //*************************************************
        // Banker function: settle()
        // Updates banker records and clearing sample to
        // reflect settling in base money any outstanding
        // balance with the clearing house.
        //*************************************************
        public bool settle(double clearingHouseBalance, double clearedNotes)
        {
            if (baseMoney > (-1.0*clearingHouseBalance))
            {
                this.baseMoney += clearingHouseBalance;
                this.clearingsArray[clearingIndex] = clearingHouseBalance;
                this.clearingIndex += 1;
                if (this.clearingIndex >= this.clearingsArray.Length)
                {
                    this.clearingsArrayFull = true;
                    this.clearingIndex = 0;
                }
                this.cash += clearingHouseBalance;

                this.issuedNotes -= clearedNotes;

                return true;
            }
            else return false;
        }
        //************************************************
        // Banker function: totalBaseMoney()
        // Returns banker's amount of base money.
        //************************************************
        public double totalBaseMoney()
        {
            return this.baseMoney;
        }
        //************************************************
        // Banker function: totalProfitsPerPeriod()
        // Returns the amount of profits per period, and
        // adds this to total profits.
        //************************************************
        public double totalProfitsPerPeriod()
        {
            this.totalProfits += this.profitsPerPeriod;
            double temp = this.profitsPerPeriod;
            this.profitsPerPeriod = 0.0;
            return temp;
        }
        //************************************************
        // Banker function: returnTotalProfits()
        // Returns the amount of total profits.
        //************************************************
        public double returnTotalProfits()
        {
            return this.totalProfits;
        }
        //************************************************
        // Banker function: totalNonperformingLoans()
        // Returns the amount of nonperforming loans.
        //************************************************
        public double totalNonperformingLoans()
        {
            return this.nonperformingLoans;
        }
        //************************************************
        // Banker function: totalBankruptcies()
        // Returns the total number of customer
        // bankruptcies.
        //************************************************
        public int totalBankruptcies()
        {
            int bankruptcies = 0;

            for (int i = 0; i < this.customerRecordArray.Length; i++)
            {
                if (this.customerRecordArray[i].bankrupt) bankruptcies++;
            }
            return bankruptcies;
        }
        //************************************************
        // Banker function: fileBankruptcy()
        // Sets a bankruptcy flag to true.  Nothing else
        // is done, since the parameters are chosen so
        // that bankruptcy is rare, and when it happens
        // the session ends is counted as a bankrupt
```

```
        // session.
        //************************************************
        public void fileBankruptcy()
        {
            this.bankrupt = true;
            // For now, just count them out, the session
            // is a bust.  Should be rare.
        }
        //************************************************
        // Banker function: printBalanceSheet()
        // Prints out key banking values.
        //************************************************
        public string printBalanceSheet()
        {
            string dataline;

            calculateReserveRatio();

            dataline = this.cash.ToString();
            if (freeBanking)
            {
                for (int i = 0; i < this.bankNotesArray.Length; i++)
                {
                    dataline += "," + this.bankNotesArray[i].ToString();
                }
            }
            dataline += "," + this.loans.ToString() + "," + this.issuedNotes.ToString() +
"," + this.deposits.ToString();
            dataline += "," + this.reserveRatio.ToString() + "," +
this.optimalReserveRatio.ToString();
            dataline += "," + this.demandForReserves.ToString();
            dataline += "," + this.depositoryRate.ToString() + "," +
this.shortLoanRate.ToString();
            dataline += "," + this.longLoanRate.ToString();
            return dataline;
        }
    }

    //************************************************
    // Class: ClearingHouse
    // Describes the properties and functions of
    // the clearing house.
    //************************************************
    public class ClearingHouse
    {
        double baseMoney;
        public ClearingHouse()
        {
            baseMoney = 0.0;
        }
        //************************************************
        // ClearingHouse function: clearNotes()
        // For each bank, sums the total submitted notes
        // and resulting clearinghouse balance to be
        // settled in base money.
        //************************************************
        public string clearNotes(Banker[] bankArray)
        {
            string dataline = "ClearNotes";
            int i = 0;
            int j = 0;
            double tempNotes = 0.0;
            double[] clearedNotesArray = new double[bankArray.Length];
            for (int c = 0; c < bankArray.Length; c++)
            {
                clearedNotesArray[c] = 0.0;
            }
            double[] bankBalanceArray = new double[bankArray.Length];
            for (int k = 0; k < bankArray.Length; k++)
            {
                bankBalanceArray[k] = 0.0;
            }

            for (i = 0; i < bankArray.Length; i++)
            {
                for (j = 0; j < bankArray.Length; j++)
                {
                    if (i != j)
                    {
                        tempNotes = bankArray[i].submitNotes(j);
                        if (tempNotes >= 0.0)
                        {
                            bankBalanceArray[i] += tempNotes;
                            bankBalanceArray[j] -= tempNotes;
                            clearedNotesArray[j] += tempNotes;
                        }
                        else
                        {
                            // NOT SUPPOSED TO HAPPEN
                            dataline += ",";
                            dataline += tempNotes.ToString();
                            dataline += ",Banknotes less than zero,";
                            dataline += i.ToString();
                            dataline += "," + bankArray[i].printBalanceSheet();
                        }
                    }
                }
            }
            for (i = 0; i < bankArray.Length; i++)
            {
                if (bankArray[i].settle(bankBalanceArray[i], clearedNotesArray[i]))
                {
                    baseMoney -= bankBalanceArray[i];
                }
                else
                {
                    dataline += ",";
                    dataline += bankBalanceArray[i].ToString();
                    dataline += ",Bankrupt: Not enough base money to clear,";
                    dataline += i.ToString();
                    dataline += "," + bankArray[i].printBalanceSheet();
                    bankArray[i].fileBankruptcy();
                }
            }
            if (baseMoney < -0.0000001)
            {
                // Problem: ClearingHouse should not have to go
                // into debt.
                dataline += ",";
                dataline += baseMoney.ToString();
                dataline += ",ClearingHouse balance negative";
            }
            else
            {
                dataline += ",";
                dataline += baseMoney.ToString();
                dataline += ",Success";
            }
            return dataline;
```

APPENDIX C4: SOURCE CODE OF THE SIMULATION SOFTWARE (FORM1.DESIGNER.CS)

```
// *****************************************************************
// This software was written and is owned and copyrighted by
// William McBride of George Mason University, for his dissertation.
// July 23, 2010.
// *****************************************************************

// This file was generated by Microsoft Visual Studio from
// using the graphical user interface designer.

namespace Bubbles_and_Banks
{
    partial class FormMainConf
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed;
otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.labelNumBanks = new System.Windows.Forms.Label();
            this.textBoxNumBanks = new System.Windows.Forms.TextBox();
            this.labelNumInv = new System.Windows.Forms.Label();
            this.textBoxNumInv = new System.Windows.Forms.TextBox();
            this.labelPercentMom = new System.Windows.Forms.Label();
            this.textBoxPercentMom = new System.Windows.Forms.TextBox();
            this.labelPercentValue = new System.Windows.Forms.Label();
            this.textBoxPercentValue = new System.Windows.Forms.TextBox();
            this.labelPercentSpec = new System.Windows.Forms.Label();
            this.textBoxPercentSpec = new System.Windows.Forms.TextBox();
            this.labelInvC1cash = new System.Windows.Forms.Label();
            this.textBoxC1money = new System.Windows.Forms.TextBox();
            this.labelInvC1assets = new System.Windows.Forms.Label();
            this.textBoxC1assets = new System.Windows.Forms.TextBox();
            this.labelBankCashEndow = new System.Windows.Forms.Label();
            this.textBoxBankCashEndow = new System.Windows.Forms.TextBox();
            this.labelPeriods = new System.Windows.Forms.Label();
            this.textBoxPeriods = new System.Windows.Forms.TextBox();
            this.labelIntsPerPeriod = new System.Windows.Forms.Label();
            this.textBoxIntPerPeriod = new System.Windows.Forms.TextBox();
            this.labelDiv1 = new System.Windows.Forms.Label();
            this.textBoxDiv1 = new System.Windows.Forms.TextBox();
            this.labelDiv2 = new System.Windows.Forms.Label();
            this.textBoxDiv2 = new System.Windows.Forms.TextBox();
            this.labelDiv3 = new System.Windows.Forms.Label();
            this.textBoxDiv3 = new System.Windows.Forms.TextBox();
            this.labelDiv4 = new System.Windows.Forms.Label();
            this.textBoxDiv4 = new System.Windows.Forms.TextBox();
            this.labelSessions = new System.Windows.Forms.Label();
            this.textBoxSessions = new System.Windows.Forms.TextBox();
            this.buttonRun = new System.Windows.Forms.Button();
            this.labelAssetPrice = new System.Windows.Forms.Label();
            this.textBoxAssetPrice = new System.Windows.Forms.TextBox();
            this.textBoxSavingsRateElasticity = new System.Windows.Forms.TextBox();
            this.labelSavingsRateElasticity = new System.Windows.Forms.Label();
            this.groupBoxBanking = new System.Windows.Forms.GroupBox();
            this.textBoxQ = new System.Windows.Forms.TextBox();
            this.labelQ = new System.Windows.Forms.Label();
            this.textBoxLoanSize = new System.Windows.Forms.TextBox();
            this.labelLoanSize = new System.Windows.Forms.Label();
            this.radioButtonFB = new System.Windows.Forms.RadioButton();
            this.textBoxReserveRatio = new System.Windows.Forms.TextBox();
            this.labelReserveRatio = new System.Windows.Forms.Label();
            this.textBoxNontraders = new System.Windows.Forms.TextBox();
            this.labelNontraders = new System.Windows.Forms.Label();
            this.groupBoxTrading = new System.Windows.Forms.GroupBox();
            this.labelDividendSelection = new System.Windows.Forms.Label();
            this.comboBoxFixedDividend = new System.Windows.Forms.ComboBox();
            this.textBoxC3assets = new System.Windows.Forms.TextBox();
            this.labelC3assets = new System.Windows.Forms.Label();
            this.textBoxC3money = new System.Windows.Forms.TextBox();
            this.labelC3cash = new System.Windows.Forms.Label();
            this.textBoxC2assets = new System.Windows.Forms.TextBox();
            this.labelC2assets = new System.Windows.Forms.Label();
            this.textBoxC2money = new System.Windows.Forms.TextBox();
            this.labelC2cash = new System.Windows.Forms.Label();
            this.groupBoxGeneral = new System.Windows.Forms.GroupBox();
            this.labelNonTradingPeriods = new System.Windows.Forms.Label();
            this.textBoxNonTradingPeriods = new System.Windows.Forms.TextBox();
            this.groupBoxBanking.SuspendLayout();
            this.groupBoxTrading.SuspendLayout();
            this.groupBoxGeneral.SuspendLayout();
            this.SuspendLayout();
            //
            // labelNumBanks
            //
            this.labelNumBanks.AutoSize = true;
            this.labelNumBanks.Location = new System.Drawing.Point(169, 36);
            this.labelNumBanks.Name = "labelNumBanks";
            this.labelNumBanks.Size = new System.Drawing.Size(92, 13);
            this.labelNumBanks.TabIndex = 0;
            this.labelNumBanks.Text = "Number of Banks:";
            //
            // textBoxNumBanks
            //
            this.textBoxNumBanks.Location = new System.Drawing.Point(265, 33);
            this.textBoxNumBanks.Name = "textBoxNumBanks";
            this.textBoxNumBanks.Size = new System.Drawing.Size(39, 20);
            this.textBoxNumBanks.TabIndex = 1;
            this.textBoxNumBanks.Text = "0";
            //
            // labelNumInv
            //
            this.labelNumInv.AutoSize = true;
            this.labelNumInv.Location = new System.Drawing.Point(117, 38);
            this.labelNumInv.Name = "labelNumInv";
            this.labelNumInv.Size = new System.Drawing.Size(98, 13);
            this.labelNumInv.TabIndex = 2;
            this.labelNumInv.Text = "Number of Traders:";
            //
            // textBoxNumInv
            //
            this.textBoxNumInv.Location = new System.Drawing.Point(222, 35);
            this.textBoxNumInv.Name = "textBoxNumInv";
            this.textBoxNumInv.Size = new System.Drawing.Size(36, 20);
            this.textBoxNumInv.TabIndex = 3;
            this.textBoxNumInv.Text = "12";
            //
            // labelPercentMom
            //
            this.labelPercentMom.AutoSize = true;
            this.labelPercentMom.Location = new System.Drawing.Point(74, 64);
            this.labelPercentMom.Name = "labelPercentMom";
            this.labelPercentMom.Size = new System.Drawing.Size(141, 13);
            this.labelPercentMom.TabIndex = 4;
            this.labelPercentMom.Text = "Percent Momentum Traders:";
            //
            // textBoxPercentMom
            //
            this.textBoxPercentMom.Location = new System.Drawing.Point(222, 61);
            this.textBoxPercentMom.Name = "textBoxPercentMom";
            this.textBoxPercentMom.Size = new System.Drawing.Size(36, 20);
            this.textBoxPercentMom.TabIndex = 5;
            this.textBoxPercentMom.Text = "25";
            //
            // labelPercentValue
            //
            this.labelPercentValue.AutoSize = true;
            this.labelPercentValue.Location = new System.Drawing.Point(99, 90);
            this.labelPercentValue.Name = "labelPercentValue";
            this.labelPercentValue.Size = new System.Drawing.Size(116, 13);
            this.labelPercentValue.TabIndex = 6;
            this.labelPercentValue.Text = "Percent Value Traders:";
            //
            // textBoxPercentValue
            //
            this.textBoxPercentValue.Location = new System.Drawing.Point(222, 87);
            this.textBoxPercentValue.Name = "textBoxPercentValue";
            this.textBoxPercentValue.Size = new System.Drawing.Size(35, 20);
            this.textBoxPercentValue.TabIndex = 7;
            this.textBoxPercentValue.Text = "40";
            //
            // labelPercentSpec
            //
            this.labelPercentSpec.AutoSize = true;
            this.labelPercentSpec.Location = new System.Drawing.Point(70, 116);
            this.labelPercentSpec.Name = "labelPercentSpec";
            this.labelPercentSpec.Size = new System.Drawing.Size(145, 13);
            this.labelPercentSpec.TabIndex = 8;
            this.labelPercentSpec.Text = "Percent Speculative Traders:";
            //
            // textBoxPercentSpec
            //
            this.textBoxPercentSpec.Location = new System.Drawing.Point(222, 113);
            this.textBoxPercentSpec.Name = "textBoxPercentSpec";
            this.textBoxPercentSpec.Size = new System.Drawing.Size(35, 20);
            this.textBoxPercentSpec.TabIndex = 9;
            this.textBoxPercentSpec.Text = "35";
            //
            // labelInvC1cash
```

```
//
this.labelInvC1cash.AutoSize = true;
this.labelInvC1cash.Location = new System.Drawing.Point(28, 142);
this.labelInvC1cash.Name = "labelInvC1cash";
this.labelInvC1cash.Size = new System.Drawing.Size(187, 13);
this.labelInvC1cash.TabIndex = 10;
this.labelInvC1cash.Text = "Class I Money Endowment per Trader:";
//
// textBoxC1money
//
this.textBoxC1money.Location = new System.Drawing.Point(222, 139);
this.textBoxC1money.Name = "textBoxC1money";
this.textBoxC1money.Size = new System.Drawing.Size(35, 20);
this.textBoxC1money.TabIndex = 11;
this.textBoxC1money.Text = "2.25";
//
// labelInvC1assets
//
this.labelInvC1assets.AutoSize = true;
this.labelInvC1assets.Location = new System.Drawing.Point(34, 168);
this.labelInvC1assets.Name = "labelInvC1assets";
this.labelInvC1assets.Size = new System.Drawing.Size(181, 13);
this.labelInvC1assets.TabIndex = 12;
this.labelInvC1assets.Text = "Class I Asset Endowment per Trader:";
//
// textBoxC1assets
//
this.textBoxC1assets.Location = new System.Drawing.Point(222, 165);
this.textBoxC1assets.Name = "textBoxC1assets";
this.textBoxC1assets.Size = new System.Drawing.Size(35, 20);
this.textBoxC1assets.TabIndex = 13;
this.textBoxC1assets.Text = "3";
//
// labelBankCashEndow
//
this.labelBankCashEndow.AutoSize = true;
this.labelBankCashEndow.Location = new System.Drawing.Point(121, 64);
this.labelBankCashEndow.Name = "labelBankCashEndow";
this.labelBankCashEndow.Size = new System.Drawing.Size(139, 13);
this.labelBankCashEndow.TabIndex = 14;
this.labelBankCashEndow.Text = "Cash Endowment per Bank:";
//
// textBoxBankCashEndow
//
this.textBoxBankCashEndow.Location = new System.Drawing.Point(265, 59);
this.textBoxBankCashEndow.Name = "textBoxBankCashEndow";
this.textBoxBankCashEndow.Size = new System.Drawing.Size(38, 20);
this.textBoxBankCashEndow.TabIndex = 15;
this.textBoxBankCashEndow.Text = "0";
//
// labelPeriods
//
this.labelPeriods.AutoSize = true;
this.labelPeriods.Location = new System.Drawing.Point(214, 29);
this.labelPeriods.Name = "labelPeriods";
this.labelPeriods.Size = new System.Drawing.Size(45, 13);
this.labelPeriods.TabIndex = 16;
this.labelPeriods.Text = "Periods:";
//
// textBoxPeriods
//
this.textBoxPeriods.Location = new System.Drawing.Point(268, 26);
this.textBoxPeriods.Name = "textBoxPeriods";
this.textBoxPeriods.Size = new System.Drawing.Size(35, 20);
this.textBoxPeriods.TabIndex = 17;
this.textBoxPeriods.Text = "15";
```

```
//
// labelIntsPerPeriod
//
this.labelIntsPerPeriod.AutoSize = true;
this.labelIntsPerPeriod.Location = new System.Drawing.Point(157, 55);
this.labelIntsPerPeriod.Name = "labelIntsPerPeriod";
this.labelIntsPerPeriod.Size = new System.Drawing.Size(101, 13);
this.labelIntsPerPeriod.TabIndex = 18;
this.labelIntsPerPeriod.Text = "Intervals per Period:";
//
// textBoxIntPerPeriod
//
this.textBoxIntPerPeriod.Location = new System.Drawing.Point(268, 52);
this.textBoxIntPerPeriod.Name = "textBoxIntPerPeriod";
this.textBoxIntPerPeriod.Size = new System.Drawing.Size(35, 20);
this.textBoxIntPerPeriod.TabIndex = 19;
this.textBoxIntPerPeriod.Text = "10";
//
// labelDiv1
//
this.labelDiv1.AutoSize = true;
this.labelDiv1.Location = new System.Drawing.Point(84, 333);
this.labelDiv1.Name = "labelDiv1";
this.labelDiv1.Size = new System.Drawing.Size(131, 13);
this.labelDiv1.TabIndex = 20;
this.labelDiv1.Text = "Dividend 1, cents (p=1/4):";
//
// textBoxDiv1
//
this.textBoxDiv1.Location = new System.Drawing.Point(222, 330);
this.textBoxDiv1.Name = "textBoxDiv1";
this.textBoxDiv1.Size = new System.Drawing.Size(36, 20);
this.textBoxDiv1.TabIndex = 21;
this.textBoxDiv1.Text = "0";
//
// labelDiv2
//
this.labelDiv2.AutoSize = true;
this.labelDiv2.Location = new System.Drawing.Point(84, 359);
this.labelDiv2.Name = "labelDiv2";
this.labelDiv2.Size = new System.Drawing.Size(131, 13);
this.labelDiv2.TabIndex = 22;
this.labelDiv2.Text = "Dividend 2, cents (p=1/4):";
//
// textBoxDiv2
//
this.textBoxDiv2.Location = new System.Drawing.Point(222, 356);
this.textBoxDiv2.Name = "textBoxDiv2";
this.textBoxDiv2.Size = new System.Drawing.Size(36, 20);
this.textBoxDiv2.TabIndex = 23;
this.textBoxDiv2.Text = "8";
//
// labelDiv3
//
this.labelDiv3.AutoSize = true;
this.labelDiv3.Location = new System.Drawing.Point(84, 386);
this.labelDiv3.Name = "labelDiv3";
this.labelDiv3.Size = new System.Drawing.Size(131, 13);
this.labelDiv3.TabIndex = 24;
this.labelDiv3.Text = "Dividend 3, cents (p=1/4):";
//
// textBoxDiv3
//
this.textBoxDiv3.Location = new System.Drawing.Point(223, 383);
this.textBoxDiv3.Name = "textBoxDiv3";
this.textBoxDiv3.Size = new System.Drawing.Size(36, 20);
this.textBoxDiv3.TabIndex = 25;
this.textBoxDiv3.Text = "28";
//
// labelDiv4
//
this.labelDiv4.AutoSize = true;
this.labelDiv4.Location = new System.Drawing.Point(85, 412);
this.labelDiv4.Name = "labelDiv4";
this.labelDiv4.Size = new System.Drawing.Size(131, 13);
this.labelDiv4.TabIndex = 26;
this.labelDiv4.Text = "Dividend 4, cents (p=1/4):";
//
// textBoxDiv4
//
this.textBoxDiv4.Location = new System.Drawing.Point(222, 409);
this.textBoxDiv4.Name = "textBoxDiv4";
this.textBoxDiv4.Size = new System.Drawing.Size(36, 20);
this.textBoxDiv4.TabIndex = 27;
this.textBoxDiv4.Text = "60";
//
// labelSessions
//
this.labelSessions.AutoSize = true;
this.labelSessions.Location = new System.Drawing.Point(209, 81);
this.labelSessions.Name = "labelSessions";
this.labelSessions.Size = new System.Drawing.Size(52, 13);
this.labelSessions.TabIndex = 28;
this.labelSessions.Text = "Sessions:";
//
// textBoxSessions
//
this.textBoxSessions.Location = new System.Drawing.Point(268, 78);
this.textBoxSessions.Name = "textBoxSessions";
this.textBoxSessions.Size = new System.Drawing.Size(35, 20);
this.textBoxSessions.TabIndex = 29;
this.textBoxSessions.Text = "1";
//
// buttonRun
//
this.buttonRun.Location = new System.Drawing.Point(558, 479);
this.buttonRun.Name = "buttonRun";
this.buttonRun.Size = new System.Drawing.Size(75, 23);
this.buttonRun.TabIndex = 30;
this.buttonRun.Text = "Run";
this.buttonRun.UseVisualStyleBackColor = true;
this.buttonRun.Click += new System.EventHandler(this.buttonRun_Click);
//
// labelAssetPrice
//
this.labelAssetPrice.AutoSize = true;
this.labelAssetPrice.Location = new System.Drawing.Point(124, 305);
this.labelAssetPrice.Name = "labelAssetPrice";
this.labelAssetPrice.Size = new System.Drawing.Size(90, 13);
this.labelAssetPrice.TabIndex = 31;
this.labelAssetPrice.Text = "Initial Asset Price:";
//
// textBoxAssetPrice
//
this.textBoxAssetPrice.Location = new System.Drawing.Point(223, 302);
this.textBoxAssetPrice.Name = "textBoxAssetPrice";
this.textBoxAssetPrice.Size = new System.Drawing.Size(35, 20);
this.textBoxAssetPrice.TabIndex = 32;
this.textBoxAssetPrice.Text = "0.0";
//
// textBoxSavingsRateElasticity
//
```

```
            this.textBoxSavingsRateElasticity.Location = new System.Drawing.Point(265, 
85);
            this.textBoxSavingsRateElasticity.Name = "textBoxSavingsRateElasticity";
            this.textBoxSavingsRateElasticity.Size = new System.Drawing.Size(38, 20);
            this.textBoxSavingsRateElasticity.TabIndex = 33;
            this.textBoxSavingsRateElasticity.Text = "0.7";
            // 
            // labelSavingsRateElasticity
            // 
            this.labelSavingsRateElasticity.AutoSize = true;
            this.labelSavingsRateElasticity.Location = new System.Drawing.Point(141, 88);
            this.labelSavingsRateElasticity.Name = "labelSavingsRateElasticity";
            this.labelSavingsRateElasticity.Size = new System.Drawing.Size(118, 13);
            this.labelSavingsRateElasticity.TabIndex = 34;
            this.labelSavingsRateElasticity.Text = "Savings Rate Elasticity:";
            // 
            // groupBoxBanking
            // 
            this.groupBoxBanking.Controls.Add(this.textBoxQ);
            this.groupBoxBanking.Controls.Add(this.labelQ);
            this.groupBoxBanking.Controls.Add(this.textBoxLoanSize);
            this.groupBoxBanking.Controls.Add(this.labelLoanSize);
            this.groupBoxBanking.Controls.Add(this.radioButtonFB);
            this.groupBoxBanking.Controls.Add(this.textBoxReserveRatio);
            this.groupBoxBanking.Controls.Add(this.labelReserveRatio);
            this.groupBoxBanking.Controls.Add(this.labelSavingsRateElasticity);
            this.groupBoxBanking.Controls.Add(this.textBoxSavingsRateElasticity);
            this.groupBoxBanking.Controls.Add(this.textBoxNontraders);
            this.groupBoxBanking.Controls.Add(this.labelNontraders);
            this.groupBoxBanking.Controls.Add(this.textBoxBankCashEndow);
            this.groupBoxBanking.Controls.Add(this.textBoxNumBanks);
            this.groupBoxBanking.Controls.Add(this.labelNumBanks);
            this.groupBoxBanking.Controls.Add(this.labelBankCashEndow);
            this.groupBoxBanking.Location = new System.Drawing.Point(331, 22);
            this.groupBoxBanking.Name = "groupBoxBanking";
            this.groupBoxBanking.Size = new System.Drawing.Size(328, 278);
            this.groupBoxBanking.TabIndex = 35;
            this.groupBoxBanking.TabStop = false;
            this.groupBoxBanking.Text = "Banking";
            // 
            // textBoxQ
            // 
            this.textBoxQ.Location = new System.Drawing.Point(266, 197);
            this.textBoxQ.Name = "textBoxQ";
            this.textBoxQ.Size = new System.Drawing.Size(36, 20);
            this.textBoxQ.TabIndex = 42;
            this.textBoxQ.Text = "40.0";
            // 
            // labelQ
            // 
            this.labelQ.AutoSize = true;
            this.labelQ.Location = new System.Drawing.Point(131, 200);
            this.labelQ.Name = "labelQ";
            this.labelQ.Size = new System.Drawing.Size(128, 13);
            this.labelQ.TabIndex = 41;
            this.labelQ.Text = "Q (Banker Convervatism):";
            // 
            // textBoxLoanSize
            // 
            this.textBoxLoanSize.Location = new System.Drawing.Point(265, 168);
            this.textBoxLoanSize.Name = "textBoxLoanSize";
            this.textBoxLoanSize.Size = new System.Drawing.Size(37, 20);
            this.textBoxLoanSize.TabIndex = 40;
            this.textBoxLoanSize.Text = "5.0";
            // 
            // labelLoanSize
```

```
//
this.labelLoanSize.AutoSize = true;
this.labelLoanSize.Location = new System.Drawing.Point(202, 172);
this.labelLoanSize.Name = "labelLoanSize";
this.labelLoanSize.Size = new System.Drawing.Size(57, 13);
this.labelLoanSize.TabIndex = 39;
this.labelLoanSize.Text = "Loan Size:";
//
// radioButtonFB
//
this.radioButtonFB.AutoSize = true;
this.radioButtonFB.Location = new System.Drawing.Point(124, 234);
this.radioButtonFB.Name = "radioButtonFB";
this.radioButtonFB.RightToLeft = System.Windows.Forms.RightToLeft.Yes;
this.radioButtonFB.Size = new System.Drawing.Size(88, 17);
this.radioButtonFB.TabIndex = 38;
this.radioButtonFB.TabStop = true;
this.radioButtonFB.Text = "Free Banking";
this.radioButtonFB.UseVisualStyleBackColor = true;
//
// textBoxReserveRatio
//
this.textBoxReserveRatio.Location = new System.Drawing.Point(266, 139);
this.textBoxReserveRatio.Name = "textBoxReserveRatio";
this.textBoxReserveRatio.Size = new System.Drawing.Size(36, 20);
this.textBoxReserveRatio.TabIndex = 37;
this.textBoxReserveRatio.Text = "0.5";
//
// labelReserveRatio
//
this.labelReserveRatio.AutoSize = true;
this.labelReserveRatio.Location = new System.Drawing.Point(154, 142);
this.labelReserveRatio.Name = "labelReserveRatio";
this.labelReserveRatio.Size = new System.Drawing.Size(105, 13);
this.labelReserveRatio.TabIndex = 35;
this.labelReserveRatio.Text = "Initial Reserve Ratio:";
//
// textBoxNontraders
//
this.textBoxNontraders.Location = new System.Drawing.Point(265, 111);
this.textBoxNontraders.Name = "textBoxNontraders";
this.textBoxNontraders.Size = new System.Drawing.Size(37, 20);
this.textBoxNontraders.TabIndex = 1;
this.textBoxNontraders.Text = "0";
//
// labelNontraders
//
this.labelNontraders.AutoSize = true;
this.labelNontraders.Location = new System.Drawing.Point(48, 113);
this.labelNontraders.Name = "labelNontraders";
this.labelNontraders.Size = new System.Drawing.Size(211, 13);
this.labelNontraders.TabIndex = 0;
this.labelNontraders.Text = "Number of Non-trader/depositor/borrowers:";
//
// groupBoxTrading
//
this.groupBoxTrading.Controls.Add(this.labelDividendSelection);
this.groupBoxTrading.Controls.Add(this.comboBoxFixedDividend);
this.groupBoxTrading.Controls.Add(this.textBoxC3assets);
this.groupBoxTrading.Controls.Add(this.labelC3assets);
this.groupBoxTrading.Controls.Add(this.textBoxC3money);
this.groupBoxTrading.Controls.Add(this.textBoxAssetPrice);
this.groupBoxTrading.Controls.Add(this.labelAssetPrice);
this.groupBoxTrading.Controls.Add(this.labelC3cash);
this.groupBoxTrading.Controls.Add(this.textBoxC2assets);
this.groupBoxTrading.Controls.Add(this.labelC2assets);
this.groupBoxTrading.Controls.Add(this.textBoxC2money);
this.groupBoxTrading.Controls.Add(this.labelC2cash);
this.groupBoxTrading.Controls.Add(this.labelNumInv);
this.groupBoxTrading.Controls.Add(this.textBoxNumInv);
this.groupBoxTrading.Controls.Add(this.textBoxDiv4);
this.groupBoxTrading.Controls.Add(this.labelPercentMom);
this.groupBoxTrading.Controls.Add(this.labelDiv4);
this.groupBoxTrading.Controls.Add(this.textBoxPercentMom);
this.groupBoxTrading.Controls.Add(this.textBoxDiv3);
this.groupBoxTrading.Controls.Add(this.labelPercentValue);
this.groupBoxTrading.Controls.Add(this.labelDiv3);
this.groupBoxTrading.Controls.Add(this.textBoxPercentValue);
this.groupBoxTrading.Controls.Add(this.textBoxDiv2);
this.groupBoxTrading.Controls.Add(this.labelPercentSpec);
this.groupBoxTrading.Controls.Add(this.labelDiv2);
this.groupBoxTrading.Controls.Add(this.textBoxPercentSpec);
this.groupBoxTrading.Controls.Add(this.textBoxDiv1);
this.groupBoxTrading.Controls.Add(this.labelInvC1cash);
this.groupBoxTrading.Controls.Add(this.labelDiv1);
this.groupBoxTrading.Controls.Add(this.textBoxC1money);
this.groupBoxTrading.Controls.Add(this.labelInvC1assets);
this.groupBoxTrading.Controls.Add(this.textBoxC1assets);
this.groupBoxTrading.Location = new System.Drawing.Point(27, 22);
this.groupBoxTrading.Name = "groupBoxTrading";
this.groupBoxTrading.Size = new System.Drawing.Size(289, 480);
this.groupBoxTrading.TabIndex = 36;
this.groupBoxTrading.TabStop = false;
this.groupBoxTrading.Text = "Asset Trading";
//
// labelDividendSelection
//
this.labelDividendSelection.AutoSize = true;
this.labelDividendSelection.Location = new System.Drawing.Point(73, 441);
this.labelDividendSelection.Name = "labelDividendSelection";
this.labelDividendSelection.Size = new System.Drawing.Size(99, 13);
this.labelDividendSelection.TabIndex = 44;
this.labelDividendSelection.Text = "Dividend Selection:";
//
// comboBoxFixedDividend
//
this.comboBoxFixedDividend.FormattingEnabled = true;
this.comboBoxFixedDividend.Items.AddRange(new object[] {
"Random",
"1",
"2",
"3",
"4"});
this.comboBoxFixedDividend.Location = new System.Drawing.Point(178, 438);
this.comboBoxFixedDividend.Name = "comboBoxFixedDividend";
this.comboBoxFixedDividend.Size = new System.Drawing.Size(80, 21);
this.comboBoxFixedDividend.TabIndex = 43;
this.comboBoxFixedDividend.Text = "Random";
//
// textBoxC3assets
//
this.textBoxC3assets.Location = new System.Drawing.Point(223, 273);
this.textBoxC3assets.Name = "textBoxC3assets";
this.textBoxC3assets.Size = new System.Drawing.Size(36, 20);
this.textBoxC3assets.TabIndex = 40;
this.textBoxC3assets.Text = "1";
//
// labelC3assets
//
this.labelC3assets.AutoSize = true;
this.labelC3assets.Location = new System.Drawing.Point(28, 276);
this.labelC3assets.Name = "labelC3assets";
this.labelC3assets.Size = new System.Drawing.Size(187, 13);
this.labelC3assets.TabIndex = 39;
this.labelC3assets.Text = "Class III Asset Endowment per Trader:";
//
// textBoxC3money
//
this.textBoxC3money.Location = new System.Drawing.Point(223, 245);
this.textBoxC3money.Name = "textBoxC3money";
this.textBoxC3money.Size = new System.Drawing.Size(36, 20);
this.textBoxC3money.TabIndex = 38;
this.textBoxC3money.Text = "9.45";
//
// labelC3cash
//
this.labelC3cash.AutoSize = true;
this.labelC3cash.Location = new System.Drawing.Point(23, 248);
this.labelC3cash.Name = "labelC3cash";
this.labelC3cash.Size = new System.Drawing.Size(193, 13);
this.labelC3cash.TabIndex = 37;
this.labelC3cash.Text = "Class III Money Endowment per Trader:";
//
// textBoxC2assets
//
this.textBoxC2assets.Location = new System.Drawing.Point(222, 218);
this.textBoxC2assets.Name = "textBoxC2assets";
this.textBoxC2assets.Size = new System.Drawing.Size(37, 20);
this.textBoxC2assets.TabIndex = 36;
this.textBoxC2assets.Text = "2";
//
// labelC2assets
//
this.labelC2assets.AutoSize = true;
this.labelC2assets.Location = new System.Drawing.Point(32, 221);
this.labelC2assets.Name = "labelC2assets";
this.labelC2assets.Size = new System.Drawing.Size(184, 13);
this.labelC2assets.TabIndex = 35;
this.labelC2assets.Text = "Class II Asset Endowment per Trader:";
//
// textBoxC2money
//
this.textBoxC2money.Location = new System.Drawing.Point(222, 192);
this.textBoxC2money.Name = "textBoxC2money";
this.textBoxC2money.Size = new System.Drawing.Size(35, 20);
this.textBoxC2money.TabIndex = 34;
this.textBoxC2money.Text = "5.85";
//
// labelC2cash
//
this.labelC2cash.AutoSize = true;
this.labelC2cash.Location = new System.Drawing.Point(25, 195);
this.labelC2cash.Name = "labelC2cash";
this.labelC2cash.Size = new System.Drawing.Size(190, 13);
this.labelC2cash.TabIndex = 33;
this.labelC2cash.Text = "Class II Money Endowment per Trader:";
//
// groupBoxGeneral
//
this.groupBoxGeneral.Controls.Add(this.labelNonTradingPeriods);
this.groupBoxGeneral.Controls.Add(this.textBoxIntPerPeriod);
this.groupBoxGeneral.Controls.Add(this.textBoxPeriods);
this.groupBoxGeneral.Controls.Add(this.textBoxNonTradingPeriods);
this.groupBoxGeneral.Controls.Add(this.labelPeriods);
this.groupBoxGeneral.Controls.Add(this.textBoxSessions);
this.groupBoxGeneral.Controls.Add(this.labelIntsPerPeriod);
this.groupBoxGeneral.Controls.Add(this.labelSessions);
this.groupBoxGeneral.Location = new System.Drawing.Point(331, 306);
```

```
            this.groupBoxGeneral.Name = "groupBoxGeneral";
            this.groupBoxGeneral.Size = new System.Drawing.Size(328, 145);
            this.groupBoxGeneral.TabIndex = 37;
            this.groupBoxGeneral.TabStop = false;
            this.groupBoxGeneral.Text = "General";
            //
            // labelNonTradingPeriods
            //
            this.labelNonTradingPeriods.AutoSize = true;
            this.labelNonTradingPeriods.Location = new System.Drawing.Point(99, 108);
            this.labelNonTradingPeriods.Name = "labelNonTradingPeriods";
            this.labelNonTradingPeriods.Size = new System.Drawing.Size(159, 13);
            this.labelNonTradingPeriods.TabIndex = 30;
            this.labelNonTradingPeriods.Text = "Preliminary, Non-trading Periods:";
            //
            // textBoxNonTradingPeriods
            //
            this.textBoxNonTradingPeriods.Location = new System.Drawing.Point(268, 105);
            this.textBoxNonTradingPeriods.Name = "textBoxNonTradingPeriods";
            this.textBoxNonTradingPeriods.Size = new System.Drawing.Size(35, 20);
            this.textBoxNonTradingPeriods.TabIndex = 29;
            this.textBoxNonTradingPeriods.Text = "0";
            //
            // FormMainConf
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(684, 530);
            this.Controls.Add(this.buttonRun);
            this.Controls.Add(this.groupBoxBanking);
            this.Controls.Add(this.groupBoxGeneral);
            this.Controls.Add(this.groupBoxTrading);
            this.Name = "FormMainConf";
            this.Text = "Configure Bankers and Asset Traders";
            this.Load += new System.EventHandler(this.Form1_Load);
            this.groupBoxBanking.ResumeLayout(false);
            this.groupBoxBanking.PerformLayout();
            this.groupBoxTrading.ResumeLayout(false);
            this.groupBoxTrading.PerformLayout();
            this.groupBoxGeneral.ResumeLayout(false);
            this.groupBoxGeneral.PerformLayout();
            this.ResumeLayout(false);

        }

        #endregion

        private System.Windows.Forms.Label labelNumBanks;
        private System.Windows.Forms.TextBox textBoxNumBanks;
        private System.Windows.Forms.Label labelNumInv;
        private System.Windows.Forms.TextBox textBoxNumInv;
        private System.Windows.Forms.Label labelPercentMom;
        private System.Windows.Forms.TextBox textBoxPercentMom;
        private System.Windows.Forms.Label labelPercentValue;
        private System.Windows.Forms.TextBox textBoxPercentValue;
        private System.Windows.Forms.Label labelPercentSpec;
        private System.Windows.Forms.TextBox textBoxPercentSpec;
        private System.Windows.Forms.Label labelInvC1cash;
        private System.Windows.Forms.TextBox textBoxC1money;
        private System.Windows.Forms.Label labelInvC1assets;
        private System.Windows.Forms.TextBox textBoxC1assets;
        private System.Windows.Forms.Label labelBankCashEndow;
        private System.Windows.Forms.TextBox textBoxBankCashEndow;
        private System.Windows.Forms.Label labelPeriods;
        private System.Windows.Forms.TextBox textBoxPeriods;
        private System.Windows.Forms.Label labelIntsPerPeriod;
        private System.Windows.Forms.TextBox textBoxIntPerPeriod;
        private System.Windows.Forms.Label labelDiv1;
        private System.Windows.Forms.TextBox textBoxDiv1;
        private System.Windows.Forms.Label labelDiv2;
        private System.Windows.Forms.TextBox textBoxDiv2;
        private System.Windows.Forms.Label labelDiv3;
        private System.Windows.Forms.TextBox textBoxDiv3;
        private System.Windows.Forms.Label labelDiv4;
        private System.Windows.Forms.TextBox textBoxDiv4;
        private System.Windows.Forms.Label labelSessions;
        private System.Windows.Forms.TextBox textBoxSessions;
        private System.Windows.Forms.Button buttonRun;
        private System.Windows.Forms.Label labelAssetPrice;
        private System.Windows.Forms.TextBox textBoxAssetPrice;
        private System.Windows.Forms.TextBox textBoxSavingsRateElasticity;
        private System.Windows.Forms.Label labelSavingsRateElasticity;
        private System.Windows.Forms.GroupBox groupBoxBanking;
        private System.Windows.Forms.GroupBox groupBoxTrading;
        private System.Windows.Forms.GroupBox groupBoxGeneral;
        private System.Windows.Forms.TextBox textBoxNontraders;
        private System.Windows.Forms.Label labelNontraders;
        private System.Windows.Forms.TextBox textBoxNonTradingPeriods;
        private System.Windows.Forms.Label labelNonTradingPeriods;
        private System.Windows.Forms.Label labelReserveRatio;
        private System.Windows.Forms.TextBox textBoxReserveRatio;
        private System.Windows.Forms.Label labelC2cash;
        private System.Windows.Forms.TextBox textBoxC2assets;
        private System.Windows.Forms.Label labelC2assets;
        private System.Windows.Forms.TextBox textBoxC2money;
        private System.Windows.Forms.TextBox textBoxC3assets;
        private System.Windows.Forms.Label labelC3assets;
        private System.Windows.Forms.TextBox textBoxC3money;
        private System.Windows.Forms.Label labelC3cash;
        private System.Windows.Forms.RadioButton radioButtonFB;
        private System.Windows.Forms.TextBox textBoxLoanSize;
        private System.Windows.Forms.Label labelLoanSize;
        private System.Windows.Forms.Label labelQ;
        private System.Windows.Forms.TextBox textBoxQ;
        private System.Windows.Forms.ComboBox comboBoxFixedDividend;
        private System.Windows.Forms.Label labelDividendSelection;
    }
}
```

APPENDIX C5: SOURCE CODE OF THE SIMULATION SOFTWARE (PROGRAM.CS)

```
// *****************************************************************
// This software was written and is owned and copyrighted by
// William McBride of George Mason University, for his dissertation.
// July 23, 2010.
// *****************************************************************

// This file was generated by Microsoft Visual Studio.

using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Bubbles_and_Banks
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new FormMainConf());
        }
    }
}
```

REFERENCES

REFERENCES

Alfarano, Simone, Thomas Lux, and Friedrich Wagner. 2005. Estimation of agent-based models: The case of an asymmetric herding model. *Computation Economics* 26(1): 19-49.

Andreassen, Paul and Stephen Kraus. 1988. Judgmental prediction by extrapolation. Mimeo, Harvard University.

Baltensperger, E.. 1980. Alternative approaches to the theory of the banking firm. *Journal of Monetary Economics* 6: 1-37.

Black, Fischer. 1986. Noise. *Journal of Finance* 41: 529–543.

Caginalp, Gunduz, David Porter, and Vernon L. Smith. 1998. Initial cash/stock ratio and stock prices: An experimental study. *Proceedings of the National Academy of Sciences*, 95(2): 756-61.

Chan, N., B. LeBaron, A. W. Lo, and T. Poggio. 1999. Agent-based models of financial markets: A comparison with experimental markets. Technical report. Cambridge, MA: MIT.

Gode, D.K., S. Sunder. 1993. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of Political Economy* 101: 119-137.

De Long, J. B., A. Shleifer, L. Summers, and R. J. Waldmann. 1990a. Positive feedback investment strategies and destabilizing rational speculation. *Journal of Finance*, June, 379-95.

1990b. Noise trader risk in financial markets. *Journal of Political Economy* 98: 703 – 738.

1991. The survival of noise traders in financial markets. *The Journal of Business*, January: 1-19.

Dowd, Kevin. 1992. (ed.) *The experience of free banking*. London: Routledge.

Duffy, John. 2006. Agent-based models and human subject experiments. In *Handbook of Computational Economics: Volume 2, Agent-Based Computational Economics,* *Handbooks in Economics Series*, eds. Leigh Tesfatsion and Kenneth L. Judd, 949-1011. Amsterdam: North-Holland.

Duffy, John and M. U. Ünver. 2006. Asset price bubbles and crashes with near zero-intelligence traders. *Economic Theory* 27: 537-563.

Edgeworth, F. Y. 1888. The mathematical theory of banking. *Journal of the Royal Statistical Association.* 51(1): 113-127.

Hussam, Reshmaan N., David Porter, and Vernon Smith. 2008. Thar she blows: Can bubbles be rekindled with experienced subjects? *The American Economic Review* 98(3): 924-937.

LeBaron, Blake. 2000. Agent-based computational finance: Suggested readings and early research. *Journal of Economic Dynamics & Control* 24: 679-702.

2001. A builder's guide to agent-based financial markets**.** *Quantitative Finance* 1: 254-261.

Noussair, Charles, Stephane Robin, and Bernard Ruffieux. 2001. Price bubbles in laboratory asset markets with constant fundamental values. *Experimental Economics* 4(1): 87 – 105.

Olivera, J. H. G. 1971. The Square-Root Law of Precautionary Reserves. *Journal of Political Economy* 79(5), 1095-1104.

Porter, David P., and Vernon L. Smith. 1995. Futures contracting and dividend uncertainty in experimental asset markets. *Journal of Business*, 68(4): 509-41.

Rust, J., J. Miller, and R. Palmer. 1992. Behavior of trading automota in a computerized double auction market. In *The double auction market: Institutions, theories, and evidence*, eds. D. Friedman and J. Rust, 155-198. Reading, MA: Addison-Wesley.

Selgin, George. 1988. *The theory of free banking: Money supply under competitive note issue*. Totowa, New Jersey: Rowman & Littlefield.

1994. Free banking and monetary control. *The Economic Journal* 104(427): 1449-1459.

1996. *Bank deregulation and monetary order*. London: Routledge.

Smith, Vernon L.. 1976. Experimental economics: Induced value theory. *American Economic Review* 66(2): 274–279.

Smith, Vernon L., Gerry Suchanek and Arlington Williams. 1988. Bubbles, crashes, and endogenous expectations in experimental spot asset markets. *Econometrica* 56(5): 1119-1151.

Smith, Vernon L., and Arlington Williams. 1983. An experimental comparison of alternative rules for competitive market exchange. In *Auctions, Bidding, and Contracting: Uses and Theory,* eds. Richard Engelbrecht-Wiggans, Marin Shubik, and Robert Stark, 307-334. New York: New York University Press.

Smith, Vernon L., Mark van Boening, and Charissa P. Wellford. 2000. Dividend timing and behavior in laboratory asset markets. *Economic Theory* 16: 567 – 583.

Tesfatsion, Leigh and Kenneth L. Judd. 2006. (eds.) *Handbook of Computational Economics: Volume 2, Agent-Based Computational Economics, Handbooks in Economics Series*. Amsterdam: North-Holland.

White, Lawrence H. 1995. *Free banking in Britain: Theory, experience, and debate, 1800-1845,* 2nd ed., revised and enlarged. London: Institute of Economic Affairs.

CURRICULUM VITAE

As part of the PhD program at George Mason University, I received a MA in economics in 2006. Prior to beginning the PhD program, I worked as a software engineer for about 7 years, primarily in the defense industry. Before that, my undergraduate education resulted in a BS in Physics from the The University of the South and a BS in Electrical Engineering from Washington University in St. Louis.