Contents

1. Introduction

1.1. Goal of the Project and its Documentation

This documentation aims to outline the functionality and usage of two different software versions of the Santa Fe Institute (SFI) Artificial Stock Market (ASM) developed by Brian Arthur, John Holland, Blake LeBaron, and Richard Palmer. The following sections on the history of the project do not claim to provide a full exegesis on the motivations and intentions of the project. For a deeper motivation see the papers listed in the bibliography in **Section 6**. Suffice it to say that there is much speculation that financial markets are rife with behavior by traders that can better be described by rules of thumb than strict rules of maximization popular in the non-empirical financial literature. To complicate matters more for those who wish to provide clean, closed form solutions to tractable analytical models, there are signs of heterogeneity in the rules that financial agents apply. As in other fields, a natural approach to such situations that does not rely on homogenous, hyper-rational utility maximizing agents is agent-based simulation. Such a method does not rely on closed-form solutions that can be analyzed with comparative statics, but rather provides stochastic paths where agents are allowed to be heterogeneous and further to adapt their behavior over time. The SFI ASM is the first such application of this tool to financial markets.

1.2. History of the Project

The first version of the ASM appeared in the early 90s. This version was written in Objective-C and had a fully functional user interface that ran only on the NextStep platform. The application could run on other platforms that supported Objective-C without the NextStep Graphical User Interface (GUI). An early publication describing the model can be found in Palmer, Arthur, Holland, LeBaron, and Tayler (1994). A more extensive and recent description can be found in "Asset Pricing under Endogenous Expectations in an Artificial Stock Market" by Arthur, Holland, LeBaron, Palmer, and Tayler (1997) in *The Economy as an Evolving Complex System II* edited by Arthur, Durlauf, and Lane.

It was recognized early on that this version lacked two important elements: a user interface that could be run across a number of platforms and extensive data collection facilities that can facilitate significant empirical investigation of a complicated stream of data. Both of these concerns have been addressed, though not presently in one product.

During the summer of 1996, Brandon Weber ported the NextStep version to SFI's Swarm platform. This version aimed to capitalize on the stated goals of the Swarm agent-based simulation environment. Swarm provides a portable product that can currently run on systems from the DEC Alpha to the Macintosh. It provides a GUI that allows dynamic parameter adjustment and in certain environments the ability to probe specific agents during the simulation to see how that particular agent evolves over time. The Swarm version allows one to both visualize a run and to replicate it. The ability of the user to interact with the simulation and to see its results dynamically (and in turn to decide whether a run is worth replicating) makes the Swarm version the ideal one for experimentation with many runs with many different parameter configurations.

For better or worse, the Swarm version also pared down some of the original NextStep version's functionality. For example, the original version provides a number of different "specialist" classes that control the clearing of the asset market that the ASM models. Naturally none of the crucial features were eliminated, but users may see some differences along this dimension. These differences will be discussed in detail below.

Since the time that the ASM was ported to Swarm, the NextStep version has undergone substantial change. Richard Palmer cleaned and reworked significant parts of the product in 1998. The changes are as follows. First, all vestiges of the NextStep GUI have been eliminated. Now the Objective-C version (hereafter the straight Objective-C version) can be run on any platform on which the Swarm version can. The costs are that it has become a full batch version (i.e., without a GUI), taking simulation parameter values from a file and writing data only to files. Second, its data output features were significantly improved making it (considerably) more functional in this dimension than the Swarm version. The Swarm version can also be run in a batch mode with some modest data collection features. One possibility is that a user might utilize the Swarm version to visualize output and to identify runs that are worth replicating, but then user the straight Objective-C version to produce analyzable data.

In more recent history, Lebaron, Arthur, and Palmer (1998) have published work in the *Journal of Economic Dynamics and Control* on the time series properties of the data produced by the ASM. Palmer, Arthur, Holland, and LeBaron (1998) have work in *Artificial Life and Robotics* on general properties of the simulation. Both of these are in addition to the paper in *The Economy as an Evolving Complex System II*. Additionally, Shareen Joshi and Mark Bedau (1999) have done work with the ASM and shown that the trading rules of agents give rise to behavior similar to that seen in an infinitely repeated prisoner's dilemma.

1.3. Goal and Extent of Documentation and Support

The authors of the project feel that there are still some valuable insights and extensions that can be accomplished using this software. The recent publications are a testament to this. But it is also clear that only with a dedicated user community can the product improve and hopefully the two separate packages be integrated to produce one solid piece of software. As such SFI is providing this concerted effort at documentation and some limited support.

The documentation will provide a full discography of the two available versions, their stated goals, the differences between them, their various strengths and weaknesses, and a complete description of their use and potential to be extended. The level of support is yet to be determined. The ideal vision is that as the user community develops support will be able to develop within that community. SFI pledges to centralize and manage user contributions, FAQs, and further documentation as it arises from users. It, however, has not made plans for a long-term commitment to specific user support.

1.4. Documentation Conventions

Throughout this documentation several conventions will be followed. First, parameter names, variable names, and mathematical symbols appear in italics (e.g., the price is *p* and the interest rate is *intrate*). When sections of the documentation are referenced, they are written in boldface (e.g., **Section x**). Filenames appear in quotations except for in section headings (e.g., “ASMModelSwarm.h”). Please note that there are times when we are talking about the notion of a file in the code, for example the market control parameter file in the straight Objective-C version, and instead we use the name of the “default” file included with the code. For example, in the case of the market parameter file, we may call it “mcontrol”, even if the user could in theory have a market control parameter file with an entirely different name. This should be clear when it occurs.

2. Code Functionality

2.1. General Functionality of Both Versions

The way the market works is the same in both versions. In the straight Objective-C version there are some additional features that were removed from the Swarm version of the code. These features by no means change significant aspects of the market. One idea of the project is that if such additions are found interesting, users can add them with modularity in mind, so that other users can easily plug these features into their own application. What follows is a description of the general structure of the market and its implementation in the code. Here similarities in file structure will be highlighted as opposed to specific differences in their content. For example, there is a specialist in the market whose purpose it is to clear the market, i.e., equate, as much as possible, supply and demand for the asset. The straight Objective-C version has many types of specialist, which are all implemented in the “specialist.h” and “specialist.m” files. The Swarm version only has a subset of these specialists, but both versions have specialist files of the same name.

2.1.1. General Model and Agent Specification

Again, for a detailed description of the economic model being considered, see Arthur, Holland, LeBaron, Palmer, and Tayler (1997). For our purposes here, suffice it to say that the model is a standard neo-classical model of asset pricing. Agents are risk averse with utility exhibiting constant absolute risk aversion. Agents, in turn, have standard demand functions for the asset that change based on their prediction of the next period price of the asset p_{t+1} and its dividend d_{t+1} (actually, they simply predict $p_{t+1}+d_{t+1}$). Once agents make their predictions, they submit their resultant demand for the asset (which can be negative, indicating a desire to sell the asset) to the specialist. The specialist either sets a price or adjusts the price (depending on the specialist type) until the price is found that most closely clears the market. This is done taking into consideration each agent's standard budget constraint. If short selling is not permitted, an agent with zero wealth cannot buy the asset. If short selling is allowed, there is a minimum negative wealth that agents can hold before they are prohibited from short selling. Resources not spent on the asset are spent on a risk free bond (called cash in the simulation) that gives a fixed rate of return, *r*. All assets held lend to the agents' wealth in the next period.

Standard economic theory assumes that agents have the same demand function for the asset and rational expectations as to the next period price and dividend. The dividend, as in this model, follows a simple process (of many types in the straight Objective-C version or AR(1) in the Swarm version) and thus can be predicted quite accurately by the simplest agent. Since agents are all the same there exists a unique price that clears the market. A result of this analysis is that the path of prices can be perfectly predicted at the beginning of time. The standard model is attractive because of the ease with which the researcher can obtain closed form, analytical solutions. Once we assume that agents are heterogeneous (a much more attractive assumption from the perspective of realism) it becomes very difficult to obtain the clean solutions found in the case with homogenous agents. This gives rise to the need for agent-based simulation. The flexibility of the model lies in the specification of the agent(s). Each agent in the agent based model employs a number of different ways to forecast the price plus dividend. In the standard Objective-C version there are even several different classes of agents. These will be described in detail below. Additionally, certain agents evolve over time according to different evolutionary algorithms (particularly genetic algorithms). There is simply no room for this in the standard economic model.

The novel approach, then, of this project is that agents are allowed to have heterogeneous demand functions as a result of their heterogeneous methods of predicting $p_{t+1}+d_{t+1}$. As was mentioned above, an agent's demand changes with her prediction of the next period's state. In this brief description several crucial components of the simulation have been identified. There must be agents, a dividend process, and a specialist; the states of the world will be controlled by a world component, and beyond this the only remaining concerns are control files (which may be extensive). The basic structure of the simulation is simple. The timing is as follows:

Time *0* → the simulation is provided with a warm-up history.

Time *t* →

Step 1 → the time *t-1* dividend is determined (after time *t-1* trading has already been completed in the previous period).

Step 2 → the World updates the state. Note that this is called in each period after a new dividend has been declared but before the bidding and price adjustment. The information seen by the agents thus does not reflect the current price. The "price" here becomes the "old price" by the end of the period. The dividend used during the updating of the state is the latest value, though it could be argued that it should be the one before, to match the price. For some statistics the old dividend is used.

Step 3 → agents submit their demand functions, based on their time *t-1* predictions of this period's dividend plus price, to the specialist who adjusts the price to clear the market. This is done taking into consideration each agent's budget constraint.

Step 4 → after realization of the price and dividend, agents update their prediction rules (i.e., determine accuracy and variance). From here on prediction rules will be called **forecasters**. The reason is that a *forecaster* may be a combination of different *trading rules*. Agents also update their wealth (determined by the return on the asset and the return on their bond/cash).

Step 5 → agents use the current (time *t*) state and their potentially heterogeneous forecasters to predict the period *t+1* price and dividend. Before this, depending on the type of agent, they may have to choose one forecaster of a number based on the state and the success of their various forecasters (determined by their accuracy and variance).

Time *t+1* → the process repeats.

Regardless of the agent, specialist, dividend process, control process, etc. being used, the basic structure of the simulation is always as above.

2.1.2. Specific Agent Specification

The main element moving the simulation is the specification of the agent. There are interesting projects that examine the results arising from the provision of different specialists or dividend processes, but the design of the market is mainly suited to analyzing the behavior of agents given the other aspects of the environment. Indeed, the model follows a standard rational expectations model of asset pricing for a reason. The point is to develop results that arise as the result of the heterogeneity of agents. This serves two purposes. First, it can immediately be recognized that the conclusion of the standard model does not hold when the homogeneity assumption is relaxed. Second, this model can better match the price discovery process of real asset markets. As such, the details of the agents included in the simulation need to be carefully considered.

2.1.2.1. The BFagent

The workhorse agent of the model is called the BFagent, or "Bit-string-Forecasting" agent. It is based on classifier systems developed by John Holland. This agent relies on a suite of *forecasters* that use combinations of technical and non-technical (fundamental) trading *rules*. Specifically, a *forecaster* is a string of bits that are either on, off, or hashed (1, 0, and #, respectively), where each bit represents one of the fixed *rules* considered in the model. Please note that in the in-code documentation the word "rule" is often used in reference to a “forecaster”. Once you have a clear understanding of what this documentation means by a “forecaster”, this should cause little confusion. Thus each forecaster is a string of bits of the same, fixed length, but it consists of a host of different rules that can be used to interpret the state of the market. For example, a rule may be “the current price is greater than the 50-day moving average of price”. This is a fact that can be observed. It is also consider a “technical trading rule” because it is a function of the history of prices. Another rule is “price is over-valued by 10 percent”. This rule is a fundamental rule, i.e., one that is not a history of prices. A potential forecaster is the one where the two aforementioned rules are turned on and all other rules are turned off. Forecasters are matched with the global string of bits held by the World. As was mentioned, rules are unambiguous (they can be observed to be either true or false [on or off] in a period). Thus in each period the World is a string of bits that are either on or off. The World is of the same length as each forecaster. A forecaster can only be used in a particular period if it exactly matches the state of the world. A particular bit on a forecaster string that is hashed matches the World regardless of the state of that rule in the World. For example, suppose that there are five possible rules. The following indicates a state of the World and forecasters that are either match or do not match the state of the World.

World	**1**	**0**	**1**	**0**	**0**	
Forecaster 1	**1**	**1**	**#**	**#**	**#**	**No match**
Forecaster 2	**#**	**0**	**1**	**0**	**0**	**No match**
Forecaster 3	**#**	**#**	**#**	**#**	**0**	**Match**
Forecaster 4	**1**	**0**	**1**	**0**	**#**	**Match**

Rules that match in a period can be used in that period to predict the next period’s price plus dividend. Associated with each forecaster are three numbers, *a*, *b*, and *c* that determine the prediction of $p_{t+1}+d_{t+1}$ as a function of p_t+d_t. The prediction is simply given by

$$p_{t+1} + d_{t+1} = a * (p_t + d_t) + b * d_t + c.$$

If multiple rules match the state of the World, the one that is used can be selected according to one of three criteria (selected by parameter at the start of the simulation).

1. Average: the active forecaster’s forecasts are averaged, weighted by their strength;
2. Best: the forecast from the forecaster with the lowest current estimated variance is used;
3. Roulette: one of the forecasters is selected at random with probability proportional to its strength, and its forecast is used.

After a forecaster is in hand, a prediction is made, demand is determined and submitted to the specialist, and the market is cleared.

After an agent uses a forecaster, the forecaster’s accuracy and variance are updated and in turn so is its “strength”. A simple squared-error $(actual - forecast)^2$ is used. The “current estimated variance”, *v*, of each rule is formed as an exponentially weighted moving average of these squared errors with

$$v_{t+1} = A * v_t + (1-A) * (\text{squared error}),$$

where $A = exp(-1/tauv)$, where *tauv* is a healing-time parameter. This variance *v* of each rule is also transformed into the rule's “strength”, using the relation:

$$strength = C - v – bitcost * specificity.$$

Here *bitcost* is a parameter and *specificity* is the number of “conditions” in the rule (the number of on/off condition bits, i.e., 0/1 or non-hashed bits). The parameter *C* is a constant chosen to make the strength positive:

$$C = maxdev + bitcost * (\text{maximum possible specificity}).$$

A BFagent starts with a number of different forecasters, and they are used according to whether they match the state of the World and according to their strength. The parameters *a*, *b*, and *c* and the forecasters themselves are changed over time according to a genetic algorithm (GA). The frequency with which the GA is activated is determined by a parameter; some interesting work has been done investigating precisely this point. Forecasters are ranked according to strength, with stronger forecasters being chosen for crossover with a higher probability. There is also a probability of mutation for each bit on the forecasters chosen as potential crossover candidates. The GA naturally selects superior rules and *a fortiorari* creates even more superior rules. The evolution of forecasters is intended to replicate a process that a real trader might follow. Deeper descriptions of these points and more can be found in the referenced papers, the in-code comments, and the documentation on the BFagent files.

2.1.2.2. Other Agents

All agents are subclasses of an agent superclass. One of the intentions of this project is that the user community will develop new agents. The straight Objective-C version contains a number of agents other than the BFagent. Brandon Weber has developed an agent in the Swarm environment that learns using an Artificial Neural Network. The latter agent is not included with this release of the code, but the agent will be added to the ASM web-site as a user contribution with documentation on how to add an agent.

The other agents in the straight Objective-C version are: the BSagent (Bit-String agent), FFagent (fixed-forecaster agent), and DUagent (dumb agent). The BSagent is a simplified version of the BFagent. It has multiple bit-string forecasters that can change over time via a genetic algorithm, but it can only make binary buy/sell decisions using these rules. The FFagent has a set of forecasting rules randomly selected from a global set of rules. These rules do not evolve but are still selected on the basis of strength. The DUagent (or dumb agent) always attempts to buy or sell in every period some number of shares randomly selected at the start of the simulation. It does so depending on the value of one attribute of the World that does not depend on the price. As such, the DUagent makes no prediction over time. It is included as a demonstration of some of the simulation's capabilities and as a benchmark for comparisons between agents. Any reasonably intelligent agent should be able to outdo the DUagent. In the sections below, the capabilities of each of these agents will be described in more depth.

In the Swarm version, the ANNagent forecasts the next period's price plus dividend using an Artificial Neural Network (ANN). Agents are heterogeneous because each is endowed with a randomly selected ANN architecture. This agent is simple. There is no evolution, but the nature of an ANN makes them fairly sophisticated. There are potentially interesting comparisons to be made between this agent and the BFagent described in depth above (**Section 2.1.2.1.**).

2.2. Files and File System Structure

We begin this section by outlining the files that implement the main actors in the simulation, viz., the Agents, the Specialist, the Dividend, and the World. We point out the differences in these files between the two versions. We then outline the main control files, which hold most of the differences between the two versions. Finally, we discuss the parameter files. These are relatively simple in the Swarm version, but are indicative of some complex issues in the straight Objective-C version.

As was mentioned above, this set of documentation does not aspire to provide an introduction to Swarm. It is assumed (in part) that the user of the Swarm version knows Swarm. One objective of Swarm is to act as a common language for the writing of agent-based simulations. This serves a number of purposes – not the least of which is to provide a common language across researchers and across a researcher's own projects. To that extent any user who feels comfortable with the straight Objective-C version is strongly encouraged to use the Swarm version and to implement the more sophisticated input/output aspects of the Objective-C version into the Swarm version.

Regardless, there are some differences in the structure of the code. Fortunately Swarm is based on Objective-C, and as such much of the syntax is the same. The differences in syntax mostly lie in the use of Swarm objects and the way Swarm maintains lists. This, however, leads to significant overhead in the straight Objective-C version with respect to control files. The ASM itself is like a mini-Swarm. The non-syntax differences lie in the fact that the straight Objective-C version contains a number of additional features, e.g., multiple specialist types and multiple agent types. These additions often add complications that overwhelm the benefit of the feature. Other times this is not the case. In this situation the feature can typically be easily added to the Swarm version. Finally, the Objective-C version runs entirely without a GUI. All parameters are loaded from files, and they are loaded differently than they are in even Swarm's batch mode. The Swarm system is easier to maintain. However, the Objective-C version does provide a number of *important* output features that should be included in the Swarm version. This is left for an interested user.

Many of the differences derive from the way in which the two versions read parameters (and the extent to which the user can change parameters) and write output.

Almost every module in the simulation is parameterized so that the user can change parameters between runs. The straight Objective-C version sets parameters in a number of files. When each item of the simulation is initialized (e.g., the World, the Specialist, the Agents, etc.) they call a function that reads the relevant parameters from a file specific to that item. In the Swarm version on the other hand, if the simulation is running with the GUI the user can set a number of parameters on the interface. If the simulation is run in batch mode, the same parameters that the user can set on the GUI can be changed in one file (called "param.data"). These parameters are set in the main control module "ASMModelSwarm.h/.m", where each class needs to ask the instantiation of the ASMModelSwarm for each of its parameters.

There are a number of parameters set in files in the straight Objective-C version that the user cannot change in the Swarm version without changing the code. This, at the time, was done for convenience. If the user wants to experiment with the hard-coded parameters, it will require a minimal amount of work to mimic the way in which the parameters that can be changed are coded in the ASMModelSwarm. For example, all the parameters for the BFagent are hard-coded in the Swarm version. To add these parameters so that they appear on the GUI (or are at least centrally housed in the main control file) requires that a message be added to the BFagent class that acquires these parameters from ASMModelSwarm. These, as was alluded to, can either be controlled on the GUI or not. The advantage of this approach is that all parameters are controlled by one module (and hence are easy to find) and are likewise centralized when the batch mode is invoked. The World module for example takes all of its parameters from the ASMModelSwarm (even those that are not found on the GUI). All this will be described in much more detail later.

2.2.1. Common Files

This section concerns files that are common to both versions. We provide the names of the files for both versions (as there are a few cases where they diverge and in all cases the straight Objective-C version's filenames start with lowercase letters whereas the Swarm version's filenames usually, though not always, start with capital letters. We provide a description of the general function of each file following the list.

	Objective-C	Swarm
1)	agent.h, agent.m	Agent.h, Agent.m
2)	bfagent.h, bfagent.m	BFagent.h, BFagent.m
3)	dividend.h, dividend.m	Dividend.h, Dividend.m
4)	main.m	main.m
5)	makefile	Makefile
6)	output.h, output.m	Output.h, Output.m
7)	random.h, random.m	random.h, random.m
8)	speclist.h, speclist.m	Specialist.h, Specialist.m
9)	world.h, world.m	World.h, World.m

1) (a)Agent.h, (a)Agent.m

The agent header (".h") and implementation files (".m") institute the class "Agent", which is the main agent superclass of which all agent types are a subclass. The basic features are the same between the two versions. All agents have a demand for the asset (*demand*), an exponentially weighted profit (*profit*), current holdings of the asset (*position*), a current cash holding (*cash*), an initial cash holding (*initialcash*), a minimum position (a short selling constraint, *minholding*) and a minimum amount of cash (a borrowing constraint, *mincash*), the interest rate (global but owned by each for efficiency, *intrate*) and gross interest rate (interest rate plus one, *intratep1*), the current price and dividend (also global, *price* and *dividend*, respectively), and an id. Of these only the id differs across versions due to the different ways the simulations maintain the list of agents. In the Swarm version the id is an integer (*myID*) whereas in the straight Objective-C version it is both a number (*tag*) and a pointer to a name in the "nametree" (*nameidx*).

The agent superclass in the straight Objective-C version contains some additional structure in order to maintain the classifier system that is common between the BFagent and BSagent. Since the Swarm version only includes the former, these aspects are relegated to the implementation of the BFagent. Additionally there are several output features included in the straight Objective-C files.

There are also a number of messages (the name for methods or member functions in Objective-C) defined for the agent superclass. First, the agent superclass has messages for initialization. In the straight Objective-C version all initialization is done with the class message +initClass: theClass. In the Swarm version, this is replaced by a number of messages such as –setWorld: (World *)theWorld and –setID: id. The reason for this is that the Swarm version better respects the mores of object oriented programming. There is *one* instance of the World class controlled by the ASMModelSwarm. If another object wants access to this instantiation, then it must obtain a copy from the ASMModelSwarm. The straight Objective-C code often makes these objects global and hence potentially corruptible. Likewise, the Swarm version's Agent class has messages to obtain the price and dividend from the World, which is the owner of these variables. In the straight Objective-C version these are global. The straight Objective-C version contains a number of messages controlling bit strings, parameter settings, and output for the reasons stated above. See the code for a complete list of messages.

2) (bf)BFagent.h, (bf)BFagent.m

The BFagent files implement the BFagent class, a subclass of the Agent superclass. Each BFagent holds a *struct* of forecasters (called rules in the straight Objective-C version) that are all strings of bits that correspond to trading rules as described above (**Section 2.1.2.1.**). Forecasts hold a myriad of information about its current forecast (*forecast*), previous forecast (*lforecast*), *strength*, *variance*, *specificity*, *a*, *b*, and *c* (for demand), among other variables.

The BFagent is characterized by a number of parameters (held in a *struct*). These parameters are the same for all BFagents in the simulation, and they dictate the rate of the GA, restrictions on demand, probabilities for bits in forecast rules, ranges for the demand parameters, etc. The parameters for the BFagent in the straight Objective-C version appear in the file “bfparams” and are called at the beginning of the simulation during initialization by the BFagent class constructor. In the Swarm version the BFagent parameters are currently hard-coded. They can be easily added to the ASMModelSwarm and controlled from the GUI or in batch mode from a global parameter file. For a list of all BFagent parameters see **Section 2.2.2.3** (under “bfparams”).

In the straight Objective-C version there are a number of messages for the situation where the simulation runs with both BFagents and BSagents. These do not arise in the Swarm version. Many messages defined in the agent superclass are overridden. These generally simply return self in the agent superclass. For example, –prepareForTrading does nothing in the Agent superclass (but is defined nevertheless). The BFagent fills this function in. The reason these messages are defined at all in the agent superclass is that all agents have them in common. Including a blank message simply helps the user to understand what the minimum functionality is for any agent they are to add. Finally there are a number of new messages defined in the BFagent that control the genetic algorithm. See the extensive (but technical) in-code documentation as well as the documentation here on “bfparams” for a fuller understanding of the complex BFagent.

3) (d)Dividend.h, (d)Dividend.m

The dividend files manage the dividend process. These files are relatively simple. The straight Objective-C version allows for seven different dividend processes. They are an AR(1) process with Gaussian noise (*DIV_AR1*), a Markov process with two dividend states (high low) with fixed transition matrix (determined by parameter values at the start of the simulation) (*DIV_MARKOV*), a White Noise process where the dividend in each period is an independent draw from a Gaussian distribution (*DIV_WHITE*), a process where the state is independently high or low in each period determined by a Bernoulli trial (*DIV_IID*), a process following a square wave with potentially different up/down durations (*DIV_SQUARE*), a process following a periodic triangle wave with alternate up/down ramps (*DIV_TRIANGLE*), and a sine wave with up/down halves having potentially different durations (*DIV_SINE*). There are a number of different parameters that determine these different processes. The Swarm version on the other hand allows for only an AR(1) dividend process, and as such has less code and lighter parameterization. The dividend’s parameters are set in the straight Objective-C version in the “dparams” file. See the documentation on this for extensive discussion of these parameters. In the Swarm version, the user can control all parameters on the GUI or within the file “param.data”, which is loaded by the batch mode.

The straight Objective-C version’s dividend has the additional feature that it can be shocked by the message –shock. The shock is initially of size *shocksize* and fades away exponentially with time-constant *shocktime* (so that the shock decays by a factor of *exp(-1/shocktime)* in each period). The message –resetShock sets the shock in the current period back to 0. Shocks must be scheduled in the file “timelist” for addition to the schedule of events. This is described extensively in the section on “schedule.h,.m” (**Section 2.2.2.2.**) and “timelist” (**Section 2.2.2.3.**).

4) main.m

Main in a Swarm simulation always has the same basic structure. It creates an object called *theTopLevelSwarm*. This object is an instance of the class ASMObserverSwarm (defined and implemented in “ASMObserverSwarm.h,.m”) if the simulation is in GUI mode and of type ASMBatchSwarm (defined and implemented in “ASMBatchSwarm.h,.m”) if the simulation is in batch mode. Whichever type of object is created, *theTopLevelSwarm* is told to –buildObjects, –buildActions, –activateIn, and –go. Each will be described in turn.

The message –buildObjects always has *theTopLevelSwarm* create an instance of the class ASMModelSwarm (defined and implemented in “ASMModelSwarm.h,.m”). The ASMModelSwarm also has a message –buildObjects, and is told to do so. This message initializes all the objects it controls (the World, the Specialist, the Dividend process, the list of agents, etc.). The ASMObserverSwarm additionally creates the graphical objects.

Next *theTopLevelSwarm* is told to –buildActions. Both the ASMBatchSwarm and the ASMObserverSwarm tell the ASMModelSwarm to –buildActions. This message creates the schedule of events in the simulation. Addtionally, *theTopLevelSwarm* of either type creates a schedule to write output, and in GUI mode, the ASMObserverSwarm creates a schedule to display data in potentially several formats. The message –activateIn sets the simulation schedule and –go starts the schedule. Swarm has a built-in error handling system for all of its objects.

The straight Objective-C version is similar. It calls some global functions defined and implemented in “control.h,.m” that create instantiations of all relevant objects, write parameters, and control the writing of output. These functions are (in order) setEnvironment(argc, argv, NULL), startup( ), performEvents( ) (performs any scheduled startup events). After the “initialization” functions are called, the main loop over the number of simulation periods (*t*) commences. In each period either warmup( ) or period( ) is called along with performEvents( ). The function warmup( ) is used to create a pre-history for the simulation. After the warm-up time, the function period( ) is used. Finally, when the loop finishes, main calls some functiosn to close the simulation. All of the above functions will be described in depth in **Section 2.2.2.2.** under “control.h/.m”. These functions use globally defined parameters defined in “global.h”. Main (in the straight bjective-C version) also calls the error handling functions defined and implemented in “error.h,.m”. These files will also be discussed in **Section 2.2.2.2.** under their respective names.

5) (m)Makefile

This documentation cannot provide a complete description of the syntax of a Makefile, and the discussion that it will provide can be found under the section on Supported Platforms, Configuration, and Compilation (**Section 3**).

6) **(o)Output.h, (o)Output.m**

The straight Objective-C version includes much more sophisticated output facilities, albeit not easily digestible. One of the first priorities of any user should be to implement some of these facilities in the Swarm version, and a priority of Swarm should be to implement some general output mechanisms. Unfortunately, since Swarm is not well equipped to deal with serious output and the straight Objective-C version damn near relies on obscurity to produce output, it will be difficult for the average user to port the straight Objective-C version's output features to the Swarm version. The Swarm version writes several variables (price, dividend, and volume), but these are hardly sufficient. It does write the parameters that the user can control (along with the random seed), so that runs can be easily replicated. Implementation of a better output structure is not hard (given good C skills), but it is tedious. It is tedious because it requires a host of new functions to work through the data encapsulation. For example, the output module currently obtains an instance of the World from the ASMModelSwarm, and then it outputs the price and dividend by telling that instantiation to –getPrice and –getDividend. These message calls are easy and quick to write, but there may be many that a user might need. The straight Objective-C version gets around this "problem" by globally defining many variables that should be protected. The straight Objective-C version has the laudable feature that much data about agents (even specific ones) can be collected. Swarm should be close to these capabilities. Regardless, the deep features of the straight Objective-C version's output are described in **Section 2.2.2.3.** under the parameter file "output". The code in "output.h,.m" is too tedious to discuss here, so we focus on usage there. One point that must be made now is that the output streams specified in the "output" parameter file must be scheduled in the "timelist" file. For instance, in "output" there could be an output stream called *priceStream* that indicates that the price should be written every period for 5000 periods. The *priceStream* will never be written, however, without there being a corresponding "event" scheduled in the "timelist" file that instructs the simulation to write the *priceStream*. See **Section 2.2.2.2.** under "schedule.h,.m" and **Section 2.2.2.3.** under "timelist".

7) **random.h, random.m**

Unfortunately at the time the Swarm version was written, there was no double random number generator in Swarm. As such, the random number generators developed in the original version were maintained. Another reason for this is that most of the stochastic aspects of the simulation are in the BFagent. Since the BFagent is quite messy, it was easier to leave the files as they were. Since that time, however, Swarm has implemented a double random number generator. A cleaner implementation of the code will use this and other Swarm generators. So between the two versions, the generators in the straight Objective-C version are better and more recent. Therefore it is of some importance that the Swarm version is updated (preferably with Swarm generators). Most of the changes will involve the BFagent.

"Random.h/.m" define and implement random number generators that can generate a double uniform in [0,1) (drand( )), a double normally distributed with mean 0 and variance 1 (normal( )), a double uniform in [-1,1] (urand( )), and an integer between *0* and *n-1*, where *n* is an argument to the function (irand(int n)). It also can generate a random three letter "name" (randomName(char*)). The name generator is used for agent names in the straight Objective-C version. These generators are implemented within a "Random" class in the straight Objective-C version. The Swarm version implements them as global functions.

The seed for the generators is set using the randset(int seed) function. This function sets the random number seed to the integer *seed*, or if *seed = 0*, the seed is set randomly according to the system clock. The function returns the seed.

8) speclist.h, speclist.m / Specialist.h, Specialist.m

These files declare and implement the Specialist class (where the specialist is also known as the market maker). The Specialist sets the price, ideally to clear the market (match the total number of bids and the total number of offers). The straight Objective-C version has five different specialists whereas the Swarm version has three. There is only one instance of the Specialist in the simulation, and its type (one of five or one of three depending on the version) is chosen at the beginning of the simulation. In the straight Objective-C version an event can be scheduled to change the type. Each specialist is different (and will be explained), but they all follow the same procedure.

1. Set a trial price.
2. Send each agent a –getDemandAndSlope: forPrice: message and accumulate the total number of bids and offers at that price.
3. In some cases go back to step 1, and formulate a new trial price.
4. Return the last trial price.

Once the price is set, the Specialist instructs the agents to update their *profit*, *position*, and *cash* total.

The five types of Specialist available in the straight Objective-C version are the Rational Expectations (*RE*) Specialist, the Adaptive ETA (*ADAPTIVE*) Specialist, the Fixed ETA (*FIXED*) Specialist, the Variance/Covariance ETA (*VCVAR*) Specialist, and the Slope (*SLOPE*) Specialist. The three types included in the Swarm version are the RE Specialist, the Fixed ETA Specialist, and the Slope Specialist. These specialists are included without change. Each specialist has different strengths and weaknesses depending on the agent.

The RE Specialist sets the price without iteration to be a fixed function of the dividend. This is the Specialist that is used when agents are homogenous and have rational expectations. The Specialist ensures that those rational expectations are correct by setting the expected price. The result is a static market without trade. This confirms that the model can support the predictions of the standard neoclassical model. This specialist is used only for this purpose.

There are three specialists that use the parameter *eta* to adjust the price. They are the Adaptive Specialist, the Fixed Specialist, and the VCVAR Specialist. The Fixed Specialist does not adjust the *eta* parameter throughout the simulation. The Adaptive Specialist adjusts the *eta* (if the price has moved in the same direction for the previous five periods) by a fixed factor called the *etaincrement*. The VCVAR Specialist sets the *eta* to be the negative ratio of the price's variance to the covariance. Each of these Specialists sets the initial trial price to be last period's price, collects bids and offers at that price, adjusts the *eta* (if VCVAR or Adaptive), and then sets a final price according to

*price = oldprice * (1.0 + eta * (bidtotal - offertotal)).*

That is the Specialist takes the old price and adjusts it by the difference in the total number of bids and offers weighted by the *eta*. The entire process is completed in two steps.

The Slope Specialist iterates the price until the difference between the total number of bids and the total number of offers is small enough (parameterized by *minexcess*). She does so by accumulating the slope and bid/offer of each agent at the current trial price. If the difference is small enough the price becomes the current trial price, otherwise the trial price is adjusted according to

trialprice = trialprice – imbalance / slopetotal,

where *imbalance = bidtotal – offertotal* and *slopetotal* is the sum of the slopes of each agent's demand function.

In the straight Objective-C version, the Specialist takes its parameters (including the parameter determining Specialist type) from the file "mparams" (for market parameters) at the beginning of the simulation. The general market parameters also come from this file. In the Swarm version, all Specialist parameters can be changed on the GUI. In batch mode the parameters come from the file "param.data" along with all other user controlled parameters.

9) (w)World.h, (w)World.m

The World class is very similar between the two versions. As was explained in **Section 2.1.2.1.**, each forecaster that a BFagent holds consists of a fixed number of bits. Each bit corresponds either to a technical or a fundamental trading rule. The World holds a string of bits that corresponds to the actual state of each of these rules. There are 60 rules in the simulation. These rules are defined in "World.h/.m". We describe each in **Section 2.2.2.3.** under the "names" file. Rules range from, say, "the dividend went up this period" to "the 100 period moving average of price is greater than the 500 moving average of price". After the realization of the price and dividend in the current period, the World obtains the information about the dividend from the Dividend object and the information about the price from the Specialist object and holds this information. All other objects retrieve these pieces of information from the World. The relevant control module then calls on it to update its state. Recall that at this time, the new dividend is realized but the price is the one that was realized based on the trading process that took place in the previous period. In turn the rules for predicting period $t+1$ are based on the current realization of the price and dividend in period t. The World has messages to send the price and dividend.

The World retrieves all its parameters in the straight Objective-C version from the "mparams" file (the same file from which the Specialist obtains its parameters). These include the interest rate and the parameter dictating whether the moving averages are weighted (*exponential*) or are just a straight average (*uniform*). A uniform moving average is just the average of the variable of concern over the period being considered. Exponentially weighted, however, the MA is

$$ma' = a * ma + b * value,$$

where $a = exp(-1/N)$ (with N the period of the moving average), $b = 1-a$, ma' is the new moving average, ma is last period's moving average, and *value* is the new value of the variable being considered.

2.2.2. Different Files

One attractive aspect of the Swarm version is that it significantly cuts down on the amount of code needed to run the simulation. Swarm was designed precisely to this end. It has built in collection classes, scheduling classes, and control classes (at the simulation level [back-end] and the user level [front-end]). Some of the files that the straight Objective-C version has that the Swarm version does not are amenities and the files that support these amenities (e.g., additional agents and their parameters), but a number of them are control files hidden from the user of Swarm. Firs all other agents will be described, the necessary control files will be compared and contrasted, and finally a lengthy discussion of the straight Objective-C parameter files will commence. It is in the latter where much of the user documentation for that version resides.

2.2.2.1. Additional Agent Files

	Objective-C	**Swarm**
1)		ANNagent.h, ANNagent.m
2)	bsagent.h, bsagent.m	
3)	duagent.h, duagent.m	
4)	ffagent.h, ffagent.m	

1) ANNagent.h, ANNagent.m

Each ANNagent predicts the next period's price plus dividend using an Artificial Neural Network (ANN). The ANN that each agent is endowed with is a simple backpropagation feedforward network with one hidden layer with a sigmoidal squashing function. Agents are heterogeneous in the sense that they have a random number of hidden units in their hidden layer bounded below by 3 and bounded above by 3 + *maxHidden*, where *maxHidden* is a parameter. Each agent's weights are initially randomly selected, so that with measure one no two agents with the same network structure will begin with the same initial conditions.

The inputs to each agent's ANN are the same and are fixed throughout the simulation. The inputs are the 5 and 100 day moving averages of price (normalized by the price, *pmav1* and *pmav2*), the 5 and 100 day moving averages of the dividend (normalized by the dividend, *dmav1* and *dmav2*), the current trading *volume*, the natural log of the current period's price divided by the previous period's price (the *priceRate*), the Moving Average Convergence/Divergence (MACD, parameterized by *macdParam*) between the 5 period moving average of price and the 20 period moving average of price (*MACD1*), the MACD between the 5 and 100 period MA of price (*MACD2*), and the MACD betweeen the 20 and 100 period MA of price (*MACD3*). A natural extension would be for agents to select their inputs, perhaps according to a string of bits like those of the BFagent.

ANNagents make decisions at the same time as any other agent, but they update their forecasting rule in every period according to the backpropagation algorithm. Since the ANNagent does not have a forecast rule with an associated demand function, as does the BFagent, the ANNagent cannot report a slope. In turn, the ANNagent requires that the Specialist be of the Fixed ETA type (since the Swarm version does not contain the Adaptive or VCVAR Specialists). If the Slope Specialist is selected when ANNagents are present in the simulation, the simulation automatically reverts to the Fixed Specialist.

2) bsagent.h, bsagent.m

As was mentioned before, the BSagent (the Bit-String agent) is a special case of the BFagent. Each BSagent has a number of forecasters (i.e., strings of bits where each bit represents a trading rule) that function the same as any forecaster of the BFagent. The BSagent also has a genetic algorithm that changes its forecasters over time like with the BFagent. However, the BSagent only makes a binary buy/sell decision as opposed to one based on a linear demand function like that of the BFagent. The first paper on the ASM, "Artificial Economic Life: a Simple Model of a Stockmarket" by Palmer, Arthur, Holland, LeBaron, and Tayler in Physica D, 1994, primarily used this agent. The BFagent was developed later to correct some problems inherent with the binary buy/sell decision. The BSagent's parameters are located in "bsparams".

3) duagent.h, duagent.m

The DUagent (or dumb agent) has a randomly assigned bid size between –100 and 100. If one particular bit (selected at the beginning of the simulation and the same for all DUagents) is on in a given period the demand of the agent is their bid size, otherwise it is the negative of their bid size. This agent is intended to be used only to display the capabilities of the simulation (or for testing). Some users may find it interesting to include this agent along with others in a simulation for performance comparisons. Naturally a more sophisticated agent should prove more successful. The DUagent's parameters are located in "duparams".

4) ffagent.h, ffagent.m

The FFagent has a parameterized number of forecasters that are also bit-strings like those of the BSagent and BFagent. Each agent randomly chooses this number of forecasters from a fixed set of forecasters. These forecasters are immutable throughout the simulation (i.e., the agent does not have a genetic algorithm). The forecaster that is used in each period is determined in the same way as with the BFagent (by selected the forecaster with the highest strength). The FFagent is risk averse, and by a simple argument has a linear demand function like that of the BFagent. The BSagent, recall, only makes binary buy/sell decisions based on multiple bit-string forecasters that are subject to change, whereas the FFagent makes a linear demand decision based on multiple forecasters that are not subject to change. Each forecaster has two variables for the prediction: *a1* and *a2*. The forecast is then given by

$$forecast = a1 * dividend + a2.$$

This is much simpler than the BFagent.

The BFagent has a number of forecasters that are all subject to change, and to this extent it is a combination of the BSagent and FFagent. The FFagent does not work very well except in a mixed market of other agents. Their risk aversion makes them all want to be out of the market at first, unless they have other agents to which they can sell. In conjunction with the slope specialist, they often induce wild price fluctuations – once any large fluctuation occurs the accuracy of their forecasters degrades, further driving wild behavior. This is probably fairly easy to fix, but the project's attention turned instead to the BFagent where the bit conditions lead to considerably more stable behavior. The FFagent's parameters are located in "ffparams".

2.2.2.2. Control and Utility Files

	Objective-C	Swarm
1)	amanager.h, amanager.m	
2)		ASMBatchSwarm.h, ASMBatchSwarm.m

3) ASMModelSwarm.h, ASMModelSwarm.m
4) ASMObserverSwarm.h, ASMObserverSwarm.m
5) config.h
6) control.h, control.m
7) error.h, error.m
8) global.h
9) nametree.h, nametree.m
10) procols.h
11) schedule.h, schedule.m
12) util.h, util.m
13) version.h, version.m, newversion

1) amanager.h, amanager.m

The agent manager files (found only in the straight Objective-C version) create and maintain the list(s) of agents. One instance of the AgentManager class is created in "control.m". It manages the lists of agents (categorizing them by class, type, and individual), dispatches certain messages to all agents, and computes some agent-wide distributions. It also manages the selection of a new generation of agents when evolution of the agent population occurs. Evolution is invoked by scheduling an event in the "timelist" file (see "schedule.h,.m" in this section and **Section 2.2.2.3.** under "timelist"). The list of agents is a doubly linked list. The Swarm version's way of handling lists is better encapsulated and thus of less concern to the user. The straight Objective-C version has the advantage that not only can there be multiple types of agents in one simulation (which is easy to implement in the Swarm version), but there can be multiple types of the same class of agent with different paramaterizations. The agent manager initializes and controls, then, the classes of agents in use, the types of each of these agents in use, and how they evolve over time.

The main messages of the AgentManager class are –makeAgents, –createAgent, and –getLine. The –makeAgents message creates the list of agent classes, types, and agents. It uses the –createAgent message to add an agent of a certain class and type to the actual agent list. This list is linked to the class and types lists. The number of agents of each type and class are read in –makeAgents from a user defined file called "agntlist" (called by the variable name *agentlist* in the –makeAgents message). This file is described in **Section 2.2.2.3.** below. Again, we use the name "agentlist" for the file since this is the file included with the demo. The user can change the name of the file in the control parameter file "mcontrol". The –getline message reads a line from the "agentlist" file. Each line in the file includes an agent class, the parameter file that is to be used for this class of agent (giving potentially multiple types of the same class), the number of agents of this class and type, and the initial asset holding of this agent. In turn, the "type" represented by a line is determined by the parameter file and the initial asset holding. For example, there could be two lines of class BFagent (each with, say, 25 agents) that use different parameter files and/or different initial asset holdings.

The AgentManager also has the message –evolveAgents. This message evolves the population of agents, constructing a new list with membership proportional to fitness. Agents no longer needed are removed by sending them the –free message (in "agent.h/.m"). Clones of old agents are created by sending them the –copy message (also in "agent.h/.m"). If this is to be called it must be scheduled in the "timelist" file from which it will be added to the schedule of events (this section under "schedule.h,.m" and **Section 2.2.2.3.**). Likewise, the AgentManager can be instructed, via the –level message, to reinitialize all agents' wealth, cash, and position. This is also scheduled via "timelist".

2) ASMBatchSwarm.h, ASMBatchSwarm.m

ASMBatchSwarm is the set of main control files for the Swarm version running in batch mode (i.e., without the GUI). It defines a subclass of the class Swarm called the ASMBatchSwarm. This class is initialized by main, at which time it initializes everything that the GUI version initializes (e.g., the ASMModelSwarm), with the exception of the graphical objects owned by ASMObserverSwarm, and additionally makes a call to two parameter files: "batch.setup" and "param.data". These files are loaded using a Swarm object called the "ObjectLoader". The ObjectLoader does not need to be initialized. The "batch.setup" file holds two control parameters owned by the ASMBatchSwarm: *loggingFrequency* and *experimentDuration*. The former determines how often data is written to the specified output file (1, for example, represents data being written every time step). The latter determines the number of periods the simulation will run. The "param.data" file holds all of the parameters that the user can control on the GUI. The simulation runs in batch-mode by running the executable with the –batchmode flag (i.e., *asm –batchmode*).

As was mentioned in **Section 2.2.1.** during the discussion of "Main.m", main always creates an object called *theTopLevelSwarm*. In batch mode *theTopLevelSwarm* is of type ASMBatchSwarm. After initialization, the ASMBatchSwarm is told to –buildObjects. This message creates an instance of the ASMModelSwarm, loads the parameters just mentioned, tells the ASMModelSwarm to prepare to write data and to build all relevant objects (the ASMModelSwarm's –buildObjects message). After building all relevant objects, the ASMBatchSwarm is told to –buildActions. This involves telling the ASMModelSwarm to –buildActions as well as building a schedule, called the *displaySchedule*, for the writing of data, paramaterized by the *loggingFrequency*. The –buildActions message also creates a *stopSchedule*. The ASMBatchSwarm is finally told by main to activate and ultimately to –go.

3) ASMModelSwarm.h, ASMModelSwarm.m

The ASMModelSwarm (a subclass of the Swarm class) is the main backend control class in the Swarm version. It maintains instantiations of virtually every object in the simulation (though is owned itself by *theTopLevelSwarm*, which is either an

ASMObserverSwarm or an ASMBatchSwarm, defined in main). When an object owned by the ASMModelSwarm needs information from another object owned by the ASMModelSwarm, it receives a copy of (a pointer to) that object from the ASMModelSwarm. In turn there are many messages like –getWorld: (World *) defined in various objects. That is to say, the ASMModelSwarm owns the real instantiation of the World class. Most other classes (also owned by the ASMModelSwarm) need information the World holds (like price and dividend), and in turn they must received a copy of (read pointer to) the World so that they can ask the World for whatever variables they need.

The main messages defined for the class ASMModelSwarm are: +createBegin (a class object), –buildObjects, and –buildActions. It also holds a number of messages returning parameters and objects it holds for setting up output files. The message +createBegin initializes the object. It sets all simulation parameters to their default values (defined in the ASMModelSwarm) and initializes an instance of a "ProbeMap". The ProbeMap is a map of all the parameters that dictates the user's ability to change parameters on the fly at the beginning or in some cases throughout the simulation through the GUI. In GUI mode, when the ASMObserverSwarm initializes the ASMModelSwarm, the ProbeMap is translated into a graphical object that the user can control and is set initially to the default parameter settings located in "ASMModelSwarm.m". In batch mode, however, the ASMBatchSwarm creates an instance of the ASMModelSwarm (along with its default parameters), but these parameters are over-written by the parameters read from "param.data"; the ProbeMap is never created graphically. The instance of the ProbeMap is called *probeMap*. A typical call to the *probeMap* to create an instance of a probe on the map is as follows:

```
[probeMap addProbe: [probeLibrary getProbeForVariable: "numBFagents"
                                  inClass: [self class]]];
```

The message –buildObjects creates all objects that the ASMModelSwarm holds. It calls the global function declared and defined in "random..h,m" called randset(int seed) to set the random number seed (see **Section 2.2.1.**). Next it creates instances of the Dividend class (called *dividendProcess*), the World class (called *world*), the Specialist class (called *specialist*), and the Output class (called *output*). It also creates a list called *agentList* and adds the appropriate number of agents to the list according to the parameter *numBFagents*. If there were other classes of agents, the ASMModelSwarm would add them to the same list. The list of agents is provided to the *specialist*.

More specifically, all the objects that are created here are subclasses of the SwarmObject class, and hence they inherit messages called +createBegin: aZone and –createEnd. These are the higher level Swarm messages that control object initiation. When each object is created its class calls +createBegin:. Then the object takes some parameters or objects from the ASMModelSwarm. In turn after creation the object calls a number of functions of the type –getSomeParameter: someType *parameter*. Finally the *object* calls –createEnd. An example of this object creation process follows.

```
dividendProcess = [Dividend createBegin: [self getZone]];
[dividendProcess setBaseline: baseline];
[dividendProcess setmindividend: mindividend];
[dividendProcess setmaxdividend: maxdividend];
[dividendProcess setAmplitude: amplitude];
[dividendProcess setPeriod: period];
[dividendProcess setDerivedParams];
dividendProcess = [dividendProcess createEnd];
```

The message –buildActions creates the list of activities (of the class ActionGoup) in the simulation. By this we mean that –buildActions creates a list of actions to be completed in *each* period. These actions are then added to a schedule potentially with other actions. This will be described below. The "main loop" in a Swarm simulation has the distinction of not being a loop at all. It creates the relevant *topLevelSwarm* that creates the ASMModelSwarm that builds the list of actions that is ultimately told to perform the list of actions until it is time to stop (or is told to stop). The loop is encapsulated in higher level Swarm objects. If different actions need to be performed at different times, Swarm holds the ability to implement "nested Swarms" (read threads) by constructing different schedules. In this simulation there is a simple nested Swarm. Since most of the agents (or rather all that exist in the Swarm version) rely on a history of prices and dividends, if the simulation begins without some mythical price and dividend history strange output results for a significant number of periods before the simulation settles down. To control for this, a stream of warm-up prices and dividends are created. Thus an instance of the class ActionGroup is created called *warmupActions*. These are run for a pre-determined number of periods specified by the schedule. The rest of the simulation main performs another list of actions given by the ASMModelSwarm's instantiation of the ActionGroup class called *periodActions*.

In each period where the group of actions defined by *warmupActions* is run, three messages are called. The ASMModelSwarm tells itself to –warmupStepDividend, which tells the *world* to set the dividend (–setDividend) to the new dividend simultaneously created by the *dividendProcess* (via the message –dividend). The *world* is then told to –updateWorld. This updates all moving averages and bits in the World's bit-string. Note, again, that the state of the *world* is updated before the price is realized. Thus the price lags behind the dividend by one period for most bits, though some particularly sensitive bits are set using the old dividend. This is discussed in **Section 2.1.1.**. After the *world* is updated, the ASMModelSwarm tells itself to –warmupStepPrice. This function tells the *world* to set the price to the "fundamental price", which is the price of the risk free asset. The fundamental price is defined to be *dividend/intrate,* where *intrate* is the interest rate (r) on the risk free bond.

The list of actions performed in *periodActions* and a description of each is as follows.

Step 1 → the ASMModelSwarm calls –periodStepDividend, which is identical to –warmupStepDividend as described under *warmupActions*.

Step 2 → now each agent is told to call –creditEarningsAndPayTaxes, which updates the agent's cash holdings and wealth.

Step 3 → the *world* is told to –updateWorld as above.

Step 4 → the ASMModelSwarm calls –prepareBFagentForTrading. This tells the class BFagent to prepare some common information used in the –getDemandandSlope:forPrice: message.

Step 5 → each agent is then told to –prepareForTrading. This message determines all *active* forecasts of each agent for this period. That is, each agent determines which of her forecasts is active this period. It also determines the demand slope and intercept of each of the active forecast rules. (Recall for the BFagent the prediction is given by $p_{t+1} + d_{t+1} = a * (p_t + d_t) + b * d_t + c$). For agents with only one forecast rule (e.g., the DUagent and ANNagent) –prepareForTrading merely returns self.

Step 6 → now the ASMModelSwarm calls –periodStepPrice. This message is identical to –warmupStepPrice described above under *warmupActions*.

Step 7 → here the *specialist* is told to –completeTrades. This sets the actual price according to the process described in **Section 2.2.1.** under the discussion of "(s)Specialist.h,.m". This process is dependent on the Specialist type chosen via parameter at the start of the simulation.

Step 8 → finally each agent is asked to –updatePerformance. Here agents update the quality of their forecasters and their resultant strength.

After the *warmupActions* and *periodActions* have been created, four different objects of the class Schedule are initialized: *warmupSchedule*, *startupSchedule*, *initPeriodSchedule*, and *periodSchedule*. The *warmupSchedule* consists of the *warmupActions* repeating every period. The schedule is executed when the function warmup(id *warmupSchedule*) is called by the *startupSchedule* at time $t=0$. The function starts the *warmupSchedule* and terminates it after 501 periods. From the perspective of the simulation's time, all warm-up periods have been completed by time 0. The *startupSchedule* then initializes the *initPeriodSchedule* by calling the function initPeriod(id *initPeriodSchedule*). This function runs the *initPeriodSchedule* for one period only (though it may be that this schedule is superfluous and could be included in the *periodSchedule*). The schedule consists of the *periodActions*. It is both started and terminated in period 0. Finally the *periodSchedule* is activated. All actions are not actually run until the *topLevelSwarm* tells the ASMModelSwarm to call the –activateIn: message.

4) ASMObserverSwarm.h, ASMObserverSwarm.m

The ASMObserverSwarm is a subclass of Swarm's GUI Swarm class. It is invoked by main when the simulation is run in GUI mode (as was previously discussed in **Section 2.2.1.** under "Main.m" and in the current section under "ASMBatchSwarm.h/.m"). Recall that when the instance of ASMModelSwarm is initialized it creates a ProbeMap for all simulation parameters. Likewise when the ASMObserverSwarm is initialized by main the ASMObserverSwarm creates another ProbeMap for a display parameter and some input/output commands. The display parameter is called *displayFrequency*; it dictates how often the data is displayed on the graphical objects that will also be created. The default setting is 3, since the graphical objects can be sluggish. The user can control this parameter, as is implied by its placement on a ProbeMap. The input/output commands are *writeSimulationParams* and *writeSimulationData*. *If the user wants to write the data for a particular run,* writeSimulationData *must be selected before the start of the simulation. However,* writeSimulationParams *can be selected at any time during the simulation. In this way the user can identify by the graphical output whether a run is worth replication before having the simulation write the data.* This will be discussed in more depth in **Section 4.2.** of the documentation on "Using the Swarm GUI".

After initialization, main directs *theTopLevelSwarm* (which in the graphical case is an ASMObserverSwarm) to build objects. The ASMObserverSwarm creates an instance of the ASMModelSwarm and creates both the ProbeMap just described and the ProbeMap of the ASMModelSwarm. Additionally, each Swarm simulation creates a control panel with the buttons *Start*, *Stop*, *Next*, *Save*, and *Quit*. This all occurs in higher level Swarm objects and is invisible to the user. The function of these buttons will be discussed in the user portion of the documentation (**Section 4.2.)**.

After the various ProbeMaps have been created, the ASMObserverSwarm also builds the graphical objects that display data throughout the simulation. In the Swarm ASM the following graphs are available: "Relative Wealth of Agents", "Agent Position", "Volume v. time", and "Price v. time". The first two are Histograms and the second two are line graphs, called ActiveGraphs in Swarm.

The ActiveGraphs update automatically (i.e., without user defined function) when they are told to –step in the *displaySchedule* that is created when –buildActions is invoked. They are created, and told which object holds the data and the message to use to retrieve that data. All this is specified when the ASMObserverSwarm executes –buildObjects. After every period of length *displayFrequency* this message is sent to the relevant object and the graph is updated. An ActiveGraph can display multiple streams of data. In this simulation, "Volume v. time" shows the volume of the asset traded in each period, whereas "Price v. time" shows both the actual trading price per period and the fundamental price in each period (defined above as the dividend divided by the interest rate).

The Histograms require data as to the number of "bins" to be displayed. In both ASM Histograms the number of bins is the *numBFAgents*. The "Relative Wealth of Agents" Histogram measures the wealth of each agent divided by the agent's initial cash holdings in the display period. The "Agent Position" Histogram measures the asset holding of each agent in the display period. Since each bin corresponds to a different piece of data, the data is updated by a user-defined message that is called in the *displaySchedule*. The message is called –updateHistos. This message merely asks each agent to report her position and wealth. These fill arrays of doubles of length *numAgents* that correspond to the bins in the respective Histograms.

The *displaySchedule* is built when main tells (in GUI mode) the ASMObserverSwarm to –buildActions. Naturally, the ASMObserverSwarm first instructs the ASMModelSwarm to build its actions. It then builds the *displaySchedule*, which consists of an instruction to both ActiveGraphs to –step, an instruction to itself to –update, and an instruction to the ProbeDisplayManager (the object that manages the ProbeDisplay) to update. Finally the simulation is activated and run as described above.

5) config.h

The file "config.h" is the first we will discuss in a series of utility and control files in the straight Objective-C version. This file is included in almost every source file via "global.h" (which includes a number of global variable declarations to be discussed). It defines certain preprocessor variables to adapt the code to the supported architectures. If you have Swarm running on a platform, then you can run the straight Objective-C version. There are preprocessor definitions currently defined for Solaris, Linux, SunOS-4, DJGPP for MSDOS, Windows9x, WindowsNT (tested only on NT), and NextStep. Implementation on these platforms will be described later in the Supported Platforms section (**Section 3.**). There we will describe the steps that must be taken to run the simulation on platforms not included here (e.g., Cygwin32).

6) control.h, control.m

"Control.h" and "Control.m" define and implement the global functions that run the simulation. "Control.m" also defines the *global* objects that are involved in the simulation. These are the same as the objects held by the ASMModelSwarm in the Swarm version in addition to a global *scheduler* object of type Schedule (to be described below).

As was described in **Section 2.2.1.** under "main.m" above, the first argument called in "main.m" that is defined and implemented in "control.h,.m" is setEnvironment(argc, argv, filename). This function first acquires arguments from the command line (e.g., a control parameter file other than the default or debugging flags using the –dxxx option) using a function (that is based on the publicly available Decus .cpp) called doooptions(argc, argv). The possible debugging command line flags are described in more depth under the section on "mcontrol" (**Section 2.2.2.3.**). The default market control filename is "mcontrol", but the user can specify a different filename on the command line. The function also writes the flags that have been called (via its own writeUsage(stdout, argv[0]) function), tells the *output* object to write the names of all bits defined in the World's bitsring (via the Output class's –writeNamesToFile: message), and establishes the current *runid* by calling its own getpreviousrun( ) function. Each run saves its *runid* when the startup( ) function (described below) is called. The getpreviousrun( ) function extracts this information from the "runid.txt" file, if it exists. After extraction, the *runid* is incremented. The *runid* aids in the naming of the output and log files. Finally, the control parameter file is defined. If a filename was specified on the command line it is set to be the control parameter file. Otherwise, the default "mcontrol" is specified. The demo provided with the release includes a file of this name. The path of the control parameter file (hereafter "mcontrol") is determined and all other parameter files throughout the simulation are searched for in that directory.

Next, in main, the control's startup( ) function is called. The startup( ) function first saves/writes the *runid* with the function setrunid(*runid*), defined in control. Next the "mcontrol" file is read using the function controlReadParams(*cparamfile*). The file "mcontrol" contains the random number seed (potentially 0 indicating a random seed), the names of all other parameter files and the name of a log file. These are:

dparamfilename – parameters for dividend process, demo file is "dparams";
mparamfilename – parameters for market & specialist, demo file is "mparams";
agentlistfilename – list of agents in market, demo file is "agntlist";
outputfilename – specifications, demo file is "output";
timefilename – list of times to produce output or display, demo file is "timelist";
logfilename – log of parameters, demo file is "log###".

The log file name is then renamed with the *runid* and version number ("log<runid>"). The version number is obtained by calling the global function versionnumber( ) defined and implemented in "version.h" and "version.m", respectively. Next the random number generator is initialized with the seed from "mcontrol".

Each object in the simulation initialized in "control.m" (*dividendProcess*, *specialist*, *agentManager*, and *output*) is then initialized (*dividendProcess*, *specialist*, *agentManager*, and *output*) via their respective initialization messages. These objects are global and hence are available to all objects in the simulation. Both the *dividendProcess* and *specialist* are initialized with the message (defined for the respective classes) –initFromFile: *parameterFilename*. The *output* object is initialized with the function outputReadParams(outputfilename) defined in "market.m". The output parameters take a special form that will be described in **Section 2.2.2.3.**. The *agentManager* parameters also take a special form. Each line in its parameter file lists an agent class, a number of agents, a parameter file to be used with that class, and initial holdings, whereby multiple types of an agent class can be specified. The *agentManager* calls the function –makeAgents: *agentlistfilename*, which creates the agent list with parameters from the filenames specified in "agntlist". This is also described in the current section, **Section 2.2.2.2.** under "agentmanager.h/.m". The *world* is initialized via the message –initWithBaseline: *baselineParameter*, which takes a parameter specified in the Dividend's parameter file ("dparams") called the *baseline*. The initialization returns the maximum moving average history (e.g., 500 period moving average). This becomes the length of the warm-up period (which is fixed at 501 in the Swarm version). The initial simulation time, t_0, is set to be the negative of this value in the straight Objective-C version.

Finally in startup( ), an instance of the Scheduler class (*scheduler*) is initialized with an –initFromFile: *timefilename* message. The "timelist" file also takes a special form that will be described in the current section (**Section 2.2.2.2.**) under "Schedule.h/.m" and in depth in **Section 2.2.2.3.** under "timelist". Basically, the file designates times at which certain events should occur. For example, certain debugging actions could be scheduled to take place every 1000 periods or alternatively the price could be written to the appropriate file every period. The *scheduler* also provides the final simulation time (called *lastttime* in main and control). This is based on the maximum time specified in the "timelist" file. So if one event should happen every period for 5000 periods and another event should occur every 100 periods up to 3000 periods, then *lasttime* would be 5000. After the schedule is specified, the *output* object opens the output files specified in the "output" parameter file and writes the appropriate headings to them. Finally all parameters are written to the log file and all input files are closed.

Main next calls performEvents( ), since some of the events (specifically the output events) scheduled from the "timelist" file may need to be executed even before the warm-up period. The function performEvents( ) simply executes anything that exists on the schedule. This function is also called after each call of either warmup( ) or period( ) (which occurs in each period after startup( )). This function simply executes the actions that were included in the "timelist" file and were recorded by the *scheduler*. For example, if in a particular period the agents of some type and class are to evolve, performEvents( ) will instruct the *agentManager* to perform the message –evolveAgents. In what follows, keep in mind that after every call of warmup( ) or period( ), performEvents( ) must be called by main.

At this point in time the initial value of the global variable t has already been set (usually –500), and the main loop begins (and is explicit as opposed to the Swarm version). Time runs from the initial t to *lasttime* (from the "timelist" file). Until period 0, the function warmup( ) is run. This function, defined in "control.m", does exactly what the set of *warmupActions* does in the Swarm version (with a few minor exceptions). First time is incremented, the *dividendProcess* is instructed to generate a new dividend, and the world obtains this dividend and updates its state. In the Swarm version this is done via the message –updateWorld (which calls another message –makeBitVector). In the straight Objective-C version most of the functions of –updateWorld are accomplished when the dividend is updated (in the –dividend message) and –makeBitVector is called separately. Regardless, the outcome is the same. Next the *world* sets the price to the fundamental price (defined above as the dividend divided by the interest rate). Finally, and this does not occur in the Swarm version, *output* "accumulates" output variables that need aggregation (e.g., an average across agents).

Once warmup( ) has been run until $t = 0$, the function period( ) is called (until *lasttime*). This also executes basically the same actions described in depth above in the present section "ASMModelSwarm.h/.m". During the function period( ), time passes; the *dividendProcess* realizes a new dividend, which the *world* retrieves; the *world* updates its state; each agent runs –creditEarningsAndPayTaxes; the agent types and individual agents get ready for trading (though the *agentManager* must call the class message –prepareForTrading, as opposed to the ASMModelSwarm in the Swarm version); the price is realized by the *specialist* executing its market clearing process; and agents update their forecasters' performance.

After *t* has reached *lasttime*, the simulation performs some closing functions (called in main but defined in "control.m"). The first of these is finished( ). This function calls the function writeEpilogue( ), which writes the market random seed, the dividend random seed, the simulation time, and the log file name to the file "runid.txt". Next the function marketDebug( ) is called. This asks the *agentManager* and *world* to check some data. Finally closeall( ) and exit(0) are called to close the output files (by instructing *output* to do so) and to exit the simulation.

7) error.h, error.m

These files define and implement a number of global error handler functions that complement the standard Objective-C error messages. All Objective-C objects have access to the message –error:. If an object encounters an error in one of its own messages, it makes a call to this message which calls the function objcerror(*object*, , ,) defined in "error.m". This function determines the class of the object calling the error and then calls the commonerror(*class*, , ,) function that writes the error to the error buffer. Errors in objects are "fatal errors" and cause the termination of the program via the abandonIferror( ) function (declared in "error.h" and defined in "error.m", but called in the object itself after an error has occurred) and the fatalerror( ) function defined in "main.m". If an error is encountered in a non-object portion of the code, the error function cerror(*routine*, ,) is called. Such an error is also a fatal error. Non-fatal errors are encountered in the processing of input files. They are non-fatal in that the program waits until the entire input file is processed (logging the errors as it goes) before determining whether the error(s) was (were) fatal. This is accomplished using the saveError( ) and abandonIfError( ) functions. All errors (fatal or non-fatal) are written to the file "stderr" using the specific error's message( ) function.

8) global.h

This file, in one way or another, is included in virtually every source file. It defines some constants and some global variables. The straight Objective-C version was originally written for the NextStep platform for use with NextStep's built in graphical widgets. The original version was written for notoriously slow computers. To provide more efficiency, many important variables that are used by a number of objects were defined globally. Naturally this subverts one of the chief attractive features of an object-oriented programming language. The Swarm version corrects for this; there *all* variables are owned by some object. For example, all objects (*specialist*, *world*, *dividendProcess*, etc.) that are controlled by "control.m" are declared in "global.h", whereas in the Swarm version the ASMModelSwarm owns each of these. Additionally, the *price, dividend*, *volume*, etc. are declared in "global.h"; in the Swarm version these are owned by the World. Naturally this encapsulation can be cumbersome (calling for myriad messages of the type –getVariableFromSomeObject), but it does protect the integrity of crucial variables (e.g., *price*), so be careful when using the straight Objective-C version.

9) nametree.h, nametree.m

An object of this class manages a hierarchical tree of names, such as a directory structure or an ancestry tree (for agents). Each name is up to *MAXCHARS* characters long, where *MAXCHARS* is a compile-time parameter. Each node in the tree may have one or more "children". A node without children is called a leaf node; other nodes are branch nodes. Leaf nodes can also store a non-negative integer value "v". Initially all nodes are descended from a single root node, but automatic pruning may subsequently lead to "orphan" nodes whose parentage is no longer known. The topological structure is thus a set of sub-trees, with at most one tree descending from the root node and possibly others descending from orphan nodes. For multiple root nodes, use multiple instances of this class. Nodes in the tree are referenced by an (integer) "index". The index of the root node is always 0; other indices are positive. Memory is allocated automatically. The root node has the null name " ", and is always treated as a branch node (even if it has no children).

The NameTree class is included only in "agentmanager.m". Each agent of a class and type hold one instance of the NameTree class. For example, if there are 50 BFagents of one type (i.e., with the same initial holdings and parameter values), then their names are controlled by the same NameTree. When the genetic algorithm is run and different forecasters have offspring, the offspring's name/node is related to that of its parents (though the new agent is placed on the same level as its parents since genealogy does not play a role). In turn, as is often the case in the straight Objective-C version, this class is considerably more general than is needed for the ASM. The ASM only uses balanced trees in which every leaf is at the same level. This feature is not included in the Swarm version. If you are interested in the functionality of this class please see the code.

10) procols.h

This file defines messages that **were** used for the NextStep version with front-end to update the NextStep widgets. In the current straight Objective-C version these messages are never used, though "procols.h" is included in "amanager.m", "dividend.m", and "speclist.m". When messages from "procols.h" appear in these files they are called by an object called *marketApp*. This object is set to *nil* in "main.m" since the simulation now runs only in batch-mode and thus the messages are never called.

11) schedule.h, schedule.m

As was implied above, the *scheduler* does not really maintain the actual schedule that the simulation follows (like the schedules in the Swarm version), but rather the *scheduler* maintains a list of events that occur at pre-designated periods. These events are specified in the file "timelist", and they are loaded by a call to the –initFromFile: message declared in "schedule.h" and defined in "schedule.m". Events and their description are of the following types specified by the indicated parameter in the "timelist" file:

DISPLAY: "d" → displays a variable on the front-end. This is not available in the current purely batch version.
WRITESTREAM: "w" → writes a variable to an output file. All output streams specified in the "output" parameter file must have a corresponding "w" entry in the "timelist".
RESETSTREAM: "reset" → resets an accumulated stream (e.g., a sum).
ENABLEAGENT: "E" or "enable" → enables a particular agent or a group of agents.
DISABLEAGENT: "D" → disables a particular agent (can be re-enabled).
EVOLVE: "e" → evolves an entire population of agents (using the –evolveAgents message in "agentmanager.h/.m").
SHOCK: "s" → creates a dividend shock (using the –shock message in "dividend.h/.m").
RESETSHOCK: "r" → resets the dividend shock to 0 (using the –resetShock message in "dividend.h/.m").
LEVEL: "level" → reinitializes agents' cash, wealth, and position (using the –level message in "agentmanager.h/.m")
SET_SPECIALIST_PARAM: "specialist" → changes the Specialist type.
SET_DIVIDEND_PARAM: "dividend" → changes the Dividend type.
DEBUG: "debug" → debugs according to the debugging flags set on the command line or in the "mcontrol" parameter file.

In each period the current schedule of events is executed when main calls the global function performEvents( ) defined in "control.m". The function performEvents( ) calls the Scheduler's –(EventType)nextEvent message. The performEvents( ) function is the one that takes the type of event and sends a message to the relevant object. For example, if the event is of the type EVOLVE, performEvents( ) sends a message to *agentManager* to –evolveAgents. In this sense the Scheduler is purely a schedule with no agency of its own. It also has messages that allow it to report future scheduled events.

12) util.h, util.m

These files declare and implement a number of global directory/file functions, input functions, output functions, and miscellaneous functions (e.g., memory management, type conversion, etc.). There is nothing particularly pretty about these C-style functions, so interested users can spend countless hours of fun getting to know them. They are essential to the simulation, but are precisely what the filenames indicate they are – utilities. See the files themselves for detailed comments.

13) version.h, version.m, newversion

The functions in "version.h/.m" provide version information for use in the header of output files (and in the Info panel in old front-end version). The version number and version date are intended to be updated automatically each time the source is compiled. This occurs via the "makefile", which calls the "newversion" script to edit this file. The "newversion" script finds the *VERSIONNUMBER* defined in "version.m", updates it, and then actually edits the "version.m" file to update *VERSIONNUMBER*. The *VERSIONDATE* is automatically updated via the preprocessor flag ***__DATE__***, i.e., the *VERSIONDATE* is the date of compilation. The files "version.h/.m" implement two functions: versionnumber( ) and versiondate( ), which return the version number and version date, respectively.

2.2.2.3. Parameters and Parameter Files

This section describes the proper format for both versions' parameter files. The straight Objective-C version's files are named according to the example files included with the code as a demo. These files include extensive documentation that will for the most part be repeated here. The names of the files are not important, provided that the intended file is specified correctly in the control parameter files. For instance, there can be many BFagents with different parameter files, and these would be specified in the "agntlist" file, which could also have a different name. The agent list filename (default "agntlist") is specified in the market control file (default "mcontrol") and the market control filename is specified in "control.m" or alternatively on the command line. All straight Objective-C parameter files must be in the same directory as the market control parameter file. The market control file is searched for first in the current working directory and then (if different) in the directory where the program resides. Generally it is suggested that the program be run from the directory that includes all the parameter files (see **Section 4.**). Finally, for all parameter files, the parameters are recognized solely by location in the file, not by some flags. The user must be careful. The documentation lists them in their proper order as do the demo files.

As for Swarm, the parameter filenames must be as below. The parameters in these files are located by name (not location). Naturally there is no good reason to change the ordering. The order is the same as for the default parameters given in "ASMModelSwarm.m", "ASMBatchSwarm.m", and "ASMObserverSwarm.m".

	Objective-C	**Swarm**
1)	agntlist	
2)		batch.setup

3) bfparams
4) bsparams
5) dparams
6) duparams
7) ffparams
8) mcontrol
9) mparams
10) names
11) output
12) param.data
13) timelist

1) agnlist

This parameter file is loaded in the straight Objective-C version (only) by the AgentManager when it is told to call the message –makeAgents: (const char *)filename by the function startup( ) defined in "control.m". The file is read until its end. Each line represents a different class or type of agent and has four fields, separated by spaces or tabs:

Agent_class Number Initial_holding Parameter_filename

Possible agent types are BS (or Bit-String), BF (or Bit-string-Forecast), FF (or Fixed_Forecast), and DU (or Dumb). There is no restriction on the number of agents or on the initial holding, though the latter is usually 1 and the former is dictated by system resources. Different types of the same agent can be specified by providing different parameter files with different parameters. Examples include:

BF	10	1.0	bfparams
Bitstring	10	1.0	bsparams
BS	10	1.0	bsparams2
BF	10	1.0	bfparams2
DU	10	1.0	duparams
FF	10	1.0	ffparams

The filenames for the agent parameter files follow the same naming conventions described below in this section under "mcontrol".

2) batch.setup

This file includes the main control parameters for the ASMBatchSwarm: *loggingFrequency* and *experimentDuration*. The former dictates the number of time steps that pass before the simulation writes its data and the latter dictates the number of time steps before the simulation expires. Parameter files for the Swarm version always begin with @begin and end with @end. This file must be included in the directory with the executable file.

3) bfparams

This file includes all the parameters for the BFagent. As per the discussion of different agent types of the same class, there could be multiple “bfparams” files, e.g., “bfparams1” and “bfparams2”. Any “bfparams” file must include the parameters discussed below and included in the “bfparams” demo file. Parameters are recognized solely by their position in the file, not by some flag at the beginning of each line. Users must be careful. There are three sets of parameters specified. The first are general forecaster related parameters. They are:

numrules – number of forecasters per agent.
tauv – moving average healing time for forecaster variances.
lambda – the degree of agent risk aversion.
maxbid – the maximum bid or offer.
selectionmethod – the method by which to select between activated rules: best, roulette, or average.
mincount – the minimum number of observations before a forecaster is used.
subrange – this is used for initialization of the forecast parameters *a*, *b*, and *c*.
min and *max* values of the parameter *a* (the coefficient on p_t+d_t in the prediction equation).
min and *max* values of *b* (the coefficient on d_t).
min and *max* values of *c* (the constant term).
newfcastvar – variance assigned to a new forecaster.
initvar – variance of overall forecast for time steps $t < 200$.
bitcost – penalty parameter for specificity (i.e., non-hashed bits in a forecaster).
maxdev – max deviation of a forecast in variance estimation.
individual – whether to use individual forecast variances: yes or no.
bitprob – probability each bit is either 0 or 1 (i.e., non-hashed) initially.

Next is a list of the bits to be used in each forecaster. The user can either specify "allbits" or give a list of bit names terminated by "end". The available bit names are listed in the "names" file. The term "null" may be used within the list of bits to separate them into groups visually. You can use "p x" to change *bitprob* for subsequent bits to the value "x". Some examples include:

allbits
pup pup1 pup2 pup3 pup4 null p 0.5 dup dup1 dup2 dup3 dup4 end

Finally the file includes a list of parameters for the genetic algorithm. The GA parameters are:

gainterval – the average time between GA runs (i.e., the arrival time in a poisson process).
firstgatime – the first time the GA is enabled.
newfrac – the fraction of rules replaced.
pcrossover – the probability of crossover.
plinear – the linear combination "crossover" probability.
prandom – the parent crossover probability, random for each parent.
pmutation – the per bit mutation probability.
plong – the long jump probability.
pshort – the short (neighborhood) jump probability.
nhood – the size of the neighborhood.
longtime – the amount of time after which a rule is generalized (i.e., some 0/1 bits are changed to hash #) if it remains unused.
genfrac – fraction of 0/1 bits to change to hash (#) when generalizing.

4) bsparams

This file contains all the parameters for the BSagent. Everything that applies to the “bfparams” file applies here as well (e.g., different “bsparams” files can be specified for different types, parameters are recognized by position in the file, etc.). The only difference is the list of parameters. The forecaster related parameters are:

numrules – number of forecasters per agent.
bidsize – size of bid or offer.
selectionmethod – either best or roulette.
mincount – min observations before a rule is used.
taus – moving average healing time for rule strengths.
maxrstrength – maximum rule strength.
minrstrength – minimum rule strength.
initrstrength – initial strength for new rules.
bitprob – probability each bit is 0 or 1 (non-hash) initially.

The same rules apply to the list of bits used in each forecaster (i.e., “allbits” or a list of bits terminated by “end”). The GA parameters are:

gainterval – average time between GA runs (poisson process).
firstgatime – time at which GA is enabled.
newfrac – fraction of rules replaced.
pcrossover – probability of crossover.
pmutation – mutation rate per bit.
preverse – probability of reversal for weak rules.
longtime – generalize if a forecaster remains unused for this length of time.
genfrac – fraction of 0/1 bits to make hash when generalizing.

5) dparams

This file contains the parameters for the dividend process. They are loaded when the instance of the Dividend class (created in the startup( ) function, which is defined in "control.m" and called in "main.m") is told to call the message –initFromFile: (const char *)paramfile. The parameters are:

dseed – the random number seed for the dividend stream (0 for random, -1 to chain).
divtype – the type of dividend process (ar1, markov, white, iid, square, ramp, and sine as described in the **Section 2.2.1.** on the dividend process).
baseline – the baseline dividend (sets the scale; this is not to be changed).
mindividend – the minimum dividend.
maxdividend – the upper clipping point for the dividend.
period – the mean period or auto-correlation time. This can be replaced by *rho* for ar1. The parameter will be *rho* if it is less than 1.0 and *period* otherwise.
amplitude – the amplitude of deviations from the baseline. This can be replaced by *gauss* for ar1. If the period is less than 1, then it is *gauss*, otherwise it is *amplitude*.
asymmetry – range -1 to 1; 0 means symmetrical.
shocksize – size of dividend shocks.
shocktime – decay time-scale (*1/e*) of shocks.

6) duparams

This file holds the parameters for the dumb agent. All that applies to the parameters for the other agents applies here as well. This agent has only 2 parameters:

bidsize – how large a bid or offer to make (constrained by *mincash* and *minholding*).
keybit – specifies the condition that determines when the trader is to make a bid of size *bidsize*. If the condition is on, the agent makes a bid of size *bidsize*. In all other states the agent makes a bid of size –*bidsize* (the negative of *bidsize*). The bit names that can be used for the *keybit* parameter are listed in the "names" file. Using "random" for the *keybit* gives a noise trader.

7) ffparams

This file includes the parameters for the FFagent. The usual applies. The parameters are all forecaster related. They are:

numfcasts – the number of forecasters per agent.
tauv – the moving average healing time for variances.

lambda – the risk aversion parameter.
maxbid – the maximum bid or offer.
fcastmin – the minimum forecast for price plus dividend.
fcastmax – the maximum forecast for price plus dividend.
selectionmethod – either best or roulette.
beta – the strength scaling Boltzmann parameter.
maxdev – the clipping value for forecast error.
a1 – for the forecast = a1*dividend + a2.
a2 – for the forecast = a1*dividend + a2.

8) mcontrol

This file contains the main control parameters. It is read when startup( ) is called and within startup the function (defined in "control.m") controlReadParams(controlParamFile) is called. It has three sections: debugging flags, the random number seed for the market and agents, and the names of all other parameter files (save for the specific agents' parameter files names in "agntlist"). The user can either specify debugging flags on the command line using the –dxxx option, or she can specify them in the "mcontrol" file. The possible debugging flags are:

a – debug agents (depends on individual agent classes).
b – debug agent browser; report fixable inconsistencies.
c – show CPU time at certain times.
d – debug dividend process – show some internal parameters.
e – show each event returned.
f – show filenames and path computations (-df option shows more).
h – debug holding; check conservation of the total.
n – debug NameTree, show number of nodes in use.
o – debug Output.
t – debug types and classes – checks lists, reports a summary.
w – debug world; check moving averages and record-keeping.
A – turn on ALL of the above flags.
0 – no debugging; use if no other flags are specified.

Flags specified on the command line will be used in addition to flags specified in "mcontrol".

The random number seed follows the same rules as the random number seed for the dividend process (described under "dparams"). Specifying 0 indicates that the seed is random, -1 chains the random number, and using any other number fixes the seed to be that number.

Finally, the filenames specified in the filenames section are as follows (in order):

dparamfilename – parameters for the dividend process ("dparams").
mparamfilename – parameters for the market and specialist ("mparams").
agentlistfilename – list of agents in the market ("agntlist").

outputfilename – output stream specifications (“output”).
timefilename – list of times to produce output or other events (“timelist”).
logfilename – log of all parameters (“log<runid>”).

The filenames follow a number of rules and have a number of options.

1) Unless filenames start with / or ~/ they are taken to be relative to the directory in which the “mcontrol” file was found. Absolute and relative pathnames are fully implemented for Unix, but only partially for MSDOS or Windows (use \ for separators, but no drive letters), and not at all for VMS.
2) Surround the whole filename in double quotes if it includes spaces or #'s.
3) The symbol “-“ may be used to specify stdin or stdout.
4) The symbol “=” may be used for input files to mean the same file as the one specified on the previous line. Use of this on the first line (the *dparamfilename*) indicates that the parameters will be found in “mcontrol” following the control parameters (described in this section). This allows concatenation of all the input files into a single file, in the order listed here. The log file produced by a run is just such a file, and may be used as a single input file (in place of this one) to reproduce that run exactly.
5) Other input files may be needed for specific agents. These are specified in the “agntlist” file, and follow the same naming conventions as those here. They are read after the “agntlist” file and before the “timelist” file, in the order the agents are specified. If “=” is used for the “timelist” file, it means the same file as the “agntlist” file, which is not necessarily the same as the last such agent-specific file.
6) The symbol “=” may also be used for the output streams (except the first), thus merging their output.
7) The flag “<none>” may be used for output files to turn them off.
8) Output filenames may contain one or more #'s (e.g., “log###”), which are replaced by digits of the run number, from right to left. In this case the filename **must** be enclosed in double quotes. Output filenames may also be prefixed by a “+” to imply append-mode (append to an existing file) and/or by a “*” to suppress the normal heading block at the start of the output.

9) mparams

This file is read in startup( ) when the Specialist object is given the message –initFromFile: paramFile. This file has three sections and follows all the rules specified above. The first section is general market parameters, the second is the moving average type, and the third is specialist parameters. The market parameters are:

intrate – interest rate for the fixed-asset (bond).
initialcash – initial cash holdings for each agent.
maxprice – maximum price.

minprice – minimum price.
taup – moving average healing time for profit.
minholding – minimum holding per agent (<= 0).
mincash – minimum cash per agent, <= 0 (borrowing constraint).

The *matype* (moving average type) governs how the price and dividend moving averages are calculated. The N-period moving averages (MA's) can be weighted in one of two ways:

1) uniform – the average of the last N values.
2) exponential – *ma' = a * ma + b * value*, with *a = exp(-1/N)* and *b = 1 - a*.

The specialist parameters are:

sptype – specialist type (*fixed* – fixed eta, *adaptive* – adaptive eta, *re* – rational expectations for special trial case, *vcvar* – based on the variance-covariance (elasticity), *slope* – iterative, based on agent-supplied slope). The specialist type can be changed by including an event in the "timelist" file.
maxiterations – max number of price adjustment passes.
minexcess – target for |bids - offers| for iterative specialists.
eta – amount by which price changes per bid or offer for the eta specialist.
etaincrement – amount by which *eta* changes per step for the eta specialist.
etamax – maximum value of *eta* when the specialist is of type adaptive eta.
etamin – minimum value of *eta* when the specialist is of type adaptive eta.
rea – dividend multiplier for the RE price with a re specialist.
reb – constant offset for the RE price with a re specialist.
taulpv – healing time for variance/covariance MAs.
ldelpmax – max |*log(price/oldprice)*| in elasticity calculation.

10) names

The simulation does not use this file. It is a list of the variable names and keywords that can be used in output specifications. It is found in the "src" directory. We repeat it here for convenience. Integer variables and bits need integer output formats (e.g., %d) unless they are averaged with "@" or AVG. Real variables and any averaged variables or bits need double output formats (e.g., %f). Items marked with a "*" below change their meaning according to the context in which they are used (see the documentation for output specifications in this section). Use the "–n" option on the command line to produce this listing.

i) Integer variables

1 – 1.
t – time.
oldt – time – 1.

run – run number.
mseed – random seed for market and agents.
dseed – random seed for dividend stream.
nagents – number of agents in class, type, or all.*
nclasses – number of classes.
ntypes – number of types in class, type, or all.*
generation – evolutionary generation.
classnum CLASS – class number.
typenum TYPE – type number.
nrules TYPE – number of rules or forecasters.
nbits TYPE – number of condition bits/rules (excluding nulls).
nbitsall TYPE – number of condition bits/rules (including nulls).
sumbits TYPE – total condition bit usage for type or agent.*
agentnum AGENT – agent number.
agenttag AGENT – agent number - 1 (starts at 0).
index AGENT – ordinal number of this agent in a loop.
gacount AGENT – number of GA runs.
mincount TYPE BF – forecasters are excluded if used less than *mincount*.
gainterval TYPE BF – genetic algorithm is run if due.
firstgatime TYPE BF – first genetic algorithm use.
longtime TYPE BF – unused time before generalize.
nactive AGENT BF – number of active forecasters.

ii) Real variables

p – price.
d – dividend.
d/r – dividend/interest_rate.
v – volume.
b – bids.
o – offers.
p5 – 5-period MA of price.
p20 – 20-period MA of price.
p100 – 100-period MA of price.
p500 – 500-period MA of price.
d5 – 5-period MA of dividend.
d20 – 20-period MA of dividend.
d100 – 100-period MA of dividend.
d500 – 500-period MA of dividend.
pr/d – price * interest_rate/dividend.
ppu – profit per unit (*profitperunit*).
return – return ratio (*return ratio*).
vol – volatility - 1 back.
vol5 – volatility - 5 back.
vol20 – volatility - 20 back.

vol100 – volatility - 100 back.
vol500 – volatility - 500 back.
r – interest rate (*intrate*).
eta – *eta*.
oldp – old price.
oldd – old dividend.
oldd/r – old dividend/interest_rate.
oldv – old volume.
oldb – old bids.
oldo – old offers.
avgbits TYPE – average condition bit usage per agent.*
avgtypebits TYPE – average condition bit usage in type.*
wealth AGENT – wealth.
relwealth AGENT – relative wealth.
position AGENT – stock holding.
holding AGENT – stock holding.
stockvalue AGENT – stock value.
cash AGENT – cash.
profit AGENT – profit moving average.
demand AGENT – demand.
target AGENT – target.
tauv TYPE BF – moving average estimated variance.
lambda TYPE BF – risk-aversion.
maxbid TYPE BF – maximum bid parameter.
subrange TYPE BF – random fraction of min-max range.
a_min TYPE BF – min for $p+d$ coefficient.
a_max TYPE BF – max for $p+d$ coefficient.
b_min TYPE BF – min for dividend coefficient.
b_max TYPE BF – max for dividend coefficient.
c_min TYPE BF – min for constant term.
c_max TYPE BF – max for constant term.
newrulevar TYPE BF – variance given to a new forecaster.
initvar TYPE BF – variance of overall forecast for $t<200$.
bitcost TYPE BF – penalty parameter for specificity.
maxdev TYPE BF – max deviation of a forecast in variance estimation.
bitprob TYPE BF – bits probability (p = probability).
newfrac TYPE BF – fraction of rules replaced.
pcrossover TYPE BF – probability of running crossover() at all.
plinear TYPE BF – linear combination crossover probability.
prandom TYPE BF – random from each parent crossover probability.
pmutation TYPE BF – per bit mutation probability.
plong TYPE BF – long jump probability.
pshort TYPE BF – short (neighborhood) jump probability.
nhood TYPE BF – size of neighborhood.
genfrac TYPE BF – fraction of 0/1 bits to generalize.
forecast AGENT BF – the forecast.

variance AGENT BF – variance of forecast.

iii) String variables

title – title of this program.
version – version number.
versiondate – version date.
username – current username.
hostname – current hostname.
date – date and time.
classname CLASS – name of class.
typename TYPE – name of type.
filename TYPE – type's parameter file name.
agentname AGENT – name of agent.
shortname AGENT – shortened name of agent.
selectionmethod TYPE BF – method to select between forecasts: average/best/roulette.
individual TYPE BF – whether overall variances are determined: no/yes.

iv) Special keywords

END – end of block.
MAX – maximum number of agents allowed in a loop/sum/average.
AGENTS – start of all-agents block.
CLASS – start of class block.
TYPE – start of type block.
AGENT – start of single agent block.
FOR – block prefix for loop over agents.
SUM – block prefix for sum over agents.
AVG – block prefix for average over agents.
AFTERFIRST – starting point after 1st pass.
details(string) – class/type/agent-dependent details specified by "string".*

v) Bits

on – dummy bit – always on.
off – dummy bit – always off.
random – random on or off.
dup – dividend went up this period.
dup1 – dividend went up one period ago.
dup2 – dividend went up two periods ago.
dup3 – dividend went up three periods ago.
dup4 – dividend went up four periods ago.

d5up – the 5-period MA of dividend went up.
d20up – the 20-period MA of dividend went up.
d100up – the 100-period MA of dividend went up.
d500up – the 500-period MA of dividend went up.
d>d5 – dividend > 5-period MA.
d>d20 – dividend > 20-period MA.
d>d100 – dividend > 100-period MA.
d>d500 – dividend > 500-period MA.
d5>d20 – dividend: 5-period MA > 20-period MA.
d5>d100 – dividend: 5-period MA > 100-period MA.
d5>d500 – dividend: 5-period MA > 500-period MA.
d20>d100 – dividend: 20-period MA > 100-period MA.
d20>d500 – dividend: 20-period MA > 500-period MA.
d100>d500 – dividend: 100-period MA > 500-period MA.
d/md>1/4 – dividend/mean of dividend > 1/4.
d/md>1/2 – dividend/mean of dividend > 1/2.
d/md>3/4 – dividend/mean of dividend > 3/4.
d/md>7/8 – dividend/mean of dividend > 7/8.
d/md>1 – dividend/mean of dividend > 1.
d/md>9/8 – dividend/mean of dividend > 9/8.
d/md>5/4 – dividend/mean of dividend > 5/4.
d/md>3/2 – dividend/mean of dividend > 3/2.
d/md>2 – dividend/mean of dividend > 2.
d/md>4 – dividend/mean of dividend > 4.
pr/d>1/4 – price*interest_rate/dividend > 1/4.
pr/d>1/2 – price*interest_rate/dividend > 1/2.
pr/d>3/4 – price*interest_rate/dividend > 3/4.
pr/d>7/8 – price*interest_rate/dividend > 7/8.
pr/d>1 – price*interest_rate/dividend > 1.
pr/d>9/8 – price*interest_rate/dividend > 9/8.
pr/d>5/4 – price*interest_rate/dividend > 5/4.
pr/d>3/2 – price*interest_rate/dividend > 3/2.
pr/d>2 – price*interest_rate/dividend > 2.
pr/d>4 – price*interest_rate/dividend > 4.
pup – price went up this period.
pup1 – price went up one period ago.
pup2 – price went up two periods ago.
pup3 – price went up three periods ago.
pup4 – price went up four periods ago.
p5up – the 5-period MA of price went up.
p20up – the 20-period MA of price went up.
p100up – the 100-period MA of price went up.
p500up – the 500-period MA of price went up.
p>p5 – price > 5-period MA.
p>p20 – price > 20-period MA.
p>p100 – price > 100-period MA.

p>p500 – price > 500-period MA.
p5>p20 – price: 5-period MA > 20-period MA.
p5>p100 – price: 5-period MA > 100-period MA.
p5>p500 – price: 5-period MA > 500-period MA.
p20>p100 – price: 20-period MA > 100-period MA.
p20>p500 – price: 20-period MA > 500-period MA.
p100>p500 – price: 100-period MA > 500-period MA.
allbits – all of the above bits.*

11) **output**

This is an extensive file that describes the rules for specifying output streams. Each line specifies a different output stream. However, an output stream will not be written without that stream being included in the schedule via an event in the “timelist” file described in this section.

Any number of output streams may be specified (subject to system limitations). Each one is specified with the following parameters:

streamname	
filename	(= may be used after the first)
headinginterval	
variable_or_block_1	
variable_or_block_2	
...	
AFTERFIRST	(optional)
variable_or_block_3	
variable_or_block_4	
...	
end	

The “streamname” is arbitrary, as long as different streams have different names. It is used for reference in the “timelist” file, which specifies when each output stream should be activated. The "filename" is the output filename – see the general discussion of filenames in this section under “mcontrol” above.

The "”headinginterval” is an integer specifying the frequency with which a heading (the variable names) should be added to the output file, e.g., *headinginterval = 50* would give a heading every 50 times the output stream is written. Writing 0 gives an initial heading only, -1 gives none at all, and -2 gives none and also suppresses the initial block heading

The “variable_or_block_n” parameters specify the actual variables to be output, in order. As described in detail below, each is either the name of a variable or a whole block of variables that are to written as a group for a particular agent, type, class, or set of agents. A literal “end” terminates these “variable_or_block_n” specifications. Another literal “end” must appear after the last output stream specification.

The optional AFTERFIRST keyword specifies that the variable_or_block_n specifications preceding it should only be executed the first time the stream is activated. After the first activation, only the specifications following AFTERFIRST are executed.

i) Variable specifications

Each variable_or_block_n specification can be the name of a variable or a bit, possibly with an optional prefix and/or suffix. The possible names are listed in above under “names”, or can be listed by running the program with the “–n” option on the command line. Prefix characters are:

| – do not leave a space before the variable (default is one space).
@ – average the variable's value since the last time it was printed in this output stream (or since time 0).
+ – like @, but just sum the values – don't divide by the number of samples. Useful, e.g., for getting the total trading volume.

If | is used it must come first. It may also stand alone as a separate item (followed by white space). No space may appear between @ or + and the variable name.

The only allowed suffix is a format specification, as is true for example for printf, enclosed in parentheses. A default format is used if this is not given. The format specification should be appropriate for a “double” or an “int” as appropriate (see above under “names”), except that all variables with a @ prefix are treated as double. A format may also be used alone. The variable “allbits” is a special case. It prints the value (0 or 1) of all the defined bits, in the order specified under “names”. By default these are concatenated into one long string. If a format is specified it is applied to each bit.

Some examples of variable specifications are:

p – price (double).
date – current date and time (string).
p>p20 – bit “price > 20-period MA” (*int*: 0 or 1).
|p>p20 – same, but without preceding space.
@p>p20 – average value of the bit (double).
+v – total volume since last time printed.
d/r(%.6f) – dividend/intrate with format %6f.
“t(time = %d)” – time with format string (note the double quotes).
(\n) – format standing alone.
allbits – all bits, concatenated.
allbits(%2d) – all bits, separated.

ii) Block specifications

Each variable_or_block_n specification can be a block with the following structure:

block_header
MAX n (optional)
variable_1
variable_2
...
END

The "variable_n" specifications indicate individual variables in the block, in the manner described above, but with some variations and additional cases discussed in the next section.

The block_header can be any of the following:

FOR AGENTS – loop over all agents.
SUM AGENTS – sum over all agents.
AVG AGENTS – average over all agents.
FOR CLASS c – loop over all agents in class c.
SUM CLASS c – sum over all agents in class c.
AVG CLASS c – average over all agents in class c.
FOR TYPE t – loop over all agents in type t.
SUM TYPE t – sum over all agents in type t.
AVG TYPE t – average over all agents in type t.
CLASS c – properties of class c (no loop).
TYPE t – properties of type t (no loop).
AGENT a – properties of agent a (no loop).

A class “c” may specified by a name such as “BF” (e.g., “FOR CLASS BF”) or by a number (e.g., “SUM CLASS 1”) based on the order of specification in the “agntlist” file (1 means the first class specified, etc). A type “t” may be specified by a name such as “BF2” or by a number based on the order of specification in the “agntlist” file. An agent “a” may be specified by a name such as “BF2-kib” or by its number, starting from 1. The agents are initially created in the order specified in the “agntlist” file, so, for example, “AGENT 1” means the first one specified. After an evolution event “AGENT 1” means the first remaining agent, which may or may not be a descendant of the first one originally specified. Using a named agent (e.g., “AGENT BS1-leb” or “AGENT BF2-pos-faw”) is only useful when replaying a previous run, since agent names are not otherwise known in advance.

A loop over a set of agents prints the values of the variables in the block for each agent in turn. A sum over a set of agents evaluates the variables for each agent and then prints the total for each in turn. An average over a set of agents is like a sum except that the totals are divided by the number of agents in the loop (and hence requires a double format).

The optional "MAX n" specification that can follow the block_header sets "n" as a limit for the number of agents to be included in an agent loop, sum, or average. It is only valid in a FOR, SUM, or AVG block. Note that each block must end with "END"". Here case is significant – this is not the same as the "end" used to end each stream and the whole file "output" file.

Accumulation through time (specified by a @ or + prefix on a variable name) is only allowed in a block containing a single agent, specified by "AGENT a" or by a FOR loop with "MAX 1". Normally each block starts on a new output line, and loops over agents start a new line for each agent. These new lines may be suppressed by placing a "|" before the block_header (e.g., "| FOR AGENTS").

iii) Variable specifications inside blocks

Outside a block only globally meaningful variable names can be used. Inside a block a particular class, type, or agent is defined (for each pass through a loop/sum/average if appropriate), so additional variables specific to a class, type, or agent can be used. These variables are listed above under "names" under subheadings like "agent variables". Specifying an agent implicitly specifies a type and a class, and specifying a type implicitly specifies a class, so, for example, a type-specific variable could be used in an "AGENT a" block.

Some variables change their meaning depending on the block in which they appear, as follows:

nagents – gives the total number of agents, or the number in a specified class (for CLASS c) or type (for TYPE t).
ntypes – gives the total number of types, or the number in a specified class (for CLASS c) or type (for TYPE t – always 1).
bit names – give the value (0 or 1) of a market (world) bit outside a block, but give the total number of forecasters that use that bit (i.e., not hashed [#]) for an agent. For a type (for TYPE t) the total usage in all agents in that type is given.
allbits – behaves like individual bit names. In a block specifying an agent or type, only the condition bits used by that type are given; they are given in the order specified in the type's parameter file (e.g. "bfparams"). Null bits are included, but always give 0.

Some additional special variables are only allowed inside blocks:

sumbits – gives the total bit usage (one number) for an agent or type, summed over all condition bits and over all forecasters. For a type (TYPE t) it is also summed over all agents in the type.
avgbits – like *sumbits*, except that it is normalized to give the average number of bits used per agent; normalization is done by the number of non-null condition bits and (for TYPE t) by the number of agents in the type.
avgtypebits – like *sumbits,* except that it is normalized to give the average number of bits used in the type; normalization is done by the number of non-null condition bits only.
details(string) – specifies printing of detailed statistics for an agent, type, or class. This simply sends an appropriate message to the agent, type, or class with the argument "string". The resulting output depends on the particular agent class and on "string". Documentation for these cases appears under this section in regards to the agent parameter files.

iv) Examples of blocks

a) Print the name, stock holding, and cash for the first 10 agents:

```
FOR AGENTS MAX 10 agentname holding cash END
```

b) Print the above variables plus detailed statistics for all agents in class BF, starting the details on a new line:

```
FOR CLASS BF
agentname holding cash (\n) details(stats)
END
```

c) Print the average stock holding and cash across all agents:

```
AVG AGENTS holding cash END
```

d) Print moments of the prediction coefficients across the agents in type BF1:

```
TYPE BF1 details(moments) END
```

e) Print the usage of the pr/d>1 bit for each agent in type BS1, and then the average of these values:

```
FOR TYPE BS1 agentname pr/d>1 END
AVG TYPE BS1 (average:) pr/d>1 END
```

A full example is given at the end of the demo output file.

12) param.data

This is the main parameter file for the Swarm version running in batch-mode. Like with “batch.setup”, the file begins with @begin and ends with @end. The parameters found here, incidentally, are precisely the same parameters that have default values coded in “ASMModelSwarm.m” that can be changed by the user on the GUI at the beginning of the simulation when running in GUI mode. These parameters are for all different classes. The parameters are:

numBFagents – the number of BFagents in the simulation.
initholding – the initial asset holding of each agent.
initialcash – the initial cash holding of each agent.
minholding – the minimum asset holding of each agent (short selling constraint).
mincash – the minimum cash holding of each agent (borrowing constraint).
intrate – the interest rate.
baseline – the dividend baseline.
mindividend – the minimum dividend.
maxdividend – the maximum dividend.
amplitude – the amplitude of deviations from the baseline.
period – the mean period or auto-correlation time.
exponentialMAs – whether moving averages are exponential or uniform (0 = uniform, 1 = exponential).
maxprice – the maximum price.
minprice – the minimum price.
taup – moving average healing time for profit.
sptype – the specialist type (0 = REE, 1 = SLOPE, 2 = ETA).
maxiterations – the maximum number of price iterations for iterative specialists.
minexcess – the target for |bids - offers| for iterative cases.
eta – the amount by which price changes per bid or offer for eta specialist.
etamax – maximum value of *eta* with the adaptive eta specialist.
etamin – minimum value of *eta* with the adaptive eta specialist.
rea – dividend multiplier for RE price with the re specialist.
reb – constant offset for RE price with the re specialist.
randomSeed – the random number seed for the market and specialist (0 is random).
tauv – moving average healing time for forecaster variances.
lambda – the degree of agent risk aversion.
maxbid – the maximum bid or offer.
initvar – the initial variance of forecasters.
maxdev – the max deviation of a forecast in variance estimation.

All other parameters that can be changed in the parameter files of the straight Objective-C version are currently hard-coded. However, it is relatively simple to add parameters to the above list. This will be discussed in more detail later. The file is called automatically when the instance of the ASMBatchSwarm class is created.

13) **timelist**

This file holds the schedule of "events" for the simulation. In the startup( ) function in control the instance of the Scheduler class is told to –initFromFile: timeFileName. This loads in the events. Each line has one of the following forms:

t action [params] – Event at time t.
t1 t2 action [params] – Event at time t1, t1+1, t1+2, …, t2.
t1 t2 dt action [params] – Event at time t1, t1+dt, t1+2*dt, …, t2, where "action" specifies a type of event and "params" is a parameter string that specifies further details for some types of actions.

The actions and parameters are:

d – display on screen (ignored in batch version).
dividend … – reset a dividend parameter (see **Section 2.2.1.** under "Dividend.h,.m").
D n – disable agent n.
debug – check various things if debugging flags are set.
e – evolve agent population.
enable – same as E….
E n – enable agent n.
E all – enable all agents.
level – reinitialize agents' cash, wealth, and position.
r – reset dividend shock.
reset name – reset accumulators in stream "name".
s – create a dividend shock.
specialist … – reset a specialist parameter (see **Section 2.2.1.** under "Specialist.h,.m").
w name – write an output line to output stream "name".

For example, to change to a "slope" specialist after 1000 periods you would write:

1000 specialist sptype slope

Recall that for each output stream you specify in "output", you must have a corresponding line in this file of type "w streamName".

2.3. Notes on In-Code Documentation

All part of the code includes extensive documentation on messages, parameters, and variables included. Some of this is quite technical, but any serious user of the product will probably want to understand those details anyway. The code that is not as well documented is that involving the main control files for the Swarm version (e.g., “ASMModelSwarm.h,.m”). These files, however, have been well documented here and follow quite closely the standard format for control files in Swarm applications. Additionally, the in-code documentation of the BFagent is complicated as the code there is generally hard and messy. The “demo” parameter files for the straight Objective-C version include extensive documentation that was almost entirely repeated in this documentation. On the other hand, the documentation found here is a complement to the in-code documentation (albeit with some repetition), and therefore everything there is worth reading.

3. Supported Platforms, Configuration, and Compilation

3.1. The Swarm Version

If you are able to run and compile Swarm applications, then it is straightforward to compile and run the ASM. This section does not aspire to provide a discussion on the installation of Swarm. The Swarm homepage (replete with documentation and FAQs) provides an extensive description of supported platforms and installation procedures. The homepage can be found at

http://www.santafe.edu/projects/swarm.

Here I will merely repeat the most basic points found there.

Swarm uses GCC Objective-C as its base language. Swarm is meant to run on any Unix platform. The only limitations are Unix platforms that do not support (or have poor support) for GCC Objective-C (e.g., AIX) and systems that have portability problems with libffi or ffcall. Swarm is currently known to be running on SunOS 4.1.3, Solaris, libc5- and glibc GNU-Linux systems, HPUX 9 and 11, IRIX 5.3 and 6.4, and MachTen/68k. Swarm is also running on Windows NT, 95, and 98. To run GCC on these platforms requires the Cygnus Win32 environment.

The prerequisite programs for running Swarm are as follows. You need the gcc compiler version 2.7.2 or greater, including the Objective support library, libobjc.a. The second crucial program is GNU make. It is the only make program guaranteed to work. Most Linux systems already use GNU make. The user should make sure that they are using the correct make program. Finally, you need the GNU debugger, gdb. This is not required, but highly suggested. If you are running Swarm in Windows these programs are by the Cygnus Win32 environment. The prerequisite libraries are still required.

The prerequisite libraries are as follows. The XPM Library, which adds support to X11, is needed for its colored bitmap. You need version 3.4f or later. You also need Tcl/Tk version 8.0p2/8.0p2 or later. Next Swarm requires BLT, an extension of Tcl/Tk. Swarm requires version 2.4g or later (or BLT 8.0-unoff for Windows). You need *either* ffcall version 1.5 or later or libffi version 1.20 or later. Finally you need libpng version 1.0.2 or later and zlib version 1.1.3 or later. For the availability of these programs and libraries see the Swarm homepage.

After these are installed, you need to obtain Swarm from the Swarm homepage. You will unpack it and then run a GNU configure script. After this is accomplished Swarm is compiled using “make” at the top of the source directory and then “make install”. After Swarm is installed you can run any number of Swarm applications. What follows describes how to compile and run the ASM.

There is only really one step needed to configure the ASM. In the “Makefile” you need to set the SWARMHOME directory to be its proper location. The default setting is

SWARMHOME = /..swarm-<version_number>.

This will work if the ASM application and Swarm are installed as sub-directories of the same directory. This needs to be edited (if necessary) to be:

SWARMHOME = <path in which Swarm is installed>.

A typical place Swarm might be installed is:

SWARMHOME = /usr/local/swarm-<version_number>.

Once this is properly set up, compilation is easy. You need only to run make in the ASM directory (where the Makefile is located). The Makefile compiles all the modules (creating the appropriate intermediate .o files) and creates the executable “asm”.

The executable can be run in batch-mode or in GUI mode. To run the program in GUI model, simply type “asm”. To run it in batch-mode type “asm –batchmode”. The latter will load the parameters specified in “batch.setup” and “param.data”. The former will load the program and its graphical widgets. Use is described in **Section 4**.

3.2. The Straight Objective-C Version

The only required software needed to run this program is GCC and GNU make. As such, if you have Swarm installed and working you should be able to compile and run the straight Objective-C version without any problems. This also implies that any platform on which you can run Swarm you can also run the straight Objective-C version (Unix and Windows). It should also be the case that platforms with features limiting or prohibiting the use of Swarm will also limit or prohibit the use of this version. The exceptions (per our discussion of the Swarm version above) are those platforms that run GCC but do not support libffi or ffcall. It is unclear which platforms these might be.

A note should be made about running the straight Objective-C version on Windows machines. As was mentioned above, the Swarm version will run on Windows with Cygnus Win32 installed. There is another environment for Windows that supports GCC: DJGPP. Currently Swarm is set up to work with Cygnus Win32 and not DJGPP. However, the straight Objective-C version is configured to run with DJGPP on the Windows platform and not Cygnus Win32. If you want to run both versions in the Windows environment your best hope is to adapt the “config.h” file in the straight Objective-C version to allow it to work with Cygnus Win32. Recall that “config.h” provides a number of preprocessor definitions that it allow it to work with different platforms. Currently the platforms included therein are:

Solaris,
Linux,
SunOS-4,
DJGPP for MSDOS, Windows9x, WindowsNT (tested only on NT), and
NextStep.

See the “config.h” file for the form of an entry in that file. An example (from that file) follows.

```
// For Linux
#if defined(__linux__)
#define ERROR_HANDLER_2
#define GETCLASS   objc_get_class
#define RUNTIME_DEFS     <objc/objc-api.h>
#define UNIXLIB          <unistd.h>
#define PATHSEPCHAR     '/'
```

If you are adding a configuration, you can use the command "gcc -v -E config.h" to see what special variables the compiler defines.

Now let us consider how to build the executable. In Unix you need to follow several steps.

1. First cd to the “src” directory. Under the main market directory (“sfsm”) there are two directories: “demo” and “src”. The directory “src” contains the source code. The directory “demo” contains the demo parameter files.
2. Make sure that “newversion” is executable – run “chmod +x newversion” if necessary (or anyway if in doubt).
3. Run “make”. This should ultimately create “market”.
4. [Optional] Run “strip market” to make “market” smaller. On modern powerful computers this should not be necessary.

Every time you run “make”, the “newversion” script is run (and “version.m” is re-compiled) to increment the version number. If you encounter any problems with this mechanism you can disable it in the “makefile” by deleting the relevant lines (specified therein).

For MSDOS/Windows platforms with DJGPP you probably will not install full support for makefiles and shell scripts. So it is easier to do things by hand.

1. Make sure DJGPP is installed properly, including setting the PATH and DJGPP environment variables as in the DJGPP installation instructions.
2. In a command shell (DOS-like) window, cd to the “src” directory.
3. Run the command:

   ```
   gcc –o market.exe –Wall *.m –lobjc –lm
   ```

   This should ultimately build the executable “market.exe”.

On all platforms running the program “market” (or “market.exe”) requires the control parameter file, “mcontrol” by default (as discussed in depth above). It normally looks for the “mcontrol” file in the current working directory and then in the directory where the executable resides. It is suggested that you keep the parameter files in a separate directory and run the executable from there as will be described shortly. You can specify an alternate filename on the command line, e.g.,

```
market otherControlFile.
```

In most cases you should run the program in a different directory than the “src” directory (even though the executable will reside in “src”). The provided “demo” directory is an example. That means you need to do one of several things to make “market” able to run from your working directory:

1. Move “market” to your working directory; or
2. Put a link, alias, or shortcut to “market” into your working directory (these are different OS's names for more or less the same thing); or
3. Put “market” into a directory included on your PATH, or add the appropriate directory to your PATH; or
4. Use a specific pathname, e.g. “../src/market” to run it.

Under Unix you should find that there is already a symbolic link to “../src/market” in the demo directory, but if you create other working directories you may need to make your own, for example, with a command like

```
ln -s ../src/market.
```

There are various options (command line flags) available, e.g., for debugging. As was mentioned above under “mcontrol”, these are usually specified in that file but can also be specified on the command line. You can type “market –h” for a list of command line options. Alternatively see the documentation on “mcontrol” above.

To start using your own parameter files (other than those provided in “demo”), you can edit the files found in “demo”. Alternatively, you may want to keep “demo” unchanged, make a new working directory, and modify copies of the demo files in that.

In that case you must have all the necessary parameter files there and you must run "market" from that directory.

Under the default setup (in "mcontrol") each run will produce a log file, with a name like "log123.smt" (not included with "demo"). You can rerun a previous run with a command like

market log123.smt.

Recall that the log file concatenates all parameters from each of the parameter files so that a command like the above will just read "mcontrol" and then the log file specified on the command line.

4. Demos

4.1. The Straight Objective-C Demo Directory

The documentation above on parameter files includes discussions of the parameter files included in the directory "demo" of the straight Objective-C release. All these files include documentation that was mostly repeated in these pages. As was mentioned above, the parameter files do not need to have the same names as those in the "demo" directory, but there is no reason to change them (other than adding parameter files for agents). To run the demo you need only to compile the program with "make" and then run the program from the "demo" directory. To do this you must either fully specify the path of the "market" executable or include the executable in your path. This was discussed in depth in **Section 3.2.** on supported platforms, configuration, and compilation. The demo has the following settings:

1) No debugging flags are set in the "mcontrol" file (indicated by a value of 0), but the user can specify her own on the control line with the –d flag (e.g., "market –da" to debug the agents).
2) The market/agent random number seed is set in "mcontrol" to be random (indicated by a value of 0).
3) The file "mcontrol" sets the other parameter files to "dparams" for the dividend process, "mparams" for the specialist and market, "agntlist" for the agents, "output" for the output specifications, "timelist" for the event schedule, and "log###" (indicating a file called "log<runid>") for the log file.
4) The "agntlist" file has one entry indicating that there are 30 BFagents with an initial stock holding of 1 and parameters specified in the file "bfparams".
5) The file "bfparams" has many parameters (described in depth in the section on parameter files), so we will not discuss the settings for each.
6) The "demo" directory includes parameter files for all other agents even though there are none of these types in the demo. The files are: "bsparams" for the BSagent, "ffparams" for the FFagent, and "duparams" for the DUagent. There are descriptions of these files in **Section 2.2.2.3.**.

7) The file "mparams" gives some typical values for the market and specialist parameters. It sets the specialist to be of type FIXED (indicating the fixed eta specialist).
8) The "output" file establishes two output streams. The first is called "prices" and is written to stdout (i.e., to filename "-") in the demo. The second is called "agents" and is written to the file "agents###" (i.e., agents<runid>).

   For the "prices" stream the heading interval is set to "0" so that the heading is only written the first time the output goes to the screen. The "prices" stream includes the following variables: the market seed (*mseed*) and dividend seed (*dseed*) (which are written only AFTERFIRST, that is after the first time the output is written), the time (*t*), price (*p*), the dividend divided by the interest rate (*d/r*), and the average volume (*@v*, where @ means average in the output file).

   The "agents" stream's heading interval is also set to "0". The variables included in the stream are the time, the average price (*@p*), and a number of agent specific variables. The agent specific entries are:

   (april) CLASS BF classname classnum nclasses ntypes details(gainterval) END
   (may) CLASS 1 classname classnum nclasses ntypes END
   (rick) TYPE BF1 classname classnum pshort nclasses ntypes nrules details(gainterval) END
   (jane) AGENTS BF1-ziy classname classnum pshort nclasses ntypes nrules gacount END
   (mike) AGENTS 2 agentname cash forecast details(gainterval) END
   (rob) FOR AGENTS agentname cash END
   (kate) FOR AGENTS MAX 5 agentname cash pshort END
   (carol) FOR AGENTS MAX 1 agentname +cash pshort END

   The names in parentheses are identifiers. The first, second, and third entries write the same information about the only class and type of agent specified in the "agntlist" file. Typically CLASS BF refers to all agents of class BF, class 1 refers to all agents of the class of the first specified agent class, and BF1 refers to the first type of agent of class BF. Since there is only one entry in the "agntlist" file (a BFagent with an initial holding of 1 and parameter file "bfparams") all these line write the same information. The fourth line writes information about the agent named –ziy (though this is only useful if re-running the simulation since agent names are not known in advance). The fifth entry is about the second *agent* created. The last three entries aggregate information over the agents. See the section (**Section 2.2.2.3.**) on parameter files to get more detail on these entries.
9) Finally, the "timelist" file schedules events. It schedules for the specialist to change from the type specified in "mparams" (the fixed ETA) to the slope specialist at time 1000. It then schedules that the "prices" stream (specified in the "output" file) be written every 100 periods between period 0 and 5000 (given by "0 5000 100 w prices"). Next it schedules any specified debugging

to occur every 1000 periods between time 0 and 5000. This happens only if debugging flags are set. The file “mcontrol” sets none, so they would have to be specified by the user on the control line. Finally, the “agents” stream is scheduled to be written at time 5000. This file implies that the simulation will run until time 5000 when the last event is scheduled.

When you run the demo, then, two files will be immediately created: “log<runid>” and “agents<runid>”. The log information will immediately be written to “log<runid>”. Every 100 periods the variables mentioned above will be outputted to your screen and at time 5000 the agent information will be written to “agent<runid>” and the simulation will stop. You can replicate the demo exactly (that is using the same random number seeds) by typing “market log<last runid>”.

4.2. Using the Swarm GUI

To run Swarm you must go to the directory where all the source code resides, compile the program using “make”, and run it using “asm”. When you do so, three widgets initally appear. The first is the “ProcCtrl” widget. It has five buttons: “Start”, “Stop”, “Next”, “Save”, and “Quit”. “Start” commences the program. “Stop” stops it but does not terminate it. “Next” advances the program one time step (i.e., calls the higher level Swarm message –go that advances the loop). “Next” will work either when the program is running or is stopped. “Save” creates the file “.swarmArchiver” in the main Swarm directory. This file….. “Quit” terminates the program. To activate any of these buttons the user must **left click** on the button of choice.

The second widget, called “ASMModelSwarm” because it is the ProbeDisplay owned by the ASMModelSwarm, includes all of the parameters that the user can change at the start of the simulation. These parameters are listed in **Section 2.2.2.3.** under “param.data”. Recall that “param.data” is the file read by the ASMBatchSwarm when the program runs in batch mode. The list of parameters there corresponds to the list of parameters that can be changed on the GUI. The notable default values are 50 BFagents and the Slope specialist (i.e., *sptype = 1*). The default random number seed (*randomSeed*) is 0, which indicates that the seed will be a random number. The moment the program is started (by clicking on the “Start” button) the result of the randomization is displayed. This is because it is the actual seed that is relevant from the perspective of reproduction. If the user clicks on the “writeSimulationParams” button on the third widget (to be described), they are written at the end of the simulation and so reflect the result of any randomization of the seed.

Before the program is told to “Start”, any of the parameters can be changed. To change a parameter **left click** on the are that holds the parameter, delete the old parameter and type in the new value, and then **press Enter** on your keyboard. Generally in a Swarm application, parameters can be changed after “Start” when the user stops the program by clicking on the “Stop” button. The parameters are changed in the usual way. This can be done with the ASM as well. However, and this is a big however, once the program is started, all of the initial parameter values are distributed to the objects that will actually use them. Since changing a parameter only changes them in the

ASMModelSwarm, and the ASMModelSwarm only never uses parameters after it gives them to the objects it creates, changing the parameters will not affect your simulation. It is easy enough to change the code so that this is not the case. An interested user is encouraged to do so.

The third widget is the "ASMObserverSwarm" widget (i.e., the ProbeDisplay that the ASMObserverSwarm creates). It includes one parameter, the *displayFrequency*, which as described in **Section 2.2.2.2.** and **2.2.2.3.** determines the number of time steps before the display widgets are updated. The default setting is 3 and can be changed in the same way as the parameters on the "ASMModelSwarm" widget. Also like with those parameters, the *displayFrequency* can be changed after "Stop" is clicked, but it will not affect the simulation. Finally, there are two other buttons on the "ASMObserverSwarm" widget. The first is "writeSimulationParams" and the second is "writeSimulationData". As was mentioned in **Section 2.2.2.2.**, if the user wants to write the data for a particular run, "writeSimulationData" must be selected before the start of the simulation. The data is written to the file "output.data<*runTime*>" (specified in "Output.h,.m"). However, "writeSimulationParams" can be selected at any time during the simulation. In this way the user can identify by the graphical output whether a run is worth replicating before having the simulation write the parameters. The parameters are written to the file "param.data<*runTime*>" at the end of the simulation. This file has exactly the correct format to be loaded using the ASM in batch mode. To replicate a run exactly, the user only needs to edit the filename to be simply "param.data", i.e., the user only needs to remove "<*runTime*>" from the appropriate parameter file.

The "ASMModelSwarm" and "ASMObserverSwarm" widgets have another feature in common that is available in all Swarm applications (and in turn is described in much more detail in the Swarm user documentation). On these widgets, there is a button with the name of the widget in blue writing. The user can **right click** on this button and another widget will appear. This widget has the same name as the initial widget (here either "ASMModelSwarm" or "ASMObserverSwarm") but represents the entire class that created it, not just a ProbeDisplay within that class. As such, the widget includes all of the messages and parameters for that class. So for the "ASMModelSwarm", the new widget not only includes the parameters that were mentioned before, but variables that the ASMModelSwarm class owns like *agentList* or *specialist*. These, like the parameters, also have values (e.g., List_linked for *agentList* and Specialist for *specialist* after the simulation has been started). These cannot be changed at anytime. However, the user can **left click** on the message buttons to activate them. Generally these correspond to messages that will not affect the simulation. In that case they just return some value. For example, for the "ASMModelSwarm" class widget, clicking on the "getNumBFagents" button will display the number of BFagents. Clicking on the "getSpecialist" button returns Specialist. These do not affect the simulation. However, there are buttons like "createEnd" that will affect the simulation, in this case terminating it. Worse yet, clicking on buttons like "buildObjetcs" actually re-creates all the objects but without any data or actions so that the simulation continues to run but without doing anything. For almost all users these widgets are of no use.

Even so, these class widgets have yet another feature. In the upper right hand corner there are two more buttons, one red and one green. **Left clicking** on the green button displays the variables and messages for the superclass of which the original class is a subclass. So for the "ASMModelSwarm" widget, clicking on the green button brings up the messages and variables for the "Swarm" class (and for the "ASMObserverSwarm" the "GUISwarm"). Clicking yet again on the green button gives the messages and variables for the "CswarmProcess", and so forth. These are of limited use for the vast majority of users. Clicking on the red button reduces the class by one level. So if the "Swarm" class information is displayed, clicking the red button will reduce the widget to just the "ASMModelSwarm". At that point clicking on the red button closes the widget. The same is true for the "ASMModelSwarm" ProbeDisplay where the user can change the simulation parameters. On that widget, though, there is no green button. For more information on these features see the Swarm documentation.

Of more relevance are the display widgets that appear when the user left clicks on the "Start" button. These were discussed in **Section 2.2.2.2.** under "ASMObserverSwarm.h,.m". They include "Price v. time", "Volume v. time", "Agent Position", and "Relative Wealth of Agents". The first two are line graphs and the last two histograms. The histograms have no additional features. They update at the rate *displayFrequency* and are indexed by agent id (i.e., display a bar for each agent). The line graphs do have an additional feature. The user can zoom in on different parts of the graph. To do this, **left click and hold the left button** on the portion of the graph that you wish to be the upper left corner of the area you wish to zoom in on. The words "Zoom #1" will appear. Then **drag** the mouse **down and to the right** to create a box around the entire area of interest. **Release the button** and then **click the left button again**. The graph will now display the highlighted area without "Zoom #1". The user can continue to zoom in as many times as she wishes by following the same process. Each time the user zooms the words "Zoom #*n*" will appear, where *n* reflects the total number of zooms the user has done. To step back one zoom (i.e., zoom out), simply **right click** anywhere on the graph. The words "Zoom #*n-1*" will appear until the graph returns to its default. Line graphs also update at the rate *displayFrequency*.

Finally, to run the simulation in batch mode, simply type "asm –batchmode". The same default parameters will be loaded from the file "param.data". To change parameters simply edit the "param.data" file.

5. Notes on Known Bugs

What follows is a list of known bugs. Several will not apply to the Swarm version. This is noted where appropriate.

1. Agents other than the BFagent have not yet been updated to allow multiple calls to –getDemandAndSlope in a given period. They misbehave badly and should not be used for serious research. This applies to the straight Objective-C version and is only a testament to the fact the BFagent should be the user's agent of choice.
2. The specialist may unnecessarily call –getDemandAndSlope twice for each agent?
3. The timing system (as described under the section on time) always has *oldprice=price* when the World is updated (i.e., when –updateWorld is called in the

Swarm version and –makeBitVector is called in the straight Objective-C version). In turn the price up/down bits are always off.

4. The –printAgentInfo message in AgentManager has been specialized to print information only for the first agent type (*type=0*), assuming that that type has condition bits. This only applies to the straight Objective-C version.
5. In the Dividend process, the parameter *rho* (which is always used in the calculation of the new dividend) is rounded. It could be that *rho* is rounded such that *rho*=1. This is true for both the Swarm and straight Objective-C versions.

These are known bugs, the operative word being known. Users should add to this list if necessary.

6. Bibliography

Arthur, W.B. 1992. "On Learning and Adaptation in the Economy." *SFI Working Paper* 92-07-038.

Arthur, W.B. 1994. "Inductive Behavior and Bounded Rationality." *American Economic Review* 84: 406-411.

Arthur, W.B. 1995. "Complexity in Economic and Financial Markets." *Complexity* 1: 20-25.

Arthur, W.B., J.H. Holland, B. LeBaron, R.G. Palmer, and P.J. Tayler. 1997. "Asset Pricing under Endogenous Expectations in an Artificial Stock Market." In W.B. Arthur, S. Durlauf, and D. Lane, eds., *The Economy as an Evolving Complex System II*. Redwood City: Addison-Wesley. Available electronically as SFI Working Paper 96-12-093 from http://www.santafe.edu/publications.

Arthur, W.B., J.H. Holland, B. LeBaron, R.G. Palmer, and P.J. Tayler. 1997. "Asset Pricing under Endogenous Expectations in an Artificial Stock Market." *Economic Notes* 26: 297-330.

Joshi S., and M. Bedau. 1999. "Four Kinds of Behavior in an Artificial Evolving Economic Market." *Mimeo*.

B. LeBaron, W.B. Arthur, and R.G. Palmer. 1998. "Time Series Properties of an Artificial Stock Market." *Journal of Economic Dynamics and Control* xx, xxxx.

Palmer, R.G., W.B. Arthur, J.H. Holland, B. LeBaron, and P.J. Tayler. 1994. "Artificial Economic Life: a Simple Model of a Stock Market." *Physica D* 75: 264-274.

R.G. Palmer, W.B. Arthur, J.H. Holland, and B. LeBaron. 1998. "An Artificial Stock Market." *Artificial Life and Robotics* 3: xxxx.