"What I learned from the Artificial Stock Market"

Paul E. Johnson
pauljohn@ku.edu
Dept. of Political Science
University of Kansas
Lawrence, Kansas 66045

November 5, 2001

Abstract

The Santa Fe Artificial Stock Market is a well known agent-based model. A new revision of the code (ASM-2.2) is now available. This essay describes some of the changes that were made in the code base and also presents some important lessons for agent-based modelers that can be illustrated with the code. Since the code is available on the internet, it is hoped that this discussion and the code base it represents will be helpful to people who are planning projects in the field of agent-based modeling.

Acknowledgements: I would like to thank the ASM research team for making this code available. I would especially like to thank Blake LeBaron, who has answered quite a few questions about the details of this big set of code, and Brandon Weber, who did the detail work to release ASM-2.0. All errors are my responsibility alone.

There is plenty of talk these days about "agent-based modeling" in social science, but precious little of it helps with the actual details of designing such a model and writing code for it. This essay explores some conclusions that were reached while working on a revision of the code for the Santa Fe Artificial Stock Market (ASM) model that was made famous in a very widely cited article by R.G. Palmer, W. Brian Arthur, John H. Holland, Blake LeBaron, and Paul Taylor (1994). The ASM was one of the first projects to demonstrate the potential of agent-based models for the exploration of theories of decentralized, individualistic, and boundedly rational behavior. It has sparked a small cottage industry of research on artificial stock markets (see LeBaron, 2000).

This essay explores practical issues that any prospective model-builder will confront. These issues are of general importance, not just to people who want to write about stock markets or use a particular programming library. The overall message is that models should be designed as separate, independently functional pieces of code (objects), and that some guidelines can be followed to improve readability and reduce unintended consequences (bugs).

I don't write from the perspective of a computer scientist who arrives on the scene to tell the social scientist what she ought to do. Quite the contrary. I am a social scientist by training. When I started learning about the Swarm Simulation System (Minar, Burkhart, Langton, Askenazi, 1996) in 1997, I did not know a great deal about object-oriented programming or agent-based modeling. For me, there was a hand-in-hand development of computing skills along with a comprehension of agent-based modeling. I suspect it will be the same for many social scientists who are drawn to the general appeal and claims about agent-based modeling.

Background on the Stock Market Code Base

The Swarm Development Group (http://www.swarm.org) makes available a set of working

code examples for the Swarm Simulation System. In addition to those supported models, there are many models contributed by the user community. These contributed models were historically kept in an ftp directory under the label "anarchy." Users who would venture into the anarchy would find some simple, interesting models, and some very complicated, hard-to-understand models.

The ASM code base is one of the difficult ones. The original model was written in the language Objective-C for the Next operating system. The passing of the Next operating system made a casualty of the original stock market model. There is, however, a package of code available under the name "sfsm," which is as close to the original code as we are likely to get. The sfsm code is the essence of the original model, except that the graphical interface which relied on the Next libraries has been removed. With only slight modification to work-around some conflicts due to changes in the GNU C library, the sfsm code can be compiled with a current version of gcc (the Gnu Compiler Collection; http://www.gnu.org/directory/gcc.html).

A glance through the sfsm code reveals that, although it is written in Objective-C, it is in fact ordinary C that is doing most of the work. (Objective-C is a superset of ordinary C, so anything that can be done in a C program can be housed inside an Objective-C program as well.) The importance of the distinction is in the extent to which an **object-oriented** framework is used. The sfsm is object-oriented to the extent that the significant actors in the model are housed in separate files and when the simulation runs, objects built from those classes are instantiated. The design on the macro level is excellent, separating the functionality of the components into clearly demarcated containers. One finds files to describe the behavior of stock investors (most importantly, the bitstring forecasting agent: bfagent), the dividend generator (dividend), a specialist who manages the market (speclist), a world object that collects data and makes it available to agents when asked (world), and

various other classes that orchestrate the interaction of these classes. However, inside these largish files, the design is decidedly more like C than Objective-C. Each bitstring forecasting agent maintains a population of forecasts and tries to perfect them over time, but each forecast is not kept as a separate object. Rather, there are plenty of instances of dynamic memory allocation and pointer usage.

Various members of the original ASM team, along with staff members and an intern at the Santa Fe Institute, set to the task of revising the sfsm model to run with the Swarm libraries. There are a number of benefits to doing this. First, Swarm provides a set of well-tested routines for things as mundane as random number generation and scheduling of agent actions. Second, Swarm has a built-in graphical interface, including graphing capabilities. Since the graphical interface was lost with the demise of the Next system, this was an important consideration. In order to put the sfsm code into a Swarm framework, a significant amount of work was required to create the standard multi-level Swarm simulation in which a top level swarm object observes events and creates graphs, while a middle level model Swarm manages the actual creation and interaction of the substantively important actors in the model.

During 1998 and 1999, a preliminary version of the Swarm ASM was available in the "anarchy" ftp directory. Since I was a relative neophyte in the world of Swarm, I looked on the code in a state of wonder and amazement. I could not imagine how all of this detail could possibly fit together into a coherent pattern. I was constantly asking people in the Swarm community about the status of that code because I wanted to see a big, social science model that used Swarm. Many of the Swarm code examples are about bugs and other ecological critters. There were not as many examples that dealt with social science research. I thought the ASM could serve an important educational role.

In April, 2000, the Santa Fe Institute announced the release of ASM-2.0. ASM-2.0 package includes three elements . First, there is a copy of the Objective-C version called

sfsm, which appears in a directory called ASM_OBJC. Second, there is the Swarm version, which appears in the directory called ASM-2.0. Third, there is a Microsoft Word document by Brandon Weber, who was the the person most closely in contact with the actual coding of the revision of the artificial stock market. The separate elements of that release are available (ftp://ftp.santafe.edu/pub/swarm/src/users-contrib/anarchy/ASM/).

During the semester of the ASM-2.0 release, I was teaching a graduate seminar on agent-based modeling. I was eager to refer my students, many of whom were in finance and economics, to the ASM code. By that time, my coding skills had improved quite a bit (I was an author of the *Swarm User Guide* (Johnson and Lancaster, 2000) and had written several working example models). After reviewing the ASM-2.0 release, I was able to see that some changes could be made to make this code easier to read and modify. Some parts of the code were needlessly complicated and there were many details that were difficult to explain to students.

Perhaps not the least of my concerns was that the code, as distributed, had some kind of mistake in it that caused the dynamic behavior of the model to deviate from the expected pattern, the pattern one would expect from the description of the model in Palmer, et al., or from runs of the sfsm model. In Figure 1, there is a "screenshot" of a run of the ASM-2.0 code. Note that in the bottom panel, the market price goes low and says far below the risk neutral price. One would expect the price during the simulation to more closely mirror the risk neutral price.

I decided to start an open initiative to revise and update the ASM code base, not only to find the source of that problem, but also to enhance the code as a teaching example and as an illustration of the possibilities of the Swarm toolkit. There is an absolutely great resource for open source code development known as Sourceforge. Sponsored by VA Linux, Sourceforge offers web pages and code repositories (using CVS, the concurrent version

system) to people who want to develop code that is open to inspection and free usage. The Sourceforge administrators granted my application for the ArtStkMkt project and a page called http://ArtStkMkt.sourceforge.net. People who want to read about the status of the revisions or checkout code from the CVS archive can visit that site.

After working on the ASM code for slightly more than one year, I have announced the release of ASM-2.2, a significant revision/enhancement/clarification of the ASM-2.0 code base. This release signals the introduction of several new classes, the simplification of the usage of Swarm's scheduling facilities, and a radical simplification and redesign of the code that governs the way the bitstring forecasting agents behave. Midway through the revisions, the "bug" illustrated in Figure 1 was discovered and exterminated. In Figure 2, one can find a screenshot of the corrected model that was run under the same conditions.

In this essay, the intention is not to give a comprehensive review of all the changes that occurred in the ASM code. Rather, using the ASM revision as an example, there are some general points to be made about the model-writing experience. People who are interested in the details of the revision are free to check out the code from the CVS archive and use the facilities of the cvs program to track through the exact history of the revisions. On the ArtStkMkt web site, there is an html version of the documentation for the new version, which includes an overview of the model and detailed documentation on all of the classes. The code includes a Changelog and the README file inside the distribution contains periodic updates. One can also look at the periodic code patches that have been issued which include the exact changes needed to move from one edition to another.

Lessons to take away from the ASM experience

The comments here are organized in order of decreasing generality. The intention is to speak to issues that will affect all kinds of agent-based modeling projects, not just models of stock markets. Having the ASM code handy for reference, however, gives us something

concrete to talk about.

The overall message is that **object-orientation is good**. Where possible, models should be designed so that they are composed of separate, isolated pieces that fit together in understandable ways. This makes it easier to extend models, to find bugs, and to communicate with one another. In several of the specifics to be explained below, it is also argued that the isolation and confinement of calculations within particular objects, functions, or methods is important. Global variables are bad and should be avoided.

Hint #1. Replace structs with objects.

A C struct is a group of variables. An object is also a group of variables, called instance variables, but it has an additional capability. An object is created as an instance of a class, and because it is derived from a class, then an object can execute methods. (The term *method* is the object-oriented terminology for a *function*.) In addition to storing values inside an object, one can also tell an object to make calculations. This is a way of "hiding" a lot of work and getting it done in a dependable way inside separate objects.

In the ASM-2.0 code, the buying and selling of stocks is done by the BFagent class (BF stands for bitstring forecasting). The header file BFagent.h declares two large structs, *BF_fcast* and *BFparams*. In ASM-2.2, several classes were created to contain the information and these structs were removed. The *struct BF_fast* was replaced with the object *BFCast*. The *BFCast* object is quite a complicated thing because it does the work of taking certain pieces of the world into account and then formulating a new forecast of the stock's price. That work was so complicated, in fact, that another smaller class, called *BitVector*, was created to handle the lowest level details of keeping records on the bits of information. Similarly, the struct *BFparams* was replaced with a class BFParams.

As an example of the simplification that comes when objects are introduced, consider the code in ASM-2.0's file BFagent.m. When a forecast (in this case, a pointer to a struct

named *fptr*) is to be updated, the value is explicitly calculated by retrieving some values from the parameters struct (*pp*) and *fptr* itself:

```
fptr->specfactor = 1.0/(1.0 + pp->bitcost*fptr->specificity);
```

All of this calculation can be done inside the object itself by a method calle *updateSpecfactor*. Then, whenever one wants that calculation to take place, one tells the forecast to do the calculation. In Objective-C, that looks like:

```
[aForecast updateSpecfactor];
```

If data must be updated in several different places, this method can be called in each place. Any changes needed in the updating procedure can be taken care of inside the forecast object's *updateSpecfactor* method, rather than in several parts of *BFagent*.

Perhaps the most important example of introducing classes to isolate calculations is the *BitVector* class. The bitstring forecast objects (instances of *BFCast*) monitor up to 80 dichotomous variables in the stock market world. These bits in the world indicate that the price is rising or declining, or that the moving average of dividends is rising or declining, and so forth. Each forecast can monitor some or all of these bits, and so each forecast has to maintain a record of which bits it is monitoring. This is done in ASM-2.0 with explicit bit math in the file BFagent.m. (A "bit" is a 0 or 1 value. Every integer has a representation in binary (base-2) format, a string of 0's and 1's. Hence, each integer can compactly store a lot of bits.) That makes the code extremely difficult to read, even if one enjoys bit operations (and I don't).   In ASM-2.2, bit math has been confined to the *BitVector* class, and each of the forecast objects is created with a *BitVector* instance inside it. Changes to the *BitVector* values are done through an interface with meaningful (comparatively speaking, at least) method names.   The BFagents never need to precisely modify the individual bits because all of that detail is hidden inside the implementation of the *BFCast* and *BitVector* classes.

Hint #2. Keep the scope of all variables as small as possible.

This is a standard piece of advice to students of programming (Kernighan and Pike, 1999, p. 104), but it is easy to miss the importance of it. Scope is the area in which a variable is visible. A global variable is visible throughout a program, an instance variable is visible within an object, and a local variable is visible within a method only. When an object is small, there is some convenience in keeping a single instance variable, the value of which can be easily accessed by several different methods, either for reading or writing. The problem is that, as the class grows, then one may make mistakes that accidentally change the value of that variable. Programs that use variables with large scope are derisively called "spaghetti code" because a "yank" on one strand can have unexpected implications throughout.

Instead of using a single variable that is accessed in several methods, it is often better to revise the methods so that they take explicit arguments and explicitly return values, without relying on easy access to variables from a higher scope. In the ASM-2.2, for example, there are significant changes in the methods that control the way the agents conduct the genetic algorithm that optimizes their forecasts. For instance, the function declared in ASM-2.0 like so:

```
static void Generalize(struct BF_fcast *fcast)
```

assumes it has access to a global variable called *avstrength.* To prevent the possibility that the value of avstrength might be accidentally altered somewhere, the Generalize method of ASM-2.2 now takes as one of its input arguments the value of avstrength:

```
-(void) Generalize: list AvgStrength: (double) avgstrength
```

Using this approach, there is no longer a need for a global variable avstrength.

One of the risky practices that is fairly widespread throughout the ASM code is the use of static variables. A static variable in Objective-C is a variable that is shared by all instances of a class. It is a useful approach if there really is a single value that is supposed to be available for all instances of a class. However, sometimes static variables are used for

convenience, say for "workspace" calculations. The usage of static variables is inspired by the desire to save memory, since room for only a single piece of memory is needed. If there are thousands of instances of a class, the savings of memory can be significant, and even on a new computer for which memory is cheap, the savings might be worthwhile. But this is not without danger. Any instance of the class can change the value of a static variable, and in a complicated simulation, one can create very hard-to-find bugs by using static variables.

Many of the changes that have been made in ASM-2.2 were not responses to problems, but rather steps to prevent future accidents or to resolve student questions before they arise. For example, in the ASMObserverSwarm.h file, there was an instance variable:

```
double *position;
```

This points to a location in memory which is allocated in ASMObserverSwarm.m. The initialization created a permanent, dynamically allocated piece of memory like so:

```
position = xcalloc (numagents, sizeof (double));
```

That piece of memory is used during each iteration in the *updateHistos* method, where the holdings of the agents are collected into that array and then passed along to a charting object that the authors refer to as a histogram (but it rather appears to be a bar chart).

In ASM-2.2, the instance variable *position* is eliminated, as is the need to dynamically allocate memory. Instead, it is possible to rewrite the *updateHistos* method with a local array *positions* which is created to hold information from just as many agents as are currently in the model's list of agents. First the number of agents is retrieved, then the *position* array is created.

```
int numagents = [[asmModelSwarm getAgentList] getCount];
double position[numagents];
```

The advantage of this approach is not just that it keeps the scope to the absolute minimum and avoids the use of dynamically allocated memory, but it also means that the chart object can be

used to chart the fortunes of an agent list which changes in size during the simulation. In ASM-2.0, there is no flexibility to accommodate change in the number of agents.

Hint #3. Use a collections library.

A collection is a container, a "thing" that can have objects added to it and removed from it. The Java language is distributed with a large library of collections, ranging from flat (unordered) lists to sorted sets. The C, C++, and Objective-C languages are not distributed with collections libraries, but there are collections libraries that can be used with these languages. A good open source collections library for C can be found in the glib package (http://www.gtk.org). For C++, there is the Standard Template Library and for Objective-C models the Swarm toolkit has a collections library (or one could also use the recently released GNUstep toolkit: http://www.GNUstep.org).

If one does not want to use a collections library, then what? A look at the ASM-2.0 code will give an example of the alternative: use dynamically allocated memory and pointer math. (Readers who do not now about memory allocation in C might consult one of the standard texts, such as (Kernighan and Ritchie, 1988, or Kochan, 1994)). In Kochan (1994, p. 255), there is a presentation describing how one can create a linked list in the style that is used in ASM-2.0. A C struct is used to contain the information about a "thing," and one piece of information in the struct is a pointer to the next "thing" in the collection. As long as one has access to one struct, then one can always find the next one.

The explicit usage of structs and memory pointers is not so much wrong as it is prone to error. To correctly make a series of calculations, one for each struct, a sequence of relatively detailed steps must be taken. In C, there is no warning if one grabs an incorrect segment of memory while conducting such an exercise.

Using a container library, the maintenance and traversal of collections is converted into a simpler, more reliable process. If one has an object, inserting it into the collection

usually requires a simple command like "add" or "put." All of the collections libraries that I have seen include a standard set of tools for stepping through the collection with an iterator. In Swarm, the interator for a collection is often called an "index," and one simply asks a container to create an index with a *begin* statement and then each successive element of the collection is found by asking the index for the "next" element.

In Table 1, one can find a comparison of the two approaches to maintaining a set forecasts. In the left, code from ASM-2.0 is presented, which shows the dynamic allocation of a chunk of memory (enough to hold pointers to *numfcasts* structs), and then a for loop over the structs sets their initial values. After the array of structs is set, then this example shows how one can traverse over the elements in order to compute the average of a variable called *specificity*.

On the right side of Table 1, the more object-oriented version of this exercise is shown from ASM-2.2. Instead of treating the elements as structs, the elements are now objects from the class *BFCast*. The creation and initialization of those objects is segregated into a method in *Bfagent* called *createNewForecast*, and any time a new forecast is needed, that method is used. Following that approach eliminates the risk that one might forget to set all of the numerical values correctly when doing the initialization in the middle of the other methods, as is done on the left hand side. On the right hand side, one can see that the objects are put into a collection called *fcastList,* and when it is time to compute average specificity, an iterator object called *index* is obtained from the *fcastList*, and then a loop repeatedly asks for the next element of the list.

The main point of this section is that storing objects into a collection is easier and less error-prone than the other way. There is another point to recommend it, however. It would be a bit difficult to add new forecast objects if the old approach were followed. The amount of memory allocated is fixed, and so creating more "headroom" in the collection would

require further explicit work to allocate memory. Doing that would significantly complicate the pointer math that is used to transverse the collection. On the other hand, most collections libraries will make it easy to increase the size of a collection. A list class will simply grow whenever an object is added. In the Table 1 example, a Swarm Array collection is used, mostly because the retrieval of specific items is faster with than class than in an unstructured list. But the Swarm Array has a method which allows it to be resized. Of course, the same work gets done if one uses either approach. The difference is that, in the latter approach the work is done automatically, inside the (hidden!) confines of a collections library. The work is done in a well-tested, multipurpose collections library, rather than in the middle of a substantive application. This frees the user from a significant amount of detailed work.

Hint #4. Think about parameter management.

The Swarm Simulation System has features in it for creating objects and setting their initial values to particular parameter values. The input parameters are kept in separate files, either in LISP format or the National Center for Supercomputing Application's (NCSA) HDF5 data format (http://hdf.ncsa.uiuc.edu/index.html). In ASM-2.2, the LISP approach has been taken, and the default settings for the model are in a text file called *asm.scm*. (The extension .scm refers to the Scheme language, an implementation of LISP.)

One point of emphasis is to separate the adjustment of parameter values from the editing of the code itself. If the numerical values of the parameters are buried inside a file that includes C code, then it is somewhat inconvenient to dig in, find values, and adjust them. Furthermore, one must recompile the model in order to run it. It is much nicer to edit a separate parameter file and run the model again without recompiling.

The new parameter classes in ASM-2.2 are BFParams and ASMModelParams. The BFParams class contains the default settings that govern the creation of bitstring forecasting agents and the bitstring forecasts themselves. Each agent maintains many forecasts,

repeatedly comparing their quality in predicting the stock price. When an object is created from the BFCast class, the Bfagent's method *createNewForecast* is responsible for reading values of of the BFParams object and setting them in the new forecast. Because of this object-oriented design, it is possible to allow changes in BFParams over time, possibly reflecting the experience of the agent.

Another big change in ASM-2.2 is the separate parameter class for the overall model, ASMModelParams. This affords a significant simplification of the maintenance of the code. This is so because Swarm models are typically written to run either in the graphical (GUI) mode or in a batch mode (without any GUI interaction). The model itself is supposed to be the same, of course, but the top level object which manages the simulation will differ. In the GUI version, for example, the top level class is called ASMObserverSwarm, while in the batch/noninteractive version it is called ASMBatchSwarm. Because the ASMModelParams are broken off into a separate class, then either of these top level models can access the same set of parameters.

In ASM-2.2, all parameter management chores are gathered together under the umbrella of the Parameters class. Parameters is subclassed from Swarms Arguments_c class, and the end result of that is that Swarm's default object *arguments* is customized to fit the application. When a simulation model begins, the first thing that happens is the creation of a Parameters object, and when it is created, it spawns instances of ASMModelParams and BFParams, so those values are ready when the other objects are created.

Thinking ahead to the end of the project, when one wants to run a model over and over again, or explore the impact of changes in conditions, it is important to note that the Parameters class can also handle command line parameters. When the model runs, the command line options are sorted, and the parameter objects can be told to adjust their settings.

If a program is able to accept command line parameters, then there are easy

approaches that can be used to generate many runs of the model. The most sophisticated approach of which I am aware is Ted Belding's Drone program (http://Drone.sourceforge.net), which has high powered capabilities to manage parameter sweeps and distribute jobs across networks. Drone relies on the Expect package, and we have had some trouble building that for MS Windows systems (but not on Unix/Linux), so I prepared a Perl script that can run parameter sweeps as well (http://lark.cc.ukans.
edu/~pauljohn/Swarm/MySwarmCode/replicator.pl), but it will not distribute jobs around a network the way Drone can.

Hint #5. Consider strategies for data output.

In the past, I have used C and the standard methods of creating output files in text format. This is convenient and there is always confidence that the results will be useful across a variety of programs. In the ASM-2.0 *Output* class, one can find the code that causes a simulation to write plain ascii values into a file. This is one of the approaches illustrated in ASM-2.2. The file's name is stamped with the time and date. Model runs in batch mode will automatically turn-on the data writing facilities, but runs in the graphical interface will not write data unless the user pushes a button that turns on data writing.

There are some shortcomings to raw text output, however, one of which is the lack of precision in real-valued numbers. Marcus Daniels of the Santa Fe Institute and the Swarm Development Group has integrated the National Center for Supercomputing Application's HDF5 data format into Swarm, so that data can be saved in a compressed format that retains precision. Mr. Daniels has also written code to support HDF5 usage for the free software statistical program R (Ihaka and Gentleman, 1996; see http://www.R-project.org) , which is free, open source, increasingly convenient, high powered, and fun-to-use tool for creating graphs and estimating models. One can generate HDF5 files from a Swarm simulation, read them into R, analyze the data, and output the data in other formats if desired.

In ASM-2.2, these capabilities have been put to use. In order to take advantage of them, however, one needs a fairly new edition of Swarm, either a development snapshot of Swarm from July, 2001 or later, or a release version of Swarm-2.2. There are two kinds of hdf5 output that can be saved in ASM-2.2. First, there is a small HDF5 file that includes the time-series of the variables *price*, *dividend*, and *volume*. That file is created through Swarm's EZGraph class, which is typically used to show graphs on the screen, but in this case it saves values into a file rather than on the screen.

The other kind of output that the Swarm libraries can generate is typically called "serialization" information. Serialization means the ability to save the state of an object into a file and then, at a future time, re-create the same object. (People who write computer games have, of course, mastered the art of serialization.) Swarm can output the contents of whole objects in either a LISP format or HDF5. In ASM-2.2, we are not interested in a full serialization, though, and we can save some time/storage space by only saving a shallow copy of the various objects. A shallow copy records the variables inside an object, but not the objects that it has within it. In ASM-2.2, the default code will write out a LISP formatted text file that includes the data of the Specialist object and the World object. At the end of a run, one should see files like:

output.dataWed_Oct_24_11_30_18_2001

swarmDataArchiveWed_Oct_24_11_30_18_2001.scm

hdfGraphWed_Oct_24_11_30_18_2001.hdf

If, instead of the LISP formatted swarmDataArchive file, one wants HDF5 formatted output, it can be done with either a C preprocessor flag or an obvious change in the file Output.m. The usage of the LISP archiver in this example significantly slows down the speed with which the model runs because it archives a great deal of information, much of which is not

needed.

Conclusion

While working on the ASM code, several issues about model-building and programming came into focus. It is better to encapsulate information and retrieve it when necessary than it is to leave information sitting about where it can be accidentally altered. It is better to use container classes than to do math with pointers. It is a good idea to design a simulation model with an eye toward the end product, both in terms of exploring parameter combinations and data output.

All of these observations flow together into the contention that **object-oriented modeling is good**. Or, perhaps in the more cynical style of Winston Churchill, "Object-oriented coding is the 'worst' form of model-building, except all those others that have been tried from time to time." One can readily criticize the status of ASM-2.2 because there are things that could be further isolated into specific classes of their own. There are cases in which the scope of some variables could be made smaller. I have stopped at this stage for a release because the changes have been quite substantial, and further revision is likely to require more significant breaks in the conceptual alignment of the code. I hope that people who want to learn about Swarm in particular, or agent-based modeling in general, might benefit from a comparison of the various versions of the Artificial Stock Market.

References

Johnson, Paul E, and Alex Lancaster. 2000. *The Swarm User Guide. Http://www.santafe. edu/projects/swarm/swarmdocs/userbook/userbook.html*

Ihaka, Ross, and Robert Gentleman. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics.* 5(3): 299-314

Kernighan, Brian W. and Rob Pike. 1999. *The Practice of Programming*. Reading, MA: Addison-Wesley.

Kernighan, Brian W. and Dennis W. Ritchie. 1988. *The C Programming Language.* Englewood Cliffs: Prentice Hall.

Kochan, Stephen G. 1994. *Programming in ANSI C*, Revised Edition. Indianapolis, IN: SAMS Publishing.

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

Minar, Nelson, Roger Burkhart, Chris Langton, and Manor Askenazi. 1996. The Swarm Simulation System: A toolkit for building multi-agent simulations. Santa Fe Institute Working Paper 96-06-042, Santa Fe, NM. (available at http://www.swarm. org/archive/overview.ps)

Palmer, R.G, W. Brian Arthur, John H. Holland, Blake LeBaron, and Paul Taylor. 1994. Artificial economic life: a simple model of a stockmarket. *Physica D* 75: 264-274.

Table 1
Collections Usage in ASM-2.0 and ASM-2.2

```
//Example Code from ASM-2.0 (BFagent.m):
// Declare a pointer to space for BF_fcast
struct
struct BF_fcast *fcast;
// Allocate memory for forecasts
  fcast = calloc(p->numfcasts,sizeof(struct
BF_fcast));
  if(!fcast)
    printf("There was an error allocating space
for fcast.");

struct BF_fcast *fptr, *topfptr;

// Initialize the forecasts
  topfptr = fcast + p->numfcasts;
  for (fptr = fcast; fptr < topfptr; fptr++)
    {
      fptr->forecast = 0.0;
      fptr->lforecast = global_mean;
      fptr->count = 0;
      fptr->lastactive = 1;
      fptr->specificity = 0;
      fptr->next = fptr->lnext = NULL;
 /* Allocate space for this forecast's
conditions out of total allocation */
      fptr->conditions = conditions;
      conditions += condwords;
 /* Initialise all conditions to don't  care /
      cond = fptr->conditions;
      for (word = 0; word < condwords; word+)
        cond[word] = 0;
  /* Add non-zero bits as specified by
probabilities */
      if(fptr!=fcast) /* protect rule 0 */

/* Compute average specificity */
  specificity = 0;
  for (fptr = fcast; fptr < topfptr; fptr++)
    {
     specificity += fptr->specificity;
    }
  avspecificity = ((double) specificity)/p-
>numfcasts;
}
```

```
//ASM-2.2 equivalent:
//Create Array to hold forecast objects
 fcastList=[Array create: [self getZone]
                        setCount: numfcasts];

// Create Forecast objects, add to fcastList
// The "know nothing" forecast is first
  [fcastList atOffset: 0 put: [self
createNewForecast]];
//create rest of forecasts with random
conditions
  for ( i = 1; i < numfcasts; i++)
    {
     id aForecast =[self createNewForecast] ;
//initialize
     [self setConditionsRandomly: aForecast];
     [fcastList atOffset: i put: aForecast];
    }

/* Compute average specificity */

 index = [ fcastList begin: [self getZone] ];
 for ( aForecast = [index next];
      [index getLoc] == Member;
       aForecast = [index next] )
    {
    sumspecificity += [aForecast
getSpecificity];
     }
  avspecificity =
           (double)  sumspecificity/(double)
numfcasts;
  [index drop];
}
```

Figure 1
Screenshot of ASM-2.0

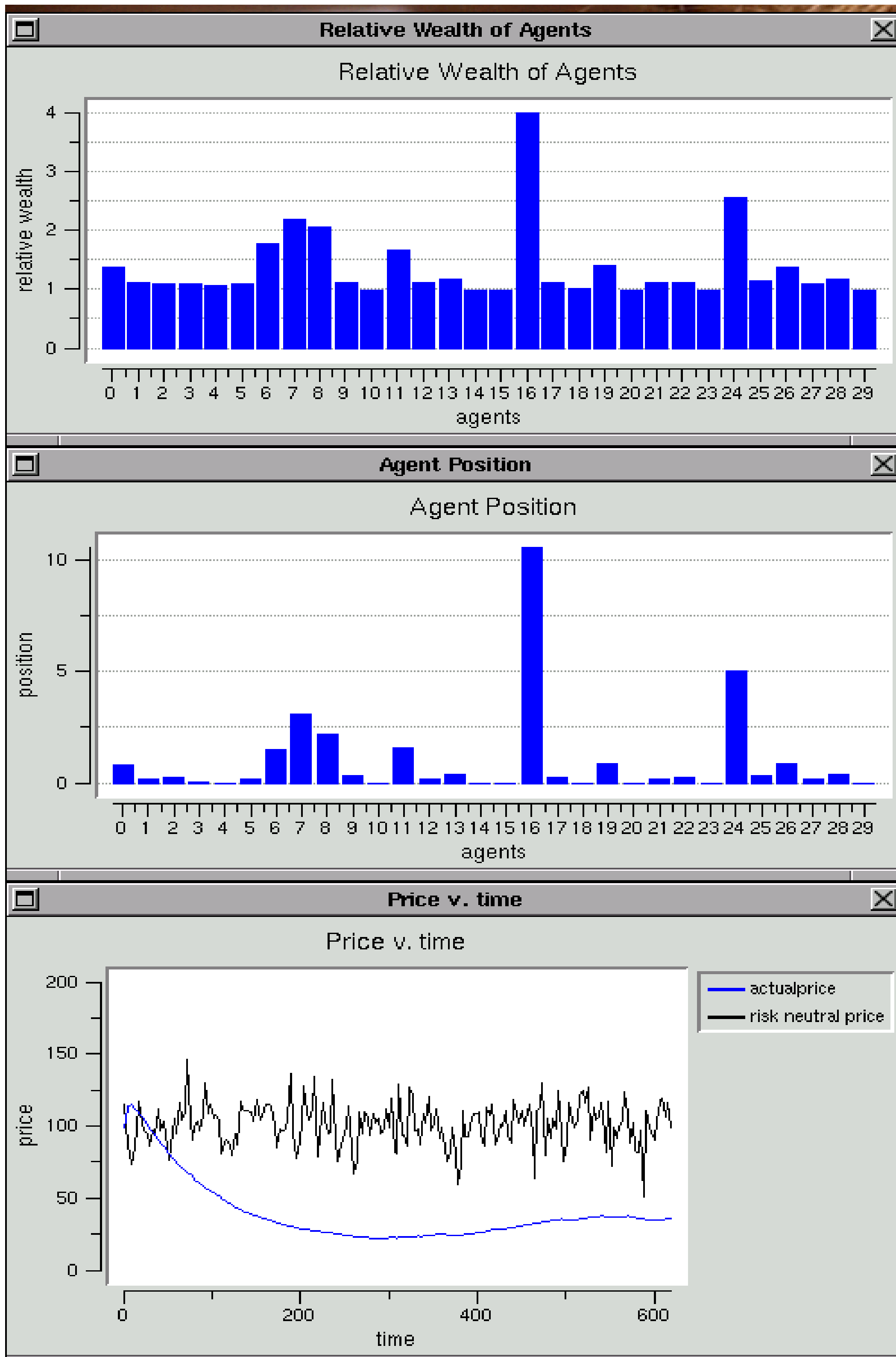

Figure 2
Screenshot of ASM-2.2