PART I

A Java Library Implementation of the Gold-Food Economic Model in Repast and MASON

Christopher K. Chan
Dept. of Computer Science
Princeton University
Princeton, NJ 08544

Date: January 7, 2008

Adviser: Prof. Ken Steiglitz

A Java Library Implementation of the Gold-Food Economic Model in Repast and MASON

Christopher K. Chan and Ken Steiglitz
Dept. of Computer Science, Princeton University
Princeton, NJ 08544

January 7, 2008

Abstract

We present a robust, extensible Java library for agent-based economic simulations. The economic model is based on the two-commodity, auction-mediated computational market described in Steiglitz et al. [8]. We implement the library on top of both the Repast and MASON agent-based Java toolkits, and compare these toolkits in terms of their feature set, documentation, execution speed, and flexibility. Though these toolkits represent distinct attempts at creating general-purpose environments for agent-based simulations, we find that their design differences are insignificant with respect to their use in our specific agent-based model, and we argue that Repast is the more appropriate toolkit for our application. Finally, we validate the correctness of our implementations by utilizing the libraries to reproduce the results from work based on the original simulation program.

1 Introduction

Steiglitz *et al.* [8] defines an agent-based computational market model consisting of a simple food-and-gold economy, wherein simulated agents with production and trading decisions, and speculators exploiting profit opportunities, interact to generate economic phenomena of interest. The work "represents an attempt to construct an economic model – albeit imperfect and highly idealized – that is complete down to all individual actions" [9]. By altering the assumptions about the market – for example, changing the agents' decision space or introducing new arbitrage strategies – and simulation the agent interactions over many time steps, researchers can explore how economic factors might influence the competitive supply-and-demand equilibrium in a model economy[1].

[1] The computational market model is thoroughly described in [8, 10]. For areas in economics that the model has been used to explore, see [1, 2, 3, 5, 7, 8, 9, 10].

However, in order to continue research using this model, we believe that a from-scratch rewrite of the original C program was required. Despite numerous attempts to improve the original codebase, from a software standpoint the program remained inflexible and difficult to understand. Due to its monolithic program design, any attempt to change the market model required modifying the core simulation code, a process which can introduce logic errors or bugs that undermine the validity of the simulation's results. Furthermore, because previous research threads each modified an independent copy of the entire program, fixes or improvements to the core code in one thread might not have been propagated back to simultaneous working copies. Usage of the computational market model, which was linked with the development of the underlying simulation program, effectively stalled a decade ago when the code became too disorganized for passing researchers to quickly learn and employ. In restarting this project, we were reminded of "Eagleson's Law"[2] and thus sought to implement the market simulation using an entirely different software approach.

The primary goal of our work is to design and implement a modular, reusable and extensible agent-based software library based on the computational market model in [8]. We first identify the aspects of the market model that should be modularized, and then describe our resulting two implementations that use the popular open-source, agent-based simulation toolkits Repast and MASON. Our rationale for programming the market model in both toolkits is so that we can explore the differences between toolkits and ensure that we chose the toolkit that best suited our application. We then evaluate these two toolkits in terms of their feature set, documentation, execution speed and robustness, and ultimately recommend our Repast implementation for further development. Lastly, in using our implementations to reproduce the results of past research, we find that our Java library can greatly aid in extending the computational market model, and it makes using the simulation a more viable and attractive research option.

[2] Eagleson's Law: Any code of your own that you haven't looked at for six or more months might as well have been written by someone else. (unknown origins)

2 Implementation

We created two independent implementations of the market model that differ in the underlying simulation toolkit they use (see Section 2.2), but share the same class structure for the market model components (see Section 2.1). The implementations each consist of an economic simulation library customized for the particular toolkit[3] and six proof-of-concept applications that validate the correctness of the library code (see Section 4). The library is essentially a modularization of the economic components of the market model. It relies heavily on inheritance and a consistent software interface to allow researchers to build highly customized derivatives of the original model. By implementing and overwriting library classes, one can add complexity to and change the components of the market model. In Appendix A, we include screenshots of example applications that utilize the two implemented libraries. Our implementations are written entirely in Java and have been tested for compatibility on Java versions 1.5 and 1.6.

2.1 Model Components

The main components of the model in [8] are the *agents*, which produce the buy and sell orders for food in units of gold, and the *market*, which decides how to processes these orders. Agents called *workers* are endowed with varying abilities to produce either food or gold, but not both, at each period. Workers must also consume a unit of food every period, and each worker tries to maintain a minimum inventory of food to guarantee consumption. For the system to survive the maximum production of food must be greater than the total food consumption per period, but this leads to a surplus of food in the economy. The market allows agents to sell their surplus food for gold that can be used to purchase food in the same market later on. Agents more skilled in the production of gold might decide to mine gold and trade it for food at every period. As one might expect, the combination of the agents' production and

[3] Due to incompatibilities in the toolkits' design patterns (see Section 3.1.2), we were unable to encapsulate the logic for the market model into a single toolkit-independent package. An unfortunate consequence is that our two implementations contain large amounts of duplicate code.

allocation decisions generates the concept of a “price” for food in units of gold, and it is the hope of every economist that this price might approach the competitive equilibrium where net supply meets net demand.

Fig. 1 illustrates the relationships amongst the components of the model and the ways in which they can be added, removed or transformed to simulate different economic factors. The model components and their interactions in our library are as follows:

2.1.1 Agents

Every agent maintains an inventory of food and gold, which is represented by the `AgentBase` abstract class. The agents incorporate the conditions reported by the market in their production and trade decisions. Currently, the latest market price for food is the only market condition being utilized by the agents, but other factors such as trade volume or market size can be included in future iterations. Agents are further separated into workers (`WorkerBase`) and speculators (`SpeculatorBase`). The first category of agents produce, consume and trade the two commodities in the market. The workers differ in their initially endowed farming and mining skills, utility

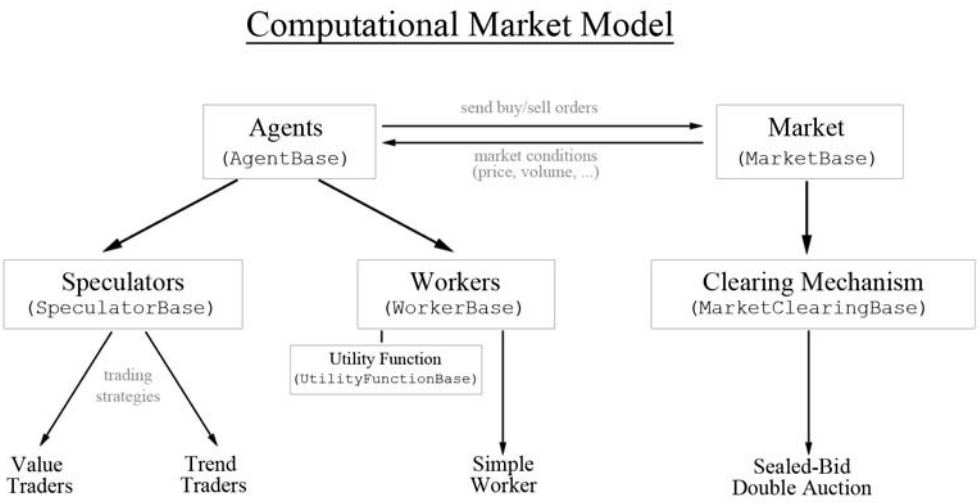

Figure 1: *Diagram of how the components in the computational market model interact. Squares represent abstractions while components without squares are concrete implementations.*

for food (`UtilityFunctionBase`), and decision space determining what to produce and trade. All worker implementations must extend `WorkerBase`, an abstract class that defines a common interface for comparing different workers. We created a basic `SimpleWorker` class that implements the "regular agents" described in [10].

The second category of agents is not involved in production or consumption, but attempt to earn profits by trading the two commodities. All speculators (or pure traders) must extend the abstract class `SpeculatorBase`. The three default speculator implementations named `ValueTrader`, `TrendTrader`, and `GlobalValueTrader` are based on the "AVG Speculator" and "DER Speculator" in [8], and "international trader" in [1], respectively. These speculators differ in their trading strategies, and they serve to exploit arbitrage opportunities and create temporary deviations from equilibrium price.

Our library is not limited to the workers and speculators already implemented. Researchers can change default agent behavior or define new agents by extending any of the "base" or implemented classes and overwriting their methods.

2.1.2 The Market

The market contains the logic that processes trade orders from agents. It can also be viewed as a pseudo-government in the sense that market implementations can define tariffs, trading rules and other deviations from a free-market economy. Every market must extend the `MarketBase` abstract class, which defines the interface that allows agents to interact with the market. The market maintains a list of its agent participants in order to calculate the fundamental equilibrium price of the underlying commodity. At any period, only a single market price should exist in a market, and this rule is enforced in the `MarketBase` interface.

Additionally, each market contains a market clearing mechanism (`MarketClearingBase`), an entity responsible for matching and filling the orders sent to the market. The clearing mechanism can take the form of an auction or can be as exotic as the "black box" deciding a command economy. We

separated the market clearing logic into its own component, so that one can endlessly customize the economic structure without having to change the agent-market interface. Our library includes the sealed-bid, single-price call auction that attempts to maximize trade volume (`SealedBidMaxVolumeAuction`) as detailed in [8].

2.2 Agent-based Toolkits

The effort to develop the original C program stalled sometime in 1997. Since then, numerous open-source, agent-based simulation "toolkits" have been created to facilitate research in broad areas of agent-based modeling. At their core, these toolkits provide software infrastructure for defining agents and scheduling agent interactions. This enables researchers to focus on modeling agent behavior, leaving technical issues such as multi-threading and saving program states for the toolkits' software developers. In addition to the infrastructure they provide, toolkits often include statistics packages, charting, visualization and other useful tools. The main drawbacks of toolkits are the limitations they place on the complexity of models and their execution speeds are limited by the programming language and the skills and objectives of the developers.

Despite of their potential limitations, agent-based toolkits are indispensible starting points for almost every agent-based simulation project, and ours is no exception. Toolkits greatly differ in terms of their software design philosophies [6], so choosing an appropriate toolkit can affect how a particular model is programmed. However, the sheer number of free and commercial toolkits available complicates the search. Due to the fast-paced nature of software development, no comprehensive review of toolkits is available or is likely ever to be published. However, we took into account the assessments by Tobias and Hofmann [11] and Railsback *et al.* [6] in our eventual choice of toolkits.

The study by Tobias and Hofmann [11] in 2004 compared the Objective-C Swarm toolkit and four freely available Java-based toolkits for use in social-science applications. Their study recommended Repast because of its extensive documentation, design philosophy, and the experiences of its evaluators and users they surveyed. The thorough and more recent study by Railsback *et al.* [6] examined five

popular agent-based platforms in terms of execution speed and their experiences in implementing a generic agent model. This study highlighted the strengths and weaknesses of each platform, but stopped short of endorsing a single toolkit. Both studies found the Objective-C Swarm toolkit, which is considered the father of all agent-based toolkits [6], to be the most appropriate for expressing agent-based models. However, like Tobias and Hofmann, we also believe in the importance of choosing a toolkit that is written in conventional, object-oriented language like Java, because the underlying programming language can influence the attractiveness of software to future researchers. For this reason, we decided against the Swarm toolkit in favor of Java or C/C++ based alternatives.

Of the remaining toolkits reviewed, the mature Java-based Repast toolkit [4] was the most logical choice for our market model simulation, because it features extensive documentation, a large, active user community, and was recommended by both studies. The development of the toolkit was prompted by the need for an easier to understand, Java-based alternative to Swarm [7]. In addition to Repast, we were also drawn to the recently developed MASON toolkit [3], which also uses the Java language and emphasizes execution speed for computational intensive models [7]. These two toolkits seemed the most attractive out of those reviewed. Rather than risk passing on a toolkit that better suited our economic application, we decided to implement our library in both toolkits. In doing so, we also hope to gain a better understanding of the potential differences and limitations of employing toolkits in general.

3 Comparison of Toolkits

The Repast and MASON toolkits form solid foundations for the agent-based economic simulation library we implemented. They provide our libraries with powerful scheduling mechanisms, built-in charting abilities, and intuitive GUI interfaces to update values in a simulation model. These features can help researchers quickly visualize, validate and gain confidence in their simulation results. Our particular agent-based model utilizes only a fraction of the total features included in these toolkits. But we can imagine how built-in features as spatial grids and Monte Carlo statistical modeling might make these toolkits highly valuable to research in other areas.

From a programming standpoint, we found Repast and MASON toolkits to be remarkably similar. This is not surprising because the development of MASON was heavily influenced by the perceived shortcomings of Repast [3]. The programming patterns promoted employed by both toolkits resulted in nearly identical logic for the market component implementations. Our experiences with these two toolkits reinforce the findings in Railsback *et al.* [6], and in particular, the observation that MASON's scheduling design restricts the complexity of the model (see Section 3.1.2). Due to these shortcomings, the MASON toolkit may have taken a step back from its predecessor. We supplement the highly insightful discussion in [6] with the observations in the following subsections.

3.1 Toolkit Design

3.1.1 Visualization Issues

Real-time visualizations such as time-series charts which display accumulated data significantly reduce the execution speed of a simulation. On our test machine, the charting tools used in Repast and MASON often crash when datasets become too large. A common complaint with Repast is that it integrates visualizations with the simulation engine [6], which makes enabling and disabling visualizations very inflexible, but real-time visualization much smoother on its dedicated visualization tools. In contrast, MASON by design decouples the two features, and instead relies on large external libraries for display. While this decoupling is important in theory, it is not practical for several reasons. First, real-time visualizations are more likely to be used in brief simulation runs, where the fluidity of display is key. However, because MASON employs large external libraries rather than implementing display tools internally as in Repast, the real-time visualizations on MASON seem to run much slower even when data is sparse. For long-running simulations, the large datasets cause memory issues with the Java-based display utilities, so in these cases, it is better to use dedicated graphic utilities in Matlab or Excel anyway.

3.1.2 Scheduler Design

Railsback *et al.* [6] notes that the major disadvantage with MASON is its inflexible scheduler design, which is a result of its use of a "template method design pattern." Any object scheduled in MASON must implement the `Steppable` class and all actions must occur within the single `step()` method defined in the object class. This design pattern forces us to integrate the scheduler with the model components in the MASON implementation. The Repast toolkit uses a more conventional approach, whereby a scheduler runs a list of event objects, each of which are free to address any aspect of the scheduler-independent economic library. To illustrate the difference between these schedulers, we can look at the following example:

Suppose a simulation contains a list of agents that send buy and sell orders to a market, and the market processes these orders after they are all sent. The most straightforward way to program these actions in each library is as follows:

> **Repast Implementation:** first, call a `sendBid` method for each *agent object*, and then call a `processBids` method on the single *market object* to clear all the bids. All of these actions can be contained within one *event object* scheduled to repeat at every simulation step.

> **MASON Implementation:** in the simulation setup, register with the scheduler each *agent object* and then the single *market object* in that order. When simulation runs, the scheduler summons the `step()` method for each *agent object*, wherein a private method `sendBid` is called. Then the scheduler summons the `step()` method for the single *market object*, wherein a private method `processBids` is called. The scheduler repeats this process at every simulation step.

Now, suppose we want to simulate two such economies simultaneously, where each economy acts independently to process its agent orders. We then introduce a single trader that can send trades to

each of the two economies. In fact, this special trader can make trading decisions based upon an endless number of economies. The most practical way to model these relationships is to create an *economy class* that encapsulates the storage and actions for a group of agents and their market. While this is straightforward to do using the Repast implementation, we immediately encounter issues when using the MASON implementation. With the latter toolkit, we would need to register an *economy object* with the scheduler, but since the *economy object* stores the agents and market, these objects can no longer be registered with the scheduler. In fact, we encountered these exact issues while coding an international trade model (see Section 4.6). Essentially, as the simulation model becomes hierarchical in nature, the MASON scheduler can become increasingly difficult to work with.

3.2 Execution Speed

We first evaluate the execution speed of the Repast- and MASON-based library implementations by testing how they scale to agent-based models of increasing size. The most expensive per-step computation in the simulation is in the marketing clearing mechanism, which uses Java's system sort to order the list of bids and asks in the auction. The on-average runtime for the system sort is O(n log n). The remaining per-step logic, including the allocation and trading strategies of the agents, is linear in runtime. We therefore expect that as the number of agents in the system increases the runtime should scale according to the Java system sort.

We created test applications that differ in the library implementations they use to simulate the baseline market model (see Section 4.1). Each application is run for 1000 simulation steps with only worker agents contributing to the market. The testing environment is a Windows-based machine with a Pentium M 1.73 Ghz processor, 2GB memory and Java Runtime version 1.6.3. At 1,000,000 agents, both simulations encounter memory errors so simulations of that size could not be run. Fig. 2 displays the results of the timed tests, which are averages over three trials. For simulations between 1,000 and 100,000 agents, the results appear to scale as we expected.

Implementation	1,000 Agents	10,000 Agents	100,000 Agents
Repast	3.58 ± 0.29 sec	25.75 ± 0.62 sec	570.24 ± 33.80 sec
MASON	3.24 ± 0.06 sec	40.01 ± 1.53 sec	600.62 ± 26.12 sec

Figure 2: *Execution speeds of Repast- and MASON-based library implementations with visualizations disabled.*

Our second test compares the execution speed of the two library implementations when a real-time graphing display is enabled. The speed of the visualizations is dependent on the toolkit's graphics implementation. The Repast toolkit provides custom charting facilities, while the MASON toolkit instead offers wrapper methods around the open-source JFreeChart library. Generally, visualizations significantly reduce the speed of a simulation, so they are mostly useful in models with few agents and short simulated durations. We evaluated the execution speeds using the same test applications as before but with visualizations enabled. In Fig. 3, we can see that the graphing functionality in the two toolkits is severely limited for large simulation sizes. The JFreeChart library employed by the MASON toolkit appears to be significantly slower than the Repast visualizations, though both graphic facilities are unacceptably sluggish when the simulation contains over 10,000 agents.

Implementation	1,000 Agents	10,000 Agents	100,000 Agents
Repast	16.37 ± 0.46 sec	31.28 ± 0.83 sec	417.04 ± 12.33 sec
MASON	54.04 ± 2.51 sec	82.90 ± 2.30 sec	588.94 ± 18.68 sec

Figure 3: *Execution speeds of Repast- and MASON-based library implementations with visualizations enabled.*

One of the main design objectives of the MASON toolkit is to offer a *fast* environment for agent-based simulation [3], so it is rather surprising to find that the MASON implementation performed as slow, if not slower, than the Repast implementation in our tests. More importantly, these results suggest that there are clear limitations in using either of these toolkits for large-scale simulations.

3.3 Documentation

The active mailing lists, thorough documentation, code samples, and user groups all attest to the maturity of the Repast toolkit. The toolkit includes several tutorials on their website that familiarizes a

new user to the environment in a matter of hours. The developers of MASON also wrote comprehensive tutorials for their toolkit, which are similar in scope to those for Repast. The MASON tutorials can be found in the downloaded toolkit folder. Both Repast and MASON provide sample implementations of many classic agent-based models, which serve to introduce a user to advanced features of the toolkits.

Though both toolkits offer abundant starting material, we find that the most important difference between the two is in their user community or lack of one. It was particularly difficult to search for anything related to MASON online, because its user base has not had enough time to grow. However, the Java source code for both toolkits is very well commented, so an experienced programmer will probably find the Java-generated documentation sufficient.

4 Evaluation of Implementations

We utilize the libraries we implemented to reproduce six key results that were based on simulations from the original C program. These results encompass both unpublished student work as well as journal articles regarding the computational market model. We create twelve applications in total – one for each result in the two library implementations – and compare the results of these simulations with descriptions by the original researchers. The extents to which the results agree help us determine whether the market model is correctly implemented.

These twelve applications also allow us to measure the robustness of our economic library. In the following sections, we provide an estimate of the total new classes, methods and lines of code required to implement each derivative of the baseline model[4]. Though these measures are not perfect proxies for robustness, we hope that these results provide a general sense of the flexibility of the library. Finally, the reader should note that, because of the original program's software design, many of these single simulation findings took students entire semesters to code and obtain. The fact that we can utilize the

[4] We report the total number of *new* lines rather than *changed* lines, because adding new lines is generally more work intensive than modifying existing lines.

library implementations to explore these results in hours rather than weeks indicates the progress our implementations have achieved.

4.1 Baseline Gold-Food Model

The work in Steiglitz *et al*. [8] reveals that in a simulation consisting only of working agents the market price for food oscillates wildly and does not stay at the long-run competitive equilibrium price. However, the introduction of "value" speculators –agents that trade for profit according to an adaptive expectations model – stabilizes the market price in the long-run. Our simulation initially consists of 100 workers endowed with varying production skills that are randomly set at the start and fixed throughout the simulation. At time t = 1000, 20 value traders are introduced into the market. As we expect, the left graph in Fig. 4, which is generated by our simulation program, shows that the market price for food indeed fluctuates until the point when the speculators are introduced. We juxtapose the corresponding graph from the work in [8] for comparison.

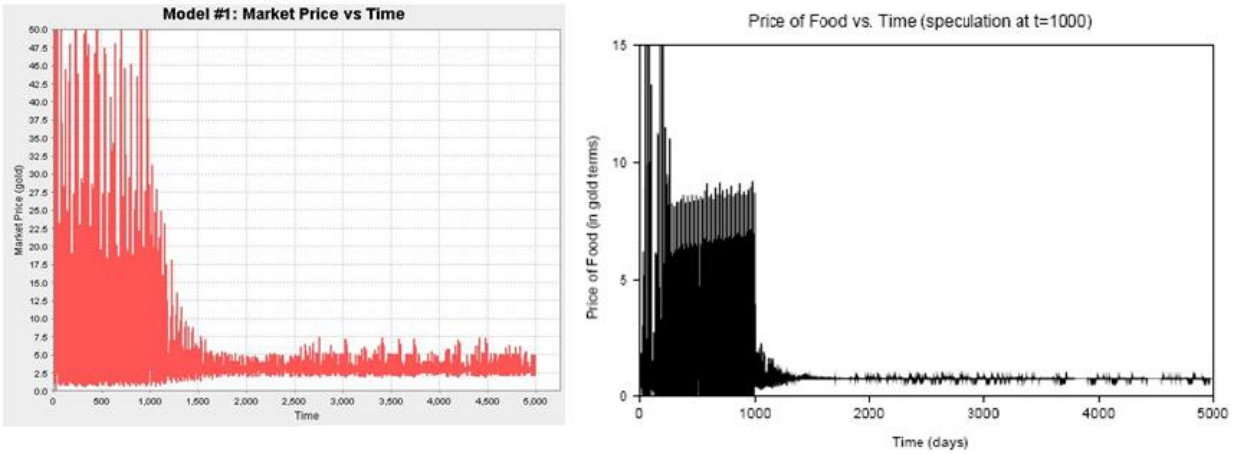

Figure 4: The left graph, which is created by our simulation, shows the price of food over time, with the introduction of 25 value traders at t=1000. The market price clearly approaches the competitive equilibrium price (not shown). On the right, the graph from Steiglitz et al. [6] shows a similar scenario but with different initial conditions

4.2 Exogenous Price Bubbles

Price bubbles in an economy are short-term deviations from the competitive equilibrium price. In the real world, such bubbles are synonymous with periods of irrational behavior and high speculation in trading markets. Steiglitz and Shapiro [10] report that price bubbles can form in the market simulations when trend-based traders are introduced *and* when the market equilibrium price is exogenously perturbed (see Appendix B). We explore this effect by implementing the `TrendTrader` class, a speculative agent that is only motivated to trade by price movements. In our two applications, we schedule a per-period event which multiplies the workers' gold skills by a sinusoidal smoothed constant. In Fig. 5, we see the effect of this exogenous perturbation on the skills, and in particular, the quickness at which the market price adapts to the changing supply and demand. Furthermore, a price bubble can clearly be seen at around t=4500, and it is consistent with the graphs and descriptions in [10].

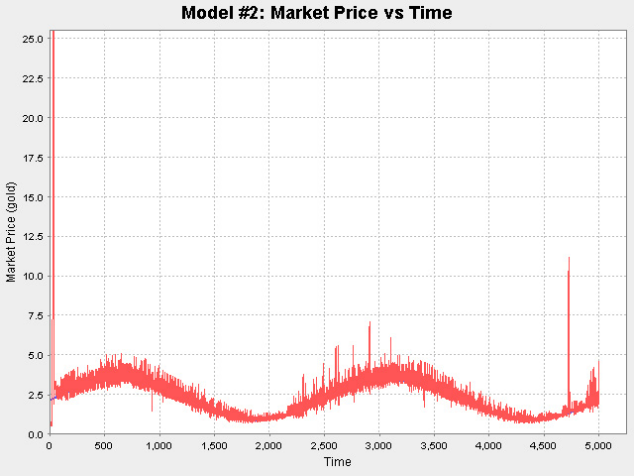

Figure 5: *Price of food over time, with workers' skills levels exogenously perturbed in a sinusoidal pattern. We clearly see price bubbles at around t=4500 and intermittently between t=2500 and t=3000.*

In terms of the programming involved, these results are very straightforward to reproduce. Beyond the implementation of the `TrendTrader` class, which is already included in our library implementations, the only coding involved is the creation and scheduling of the exogenous perturbations, which add roughly 40 lines of code to the application's main class.

4.3 Endogenous Price Bubbles

The work in [5, 9] suggest that price bubbles can also be formed endogenously by "jumpy" trend traders that are willing to trade for very low margins. These jumpy traders might model inexperienced traders prone to taking on excessive risk or risk-seeking individuals that do so for reasons of utility. O'Callaghan and Steiglitz [5] suggests that the price bubbles form because these jumpy traders initiate move prices enough to entice less risky trend traders to also buy or sell in their direction. The bubbles soon collapse, likely because of value trading strategies.

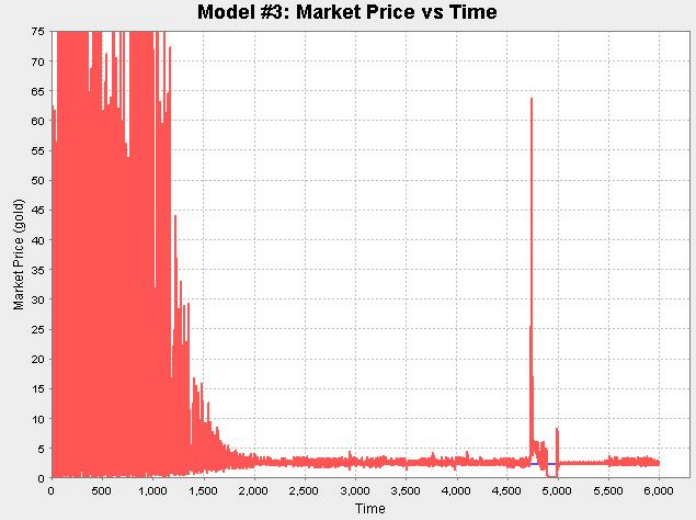

Figure 6: *Market price over time. The market begins with 25 workers. At t=1000, 25 value traders and 25 trend traders are introduced. From t=3000 to t=5000, five "jumpy" trend traders participate in the market. Price bubbles appear between t=4700 and t=5000.*

We follow the description of the simulation model, and define special trend traders with margins that are only a tenth of the normal trend traders. A fixed amount of gold is injected to all traders during the "risky trading" period in order to prevent bankruptcy for the traders who follow the "jumpy" speculators. In Fig. 6, we can clearly see positive and negative price bubbles forming during the period when these irregular traders are present in the market. The market price appears to return to the competitive equilibrium immediately following the removal of these risky traders.

The library implementations of the `TrendTrader` class include a constructor to customize a trader's margin, so one can check these results by adding a mere 5-6 lines of scheduler code.

4.4 Noisy Prices

Lenormand and Steiglitz [2] explores the effect of adding noise to the clearing price reported to the agents by the market. This noise models the fact that participants in real-world markets do not always have the most accurate information regarding commodity prices. The simulation work finds that noisiness in the market causes price bubbles to form when the noise term is small, but causes market prices to destabilize when the noise term is large. Our test application calculated the noisy price at period t using the formula

$$P'(t) = \alpha \cdot \gamma + P(t)$$

where α is a noise factor and γ is a Gaussian random variable with mean 0 and unit variance[5].

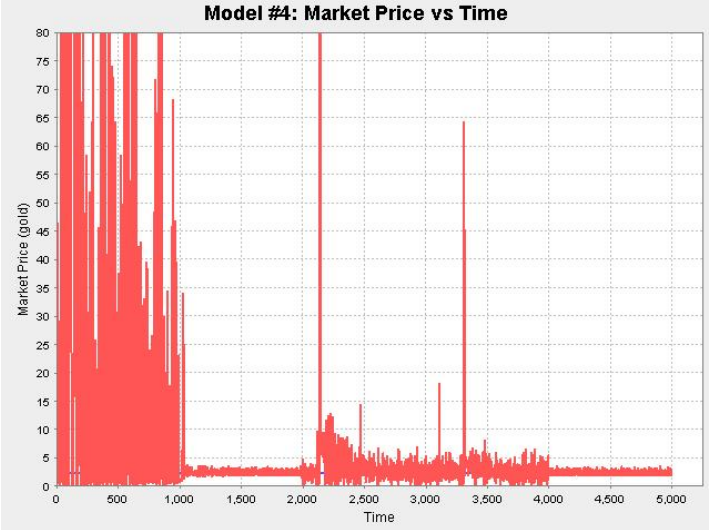

Figure 7: *Market price over time. The market begins with 25 workers. At t=1000, 25 value traders and 25 trend traders are introduced. Between t=2000 and t=4000, the market reported a noisy price to agents. The noise calculation used a factor of 0.75.*

[5] The noisy price calculation we used came directly from the work in Lenormand and Steiglitz [5]. However, the term should be normalized by the equilibrium price in order for it to be relevant for a wider range of initial conditions.

Fig. 7 shows the price bubbles that form and the increased price volatility during periods of noisy prices. When the noise is removed at t=4000, the market price quickly returns to the competitive equilibrium. To implement this noise, one can schedule an event at the end of each trading period that uses the `setClearingPrice()` attributor to manually update the clearing price before the next period. These changes require roughly 5-6 lines of schedule code and another 1-2 lines to generate the noisy prices.

4.5 Leisure Time

In the original market model, workers were rather unrealistically forced to produce gold or food during every period. What if workers could instead choose a third option called "leisure" so that no commodities are produced? O'Callaghan and Steiglitz [5] describes this scenario in detail, and his simulations suggest that the leisure option raises market prices a factor above the non-leisure competitive equilibrium price. This result seems intuitive because a leisure option should reduce the long-term supply of food in the economy.

We implemented a `LeisurelyWorker` class that differs from the `SimpleWorker` class in that the leisurely worker can refrain from all action for one period. The leisurely worker chooses to be leisurely if his total wealth in food and gold exceeds his desired food inventory level by a sufficient factor. The worker still consumes a unit of food every period, but in choosing to be leisurely, he neither produces food nor gold and does not attempt to trade his inventories. As in the other models, this added functionality is straightforward to implement; it requires subclassing `WorkerBase` and filling the required methods with around 100 lines of logic. The customized `LeisurelyWorker` class can then be integrated into any of our models.

While using our test applications to simulate the leisure option, we discovered a surprising and potentially original result. In following the conclusions by Steiglitz *et al.* [8], the work in [5] assumed that value traders were necessary to stabilize the market prices to some equilibrium level. Consequently, the simulations described in this work all incorporate value and trend traders from the first simulation step.

However, the output from our simulation suggests that a market containing only leisure-oriented workers is sufficient to stabilize market prices. In Fig 8, we can see that the stability of the market price in an economy with leisurely workers only is similar to that in an economy with value traders. While the volatility in the worker-only market appears high relative to the mixed market, the higher volatility price is still *stable*. In simulations with only normal workers, this is not the case.

Although these results require further study, we speculate that the leisure option acts to stabilize prices in the following manner: As the number of non-working agents increases, the supply of food decreases and demand for food increases. This raises prices to a point in which almost all non-working agents are tempted back to production because of the potential trading profits from selling surplus food at the high prices. As more agents participate in farming, trading their surplus and gaining wealth, this depresses the price of food to a point where taking time off has higher value than farming/trading more food. The graphs in Appendix B are evidence to support this hypothesis. We can compare the change in how workers

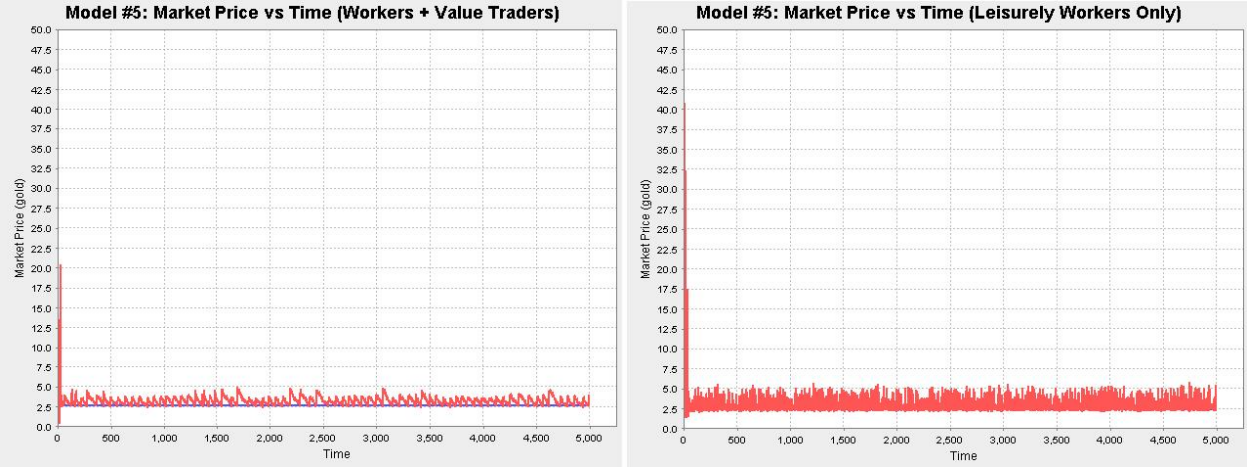

Figure 8: *The market price graph on the right shows a market with 25 leisurely workers and 25 value traders, while the graph on the right shows a market with 50 leisurely workers but no value traders.*

spend their time (Figure B.1) with the movement of market prices (Figure B.2) to see that rising prices indeed correspond to more farmers while declining prices lead to more leisure.

4.6 International Trade

Our final application confirms the results by Cohen and Steiglitz [1]. Here, the computational market model is extended to include "international" trade between two distinct economies (or countries), each with its own agents and market clearing mechanism. The first economy, which we call the "farming" economy, consists of workers that are more skilled in farming food, while the second economy, which we call the "mining" economy, consists of workers that are more skilled in mining for gold. Both economies contain value traders that stabilize the long-run market price to the competitive levels in each market. Then, [1] introduces an international trader (`GlobalTrader`), which utilizes value-based trading strategies to exploit the arbitrage opportunities between the two countries. Essentially, the international trader enforces the Law of One Price by buying cheap food in the farming economy and selling it for a profit in the mining economy. Agents in each economy quickly adjust to the market pressure created by the international trader. As a result, the long-term price of food in both countries reaches a *global competitive equilibrium price*, which is the intersection of supply and demand when agents in both economies are considered.

The international trade model is a great assessment of the robustness of the library implementations, because it tests whether the libraries facilitate the encapsulation of an entire market model into a single "country" component. As described in Section 3.1.2, the MASON toolkit's scheduler design inhibits such modularity. The Repast toolkit makes no such demands; thus, the library based on that toolkit is much more appropriate for this hierarchical model construction.

See Appendix C for details about our implementation of the international trade model.

5 Conclusions

We believe that our Java library for agent-based economic simulation, which is based on the computational market model in Steiglitz *et al.* [8] and written on top of the Repast simulation toolkit, is a viable simulation tool for microeconomic research. The library features clean interfaces that define the interactions between agents and markets, and highly customizable components for constructing an

economy. We plan to utilize this library to pursue further research in the area of microeconomic simulation.

6 Acknowledgements

We thank Prof. Ken Steiglitz for providing the guidance that made this project possible.

References

1. L. Cohen and K. Steiglitz, “A Computational Market Model Based on Individual Action: An Application to International Trade,” A.B. thesis, Princeton University, 1993.
2. J. R. Lenormand and K. Steiglitz, “Microsimulation of Markets and the Addition of Delay, Noise and Narration,” Junior independent work, Princeton University, 1997.
3. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, “MASON: A Multi-Agent Simulation Environment,” *Simulation*, vol. 81 (7), pp. 517-527, 2005.
4. M. J. North, N. T. Collier, and J. R. Vos, "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit," *ACM Transactions on Modeling and Computer Simulation*, vol. 16 (1), pp. 1-25, 2006.
5. L. I. O’Callaghan and K. Steiglitz, “Problems in Economic Modeling,” Junior Independent Work, Princeton University, 1996.
6. S. F. Railsback, S. L. Lytinen, and S. K. Jackson, “Agent-based Simulation Platforms: Review and Development Recommendations,” *Simulation*, vol. 82 (9), pp. 609-623, 2006.
7. S. Simdyankin and P. Madjidi, “Artificial Microeconomy Simulation Program,” Artificial Economy Project, 1996, <http://www.pdc.kth.se/~payam/Simulation.html> (20 December 2007).
8. K. Steiglitz, M. L. Honig, and L. M. Cohen, “A Computational Market Model Based on Individual Action,” in: S. Clearwater (Ed.), *Market-Based Control: A Paradigm for Distributed Resource Allocation*, Hong Kong: World Scientific, 1996.
9. K. Steiglitz and L. I. O’Callaghan, “Microsimulation of Markets and Endogenous Price Bubbles,” Third Int. Conf. on Computing in Economics and Finance, June 30 - July 2, 1997, Stanford, CA.
10. K. Steiglitz and D. Shapiro, “Simulating the Madness of Crowds: Price Bubbles in an Auction-Mediated Robot Market,” *Computational Economics*, vol. 12, pp. 35-59, 1998.
11. R. Tobias and C. Hofmann, “Evaluation of Free Java-libraries for Social-Scientific Agent Based Simulation,” *Journal of Artificial Societies and Social Simulation*, vol. 10 (1), 2004.

Honor Code

This paper represents my own work in accordance with University regulations.

- Christopher Chan

Appendix A: Screenshots

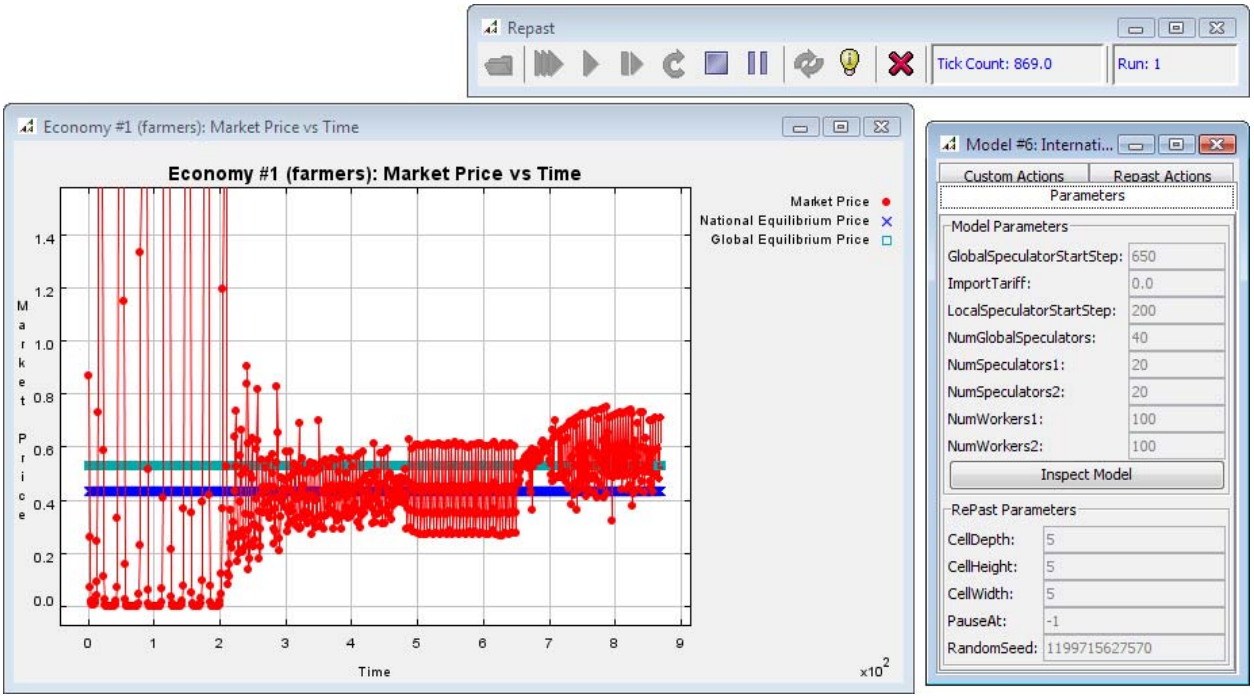

Figure A.1: Screenshot of the GUI for an application utilizing our Repast-based economics library. The library provides both the visualization (via Repast) and the agent-based market model.

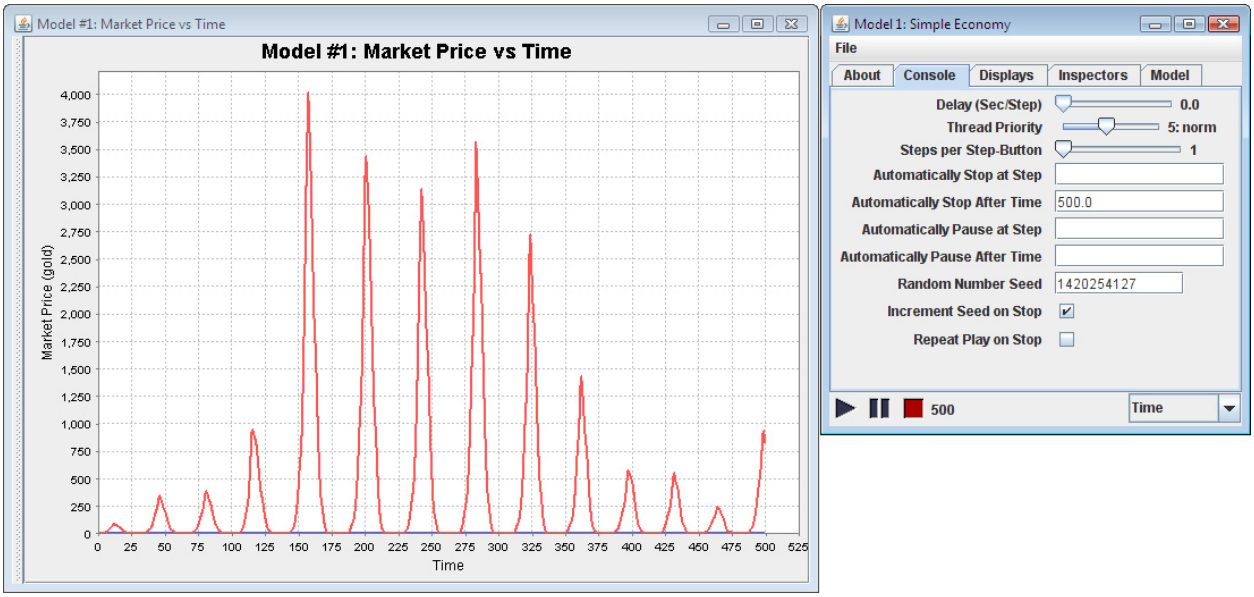

Figure A.2: Screenshot of an example application utilizing the MASON-based economics library.

Appendix B: Leisure in the Economy

Figure B.1: This graph displays the percentage of workers pursuing leisure, farming and mining during a span of 20 time steps in an economy with only leisurely workers. Workers appear to switch between leisure and farming.

Figure B.2: This graph shows the movement of the market price in a span of 20 time steps in an economy with only leisurely workers.

Appendix C: International Trade

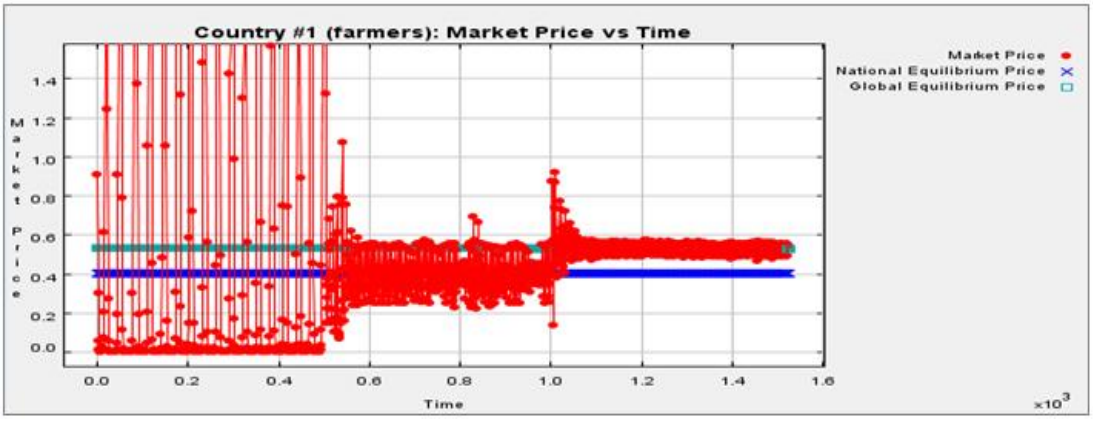

Figure C.1: Market price of food in a farming-centric economy. Value traders are introduced at time t=500 and international traders are introduced at t=1000. The arbitrage by the international traders pushes prices in this economy to the global competitive equilibrium price.

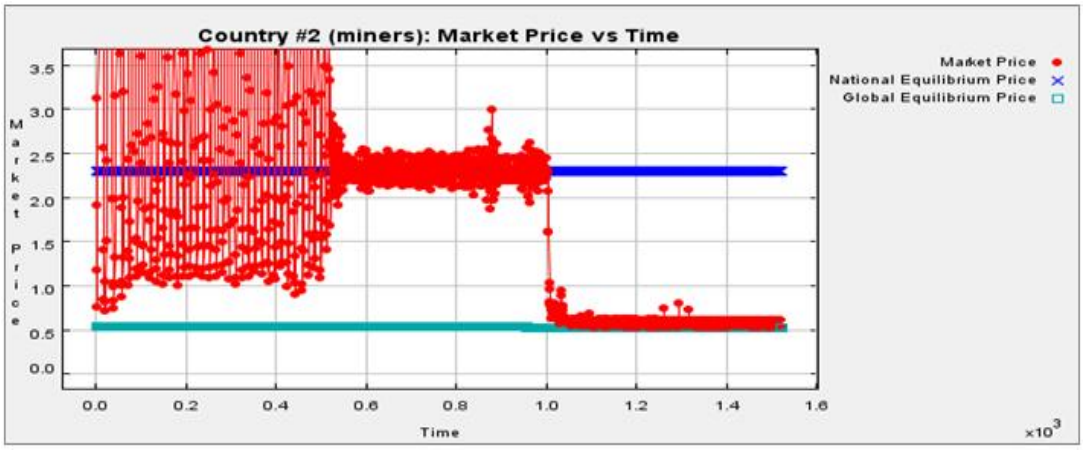

Figure C.2: Market price of food in a mining-centric economy. Value traders are introduced at time t=500 and international traders are introduced at t=1000. The arbitrage by the international traders lowers prices in this economy to the global competitive equilibrium price.