Replicating a Commodity Market Simulation Using Agent-Based Modeling Techniques

Linkan Bian
Michael Sabor
Matthew St. Peter
Paul Speaker

December 17, 2007

Abstract The ability to reasonably forecast price movements within the chemical market is of great importance to many departments within the Dow Chemical Company. This project is the first phase of two in the development of a pricing model to perform such forecasting. The focus of this phase is to apply the agent-based modeling paradigm to simulate a theoretical commodity market found in John D. Sterman's "Business Dynamics: Systems Thinking and Modeling for a Complex World." Sterman provides a price model based on the system dynamics paradigm that assumes a group of companies composing a market all behave alike and thus can be modeled as a single unit. The agent-based model is written in Java and based on the framework provided by the Repast toolkit.

Work done in partial fulfillment of the requirements of Michigan State University MTH 890; advised by Dr. Arthur Kordon and Tim Rey, The Dow Chemical Company, and Dr. Gabor Francsics, Michigan State University.

Table of Contents

Introduction 1

Agent-Based Modeling vs. System Dynamics 1

Sterman Model 3

Agent-Based Model Description 4

Prototyping 7

Repast 9

Conclusion 13

Future Work 14

References 15

Appendix 1 – DowModel Source Code 16

Appendix 2 – Producer Source Code 26

Appendix 3 – Consumer Source Code 30

Appendix 5 – Trade Engine Source Code 34

Introduction

The Dow Chemical Company is the second largest chemical manufacturer in the world with over 42,000 employees worldwide and has reported sales of $49.124 billion USD for 2006. According to Dow CEO Andrew Liveris, it is the mission of the company “to constantly improve what is essential to human progress by mastering science and technology” in order to become the largest chemical company in the world. With this mission in mind, the Dow Chemical Company explores and implements a variety of business intelligence techniques.

Dow Chemical has traditionally employed a system dynamics approach to forecasting business practices and is currently looking to explore the feasibility and strength of agent-based modeling applied to commodity markets. To that effect, a model detailed in John D. Sterman's “Business Dynamics: Systems Thinking and Modeling for a Complex World” that is well understood using a system dynamics approach been selected from as the test case to apply an agent-based approach. The model will hereafter be referred to as the Sterman model.

This project was started in order to determine the viability of simulating the theoretical scenario given in the Sterman model. More so than fit statistics, Dow Chemical is interested in a *qualitative* comparison: that is, when compared to the Sterman model, the agent-based model must possess all key constructs from Sterman and be able to replicate short- and long-term trends and ranges of market price.

Agent-Based Modeling vs. System Dynamics

Agent-based modeling and system dynamics are both designed to simulate some observable variable. Observables are measurable characteristics of interest. They may be associated either with separate individuals or with the collection of individuals as a whole. In general, the values of these observables change over time. In both kinds of models, these observables are represented as variables that take on assignments.

System dynamics begins with a set of equations that express the relationships between observable variables. The evaluation of these equations produces the evolution of the observables over time. System dynamics commonly uses ordinary differential equations to simulate variability over time. A modeler may recognize that the variable of interest varies as a result of interlocking relationships between individuals, but those behaviors have no explicit representation in system dynamics. Instead, a collection of individuals is treated as whole and their cumulative behavior is modeled as flow. For instance, traffic analysis using system dynamics is comparable to modeling the flow of a liquid through a pipe with bends and stretches of varying temperatures.

Agent-based models begin with rule sets that describe the interaction between individuals. These behaviors may involve multiple individuals directly (consumer and producer) or indirectly through a shared environment (consumer and consumer bidding). The modeler begins by representing the behaviors of each individual then turns them loose to interact. Direct relationships among the observables are an output of the process, not its input. In terms of the

traffic example, the modeler would assign logic to each individual car and then observe the model at many steps in order to fine-tune the logic and strengthen the model. Following Macal and North's "Tutorial on Agent-Based Modeling and Simulation," an agent will be defined as having the following properties:

- An agent is a discrete, identifiable individual with a set of characteristics and rules governing its behaviors and decision-making capability. The discreteness requirement implies that an agent has a boundary and one can easily determine whether something is part of an agent, not part of an agent, or is a shared characteristic. Above all, agents are self-contained.

- An agent is situated, living in an environment with which it interacts along with other agents. Agents have protocols for interaction with other agents, such as for communication, and the capability to respond to the environment. Agents have the ability to recognize and distinguish the traits of other agents.

- An agent may be goal-directed, having goals to achieve (not necessarily objectives to maximize) with respect to its behaviors. This allows an agent to compare the outcome of its behavior relative to its goals.

- An agent is autonomous and self-directed. An agent can function independently in its environment and in its dealings with other agents, at least over a limited range of situations that are of interest.

- An agent is flexible, having the ability to learn and adapt its behaviors based on experience. This requires some form of memory. An agent may have rules that modify its behaviors.

Agent-based models can produce effects that are not present in a continuous model. An example of this appears in Shnerb, who models two interacting populations. Both populations exhibit diffusion in space, while one also experiences a birth-death process. The paper compared the results of both a continuous model and a discrete model. The results demonstrated that under certain conditions, the continuous model predicted death for one of the populations, while the discrete model predicted the emergence of sustained populations. Since the birth-death process is fundamentally discrete, the conclusion is that a continuous model misses important characteristics that are present in a more realistic model. Therefore, it is worth investigating whether the system dynamics approach to modeling commodity markets might miss features necessary for accurate modeling.

Also, agent-based models better allow the use of proactive decision-making. In the Sterman model, the relationship between the model's parameters and the decisions made by a firm are very indirect. In contrast, the decisions made by firms are the parameters in an agent-based model. In this way, a firm may be more able to accurately simulate the results of a decision in an agent-based model, as compared to a systems dynamics model.

Sterman Model

Before attempting to replicate the test model, it must be analyzed in order to extract the relevant constructs. There are four components to the model: capacity, production, demand, and price. The following figure shows how the four pillars of the Sterman model are related. Capacity allows for production, while shrinking inventory might require the building of new capacity. Similarly, prices for a commodity might indicate that new capacity will be profitable. Capacity is needed for production, while the production rate will influence the decision of whether to replace capacity. Finally, the relationships between production, demand, and price come from the law of supply and demand.

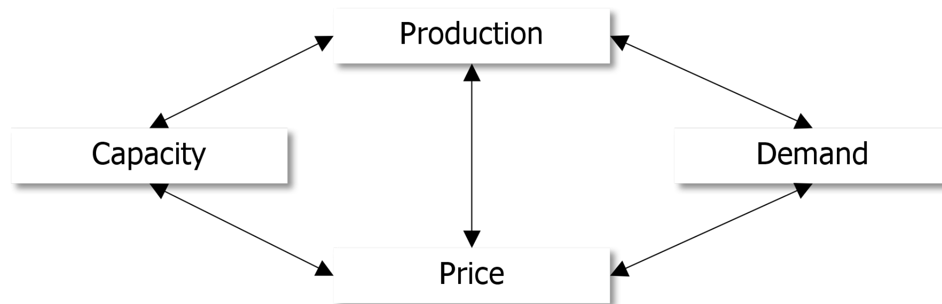

Figure 1. Box diagram of Sterman model for commodity markets.

However, this big picture obscures some of the finer details present in Sterman model. Therefore, it is essential to look at each pillar above in detail in order to get a true picture of commodity markets.

Several parameters are essential in examining production. For example, the inventory level must be accounted for when determining new production. Since the number of orders processed depends on the inventory on hand, there must be some control to ensure that there is not an unnecessary buildup of stock.

Similarly, several additional parameters are needed when investigating capacity. Capacity is defined as the maximum output generated by the system. This amount is directly proportional to the capital stock, which is a measure of the number of plants and their size. This capital stock in turn depends on how quickly the capital stock can be built and the lifetimes of the capital stock.

The system-wide industry demand depends on the price at which the commodity is supplied. The Sterman model uses a very simple relationship between demand and price: a linear function with cutoffs at some maximum and minimum value. The cutoffs are placed as a first-order control, ensuring that demand does not expand further than is possible or fall below zero.

Price depends most fundamentally on the laws of supply and demand. More precisely, the price depends on the ability of perceived inventory to cover orders. If the inventory is perceived to be less than demand requires, a shortage ensues, and prices are expected to increase.

This ability of inventory to cover orders is called "perceived inventory coverage (*PIC*)," and the price, *P*, will then be inversely proportional to *PIC*. That is,

$$P \propto PIC^{\alpha}.$$

where $\alpha < 0$. The price also depends on the expected variable cost of production, but this dependence will be weaker than the dependence on *PIC*. This dependence is modeled to be linear, with a small constant of proportionality.

The essential dependencies of the Sterman model must be replicated within the agent-based framework. The following description details how this is achieved.

Agent-Based Model Description

Constructing an agent-based model begins with populating it with a desired number of agents. This first step immediately differentiates the paradigms: the Sterman model never creates distinct agents. The commodity model requires two distinct types of agent: producers, who supply the commodity to be modeled, and consumers, who purchase the commodity.

Since each class of agent performs a different function within the model, each class is initialized with its own set of attributes. They are then operated on in different fashions. Table 1 below outlines the attributes that are assigned to each. Notice that the producers carry many more attributes than the consumers, as there are more actions for the producer to consider.

Table 1. Attributes specific to each agent.

Producer Attributes	Consumer Attributes
Inventory	Demand quantity
Capacity	Demand price
Supply price	Demand capacity
Supply quantity	Price history weights
Price history weights	
Production Characteristics	
Utilization ratio	
Production cost	
Profit slope	
Loss slope	
Break-even point	

In order to differentiate the members within each class of agent, initial values for attributes are selected from a uniform random distribution generated by the user. Thus the agents

within a particular class act similarly, but are not identical. Also, consumers are assumed to have a demand that grows at a natural rate.

Once the agents have been constructed and their initial parameters have been set, the model enters its main phase: a trading round, followed by parameter modification.

The trading algorithm matches consumers and producers based on price. At each time step, the consumer list and the producer list are randomly sorted immediately before the trade round. The algorithm then iterates down the consumer list and searches for trades. A trade occurs when the following conditions are met: both consumer and producer have nonzero quantity to trade, and the producer's trade price is below the consumer's trade price. The process is iterated through all consumers until there is no longer supply to trade at an acceptable price. After the trade round is completed, the market price is computed.

By its nature, a system dynamics model does not need a method to compute a market price: since all consumers and producers are lumped together, there is no need to do so. However, in an agent-based model there are many local interactions that must be accounted for when a market price is determined. What each producer and each consumer considers its individual trading price is never the overall market price.

Each individual producer has a fraction of the total supply weighted at its specific price. The market price is then the sum of all of the fractional supplies multiplied by their respective weights, divided by total supply. Thus, the market price is calculated at each time step after the trade round as a simple weighted average given by

$$\text{Market Price} = \frac{\sum\limits_{\text{Producers}} \text{Supply Traded}_{\text{P}} \times \text{Supply Price}_{\text{P}}}{\text{Total Supply Traded}}.$$

Once the market price has been calculated, agents must update their behaviors for next round. This is one of the defining features of an agent-based model: that each agent is capable of evaluating outcomes from the trade round in order to update their attributes for the next trade round. Note that the order in which attributes are updated does not matter, since the only global information that each agent utilizes is the market price calculated after trading. Both consumers and producers update their respective price and quantity demands, and producers also decide whether or not to add capacity.

The price at which a consumer demands goods for the next round is determined by multiple factors. Since demand is assumed to grow from the initial demand at every time step, the current "natural" demand is calculated by

$$\text{Natural Demand}_{\text{Current}} = (1.008)^{\text{Time}} \times \text{Initial Demand}.$$

Once the natural demand for the current time step is calculated, the demand price for the next round is computed using the following: demand that is left over from the last round of trading, the price elasticity of demand, and a short history of previous price demands. First, a consumer's sensitivity to short-term price fluctuations is calculated by

$$\text{Sensitivity} = \left(\frac{\text{Natural Demand}_{\text{Current}} + \text{Demand}_{\text{Leftover}}}{\text{Natural Demand}_{\text{Current}}} \right)^{\text{Elasticity}},$$

then an ideal demand price (irrespective of sensitivity) is calculated by

$$\text{Demand Price}_{\text{Ideal}} = \omega_{\text{Market}} \times \text{Market Price} + \sum_{\text{i}=0}^{3} \omega_{\text{i}} \times \text{Demand Price}_{\text{n-i}},$$

where the ω's are weights that add to unity. The weight ω_{Market} is chosen to be small, so that the consumers do not converge to equilibrium too soon. This will be seen again in the next section.

Finally, the demand price attribute is modified for the next trading round by multiplying the ideal demand price by the sensitivity to short-term price fluctuations. That is,

$$\text{Demand Price}_{\text{n+1}} = \text{Sensitivity} \times \text{Demand Price}_{\text{Ideal}}.$$

In order to update the quantity demanded by each consumer for the next trade round, the natural demand computed above is utilized again. Unmet demand is assumed to carry over from the previous time step, another feature that the Sterman model does not consider. The quantity demanded is determined as a ratio of the natural demand that is calculated at every time step. The ratio is calculated as

$$\text{Ratio} = \frac{\text{Demand Price}_{\text{Current}} - \text{Demand Price}_{\text{Last}}}{\text{Demand Price}_{\text{Current}}},$$

and is then multiplied by the natural demand to determine the new demand added to leftover demand from the previous time step. Thus,

$$\text{Demand}_{\text{n+1}} = \text{Natural Demand} \times \text{Ratio} + \text{Demand}_{\text{Leftover}}.$$

Similar to consumers, each producer determines a desired supply price, which is then tempered using a history of previous supply prices and the most current market price. The desired supply price is based on inventory remaining after the trade step as compared to capacity and the price elasticity of supply. It is calculated by

$$\text{Supply Price}_{\text{Ideal}} = \left(\frac{\text{Capacity}_{\text{Current}}}{\text{Posttrade Inventory}_{\text{Current}}} \right)^{\text{Elasticity}}.$$

Once the desired price is set, it is then applied to the formula

$$\text{Supply Price}_{\text{n+1}} = \omega_{\text{Ideal}} \times \text{Supply Price}_{\text{Ideal}} + \omega_{\text{Market}} \times \text{Market Price} + \sum_{\text{i}=0}^{3} \omega_{\text{i}} \times \text{Supply Price}_{\text{n-i}}$$

in order to determine the supply price for the next trade round.

The market price as given above will be heavily determined by the supply prices that each producer computes, since all trades occur at supply prices. In turn, supply price is determined by inventory, which further determined by capacity. Thus, a long-term period should appear in the market price based on total capacity within the model. This is one of the defining features of the Sterman model.

Before determining whether to add capacity, however, supply production for the next trade round must be determined. Production is first determined as a ratio of total capacity available to each producer, and then multiplied over capacity to determine final production. Once new production is determined it is added to any remaining inventory before the next trade round.

The ratio of capacity utilized by each producer is given as a piecewise function that depends on the profitability of new production. Thus, if the cost of production is above the market price, production is sharply curtailed until it is again profitable to produce new goods. The ratio is calculated using an if-then clause:

$$\begin{aligned}&\text{if Market Price}_{\text{Current}} > \text{Cost}\\ &\quad \text{Ratio} = \text{Slope}_{\text{Profitable}}\left(\frac{\text{Price}_{\text{Current}} - \text{Cost}}{\text{Cost}} + \text{Break Even Point}\right);\\ &\text{if Market Price}_{\text{Current}} < \text{Cost}\\ &\quad \text{Ratio} = \text{Slope}_{\text{Loss}}\left(\frac{\text{Price}_{\text{Current}} - \text{Cost}}{\text{Cost}} + \text{Break Even Point}\right).\end{aligned}$$

Once the ratio is set, new production is determined, and inventory is calculated:

$$\begin{aligned}&\text{New Production} = \text{Ratio} \cdot \text{Capacity}_{\text{Current}}\\ &\text{Inventory}_{n+1} = \text{Inventory}_{n} + \text{New Production}.\end{aligned}$$

Finally, producers require a decision method in order to build new capacity. Currently, the decision method is based on capacity utilization ratio determined above. Once the ratio rises above a threshold value set by the user (typically about 0.96) for 6 successive rounds, a capacity delay timer is engaged. Once the timer expires, capacity is added.

Prototyping

An agent-based model has sufficient complexity to warrant a prototype, allowing the modeler to test scenarios before scaling up. Even a commercial spreadsheet application such as Microsoft Excel has sufficient power to construct a (very simple) prototype. Developers may also choose to implement a prototype in a more powerful computer manipulation system.

A prototype model is constructed in MATLAB, available from The MathWorks. This has two advantages: first, it is more powerful than a simple spreadsheet, and second, MATLAB possesses a code structure that allows for a reasonable port to Repast. Figure 2 below shows the pseudo-code outline in MATLAB. Each of the function implements the methods described in the section above.

```
build_consumer()
build_producer()

for t=1:maxtime

        trade()                %% trade available supply and
                               %% available demand

        m_price()              %% compute the market price based on
                               %% trading just completed

        price_adjust()         %% producers and consumers adjust their
                               %% individual prices

        demand()               %% consumers calculate demand based on
                               %% new adjusted prices

        production()           %% each producer adjusts production
                               %% according to previous data

        capacity()             %% each producer possibly adds capacity

end
```

Figure 2. Pseudo-code of MATLAB script to run agent-based model.

Some, but not all of the critical constructs within the Sterman model are replicated in the prototype. In particular, the decision to add capacity is not implemented; rather, capacity is added in a fashion that allows for immediate comparison between capacity and market price. Figure 3 below shows that one of the most critical features of the Sterman model, the fluctuation of market price with respect to capacity, is replicated, even without a decision process.

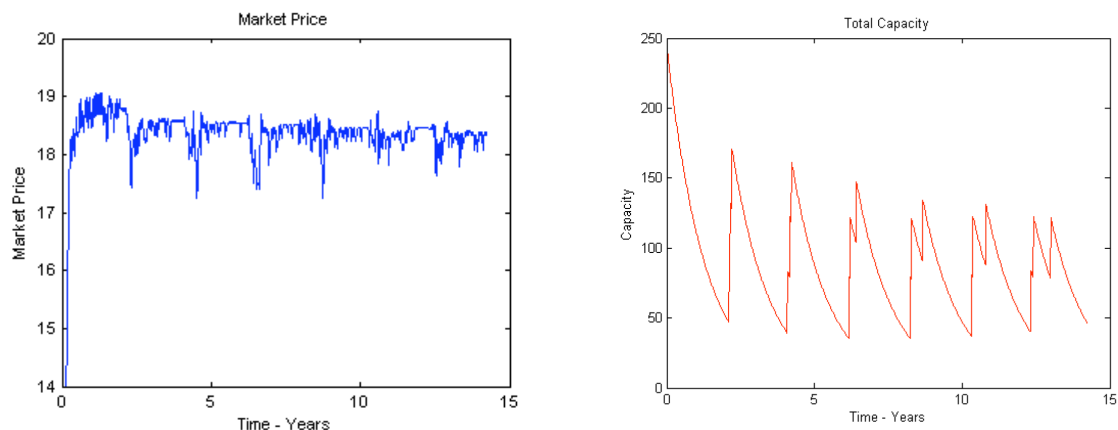

Figure 3. Market Price vs. Total Capacity.

Another feature that should be replicated by an agent-based model is a far-off convergence to equilibrium. This convergence is achieved through the consumer's price adjustment routine. Recall that ω_{Market} as defined above is chosen to be small, in order to push the equilibrium point into the far future. Choosing the market weight too large creates undesirable results.

Figure 4 below shows a fluctuating market price and fifteen consumers adjusting to the market price. On the left, the current market price affects the next demand price between 0.05% and 0.15% (that is, ω_{Market} is distributed between 0.0005 and 0.0015). On the right, the weight varies between 0.5% and 1.5%. Relative to the left graph, the right graph represents a high market price weight and has undesirable properties. Though the market price has the correct long-period characteristics, the downward spikes are both too severe and too brief. Equilibrium is reached before any interesting behavior can be observed.

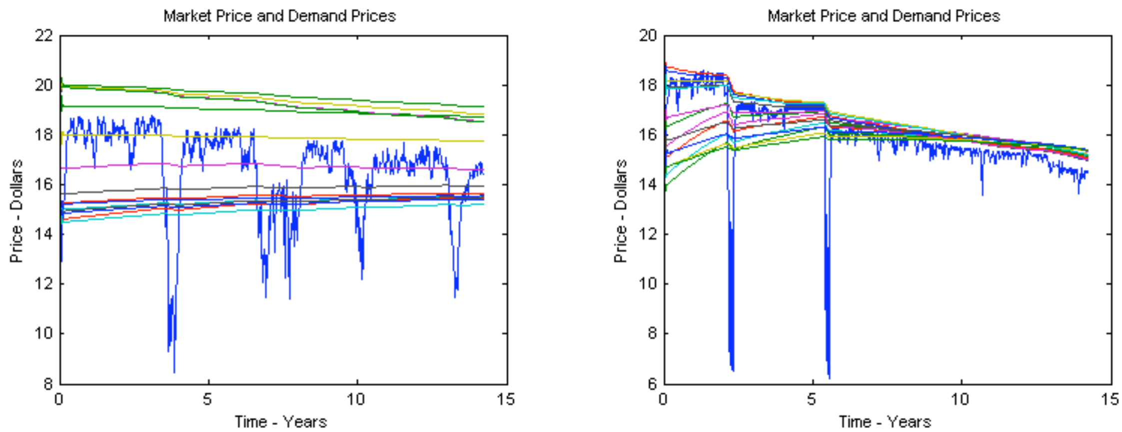

Figure 4. Comparison of market price weights when determining next demand price.

The MATLAB prototype shows that, on a small scale, the important characteristics of the Sterman model can be replicated. However, MATLAB does not scale well to large environments: there are too many interwoven function calls and the code becomes unreasonably complex to maintain. Thus, when developing a final model, a different programming language is used.

Repast

Once a reasonable prototype model is developed, it is extended to a more complex environment using the Recursive Porous Agent Simulation Toolkit, commonly known as Repast. Repast is a free, open-source toolkit designed for agent-based modeling originally developed by Sallach, Collier, Howe, North, and others at the University of Chicago. It is now managed by the non-profit volunteer Repast Organization for Architecture and Development (ROAD). ROAD is led by a board of directors that includes members from a wide range of government, academic and industrial organizations.

Repast has multiple implementations in several languages and contains mathematically advanced adaptive features such as genetic algorithms and regressions built into ready-to-use libraries. Also, Repast contains libraries and methods for the real-time display of graphs and data visualizations. For example, a modeler can harvest existing code designed to display a real time graph of time-series data rather than spending time coding it from the ground up, allowing the modeler to spend more time on the specifics of the model.

The Java programming language has been chosen for this implementation, due to its wide adoption rate and cross-platform compatibility. In order to efficiently write the Java underpinning to the Repast libraries, the free, open-source Java Integrated Development Environment (IDE) Eclipse is used. In order to smoothly use Repast with Eclipse, careful detail must be taken initially to properly import the existing Repast libraries and source code. After correctly setting up the work environment, the modeler can easily test and run. Using Eclipse also allows for easy manipulation of existing Repast demonstrations. By changing existing models, the modeler is able to rapidly learn the inner workings of Repast packages and encourages the harvesting of code to meet specific needs.

Since the prototype model was constructed using a structured programming language, implementing Repast entails modifying the syntax of the MATLAB code while leaving the overall "feel" of the model intact. More specifically, the code that determines decisions remains the same while the code designed for scheduling, building, and manipulating the model will take on the structure provided by Repast rather than the one designed in MATLAB.

The simple commodity model implemented here uses libraries from two packages: the `engine` package and the `analysis` package. The analysis package libraries are for graphing results of the model, while the engine package libraries provide most of the framework for running the model. Of particular importance is the `SimModelImpl` class in the engine package, the class from which the commodity model (and indeed many a Repast model) is based on. Utilizing these models is simple a matter of importing them into a new project.

```
import uchicago.src.sim.analysis.DataSource;
import uchicago.src.sim.analysis.OpenSequenceGraph;
import uchicago.src.sim.analysis.Sequence;
import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.engine.SimInit;
import uchicago.src.sim.engine.SimModelImpl;

public class DowModel extends SimModelImpl {
    // Everything to create, display and run the model.
}
```

Figure 5. Source code. Repast package libraries are imported to run and display the model.

Once a model has been declared with `SimModelImpl`, it is populated with agents. The initial parameters for the agents can be modified using a GUI parameter box that is constructed using the method `getInitParams` that is built into `SimModelImpl`. The Repast engine requires that the parameters defined using `getInitParams` have their own get and set methods to change these values.

```
public String[] getInitParam(){
    String[] initParams = {"Consumer_Amount", "Producer_Amount", "Price",
                                 "Consumer_Demand", Consumer_Demand_Width",
                                 "Producer_Supply", "Producer_Supply_Width",
                                 "Producer_Capacity", "Producer_Capacity_Width",
                                 "Cost", "Slope_Loss", "Slope_Break", "Slope_Profit",
                                 "Utilization_Ratio"};
            return initParams;
    }

    public int getProducer_Amount(){
            return numProducers;
    }

    public void setProducer_Amount(int prods){
            numProducers = prods;
    }
}
```

Figure 6. Source code. Strings named in `getInitParams()` must have matching get and set methods.

Parameters | Custom Actions

Model Parameters

Parameter	Value
Consumer_Amount:	1000
Consumer_Demand:	0.6
Consumer_Demand_Width:	0.1
Cost:	12.50
Price:	15.0
Producer_Amount:	20
Producer_Capacity:	60.0
Producer_Capacity_Width:	10.0
Producer_Supply:	100.0
Producer_Supply_Width:	15.0
Slope_Break:	0.2
Slope_Loss:	0.3
Slope_Profit:	1.0
Utilization_Ratio:	0.9

Inspect Model

Figure 7. Parameter box displayed to the user. Parameters match strings written in `getInitParams()`.

The populated agents are then subject to the model's schedule. Again, the scheduling framework is provided by Repast and is then extended to suit the needs of the model. Extending the `BasicAction` class imported from the engine package provides the bulk of the modifications

specific to the model. The `BasicAction` class details the actions the model takes at every tick through it's `execute` method.

The "tick" is, as Collier states, the "quantum unit of time" in Repast. The tick itself provides an ordering of events with relation to each other only. It is up to the modeler to impart some amount of meaning to the time length that a tick represents. For the purposes of this model, each tick represents one week of time.

Below, Figure 8 details the steps taken at every step. It is apparent that the steps taken here are fundamentally the same as those taken in the MATLAB prototype. The graphs generated from the process are now generated dynamically, at every time step, as opposed to statically once the simulation is finished. This also allows from open-ended simulations that can be stopped by the user at his leisure.

```
Public void buildSchedule(){

        class DowModelStep extends BasicAction{
                // Needed by Repast
                public void execute(){
                        trade();              // Conduct trades for the current time
                        setMarketPrice();     // Calculate and set the market price
                        graph.step();         // Updates the graph
                        consumerList.update();      // Updates all consumers
                        producerList.update();      // Updates all producers
                }
                schedule.scheduleActionBeginning(0, new DowModelStep());
        }
}
```

Figure 8. Schedule source code.

Once the code has been compiled and executed, the model is controlled via the Repast toolbar, shown in Figure 9 and the parameter box shown above. To run the model, a user inputs parameter values and hits the 'Run' button. If a user is interested in closely inspecting the step-by-step changes in the model, the 'Single Step' button may be used to advance the model in a slow, controlled fashion.

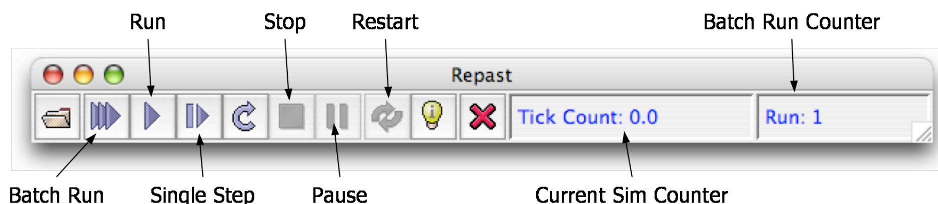

Figure 9. Screenshot of the Repast toolbar.

One of Repast's main advantages is its dynamic updating of graphs, which allows the user to see, in real time, changes in the system. Figure 10 contains screen captures of the final Repast output. The upper graph shows the market price updated at every tick. The lower graph shows two graphs superimposed upon each other: the red line represents total capacity within the system, while the blue line represents the total amount of capacity that is actually utilized.

The periodicity is apparent: as the systemwide capacity utilization approaches 100%, supply can not keep up with demand, and the market price spikes upward. Simultaneously, capacity is added in order to meet the high demand levels. A surplus ensues, driving the market price back down to normal levels, exactly as the Sterman model predicts. Thus is it shown that the Sterman model can be replicated in an agent-based setting.

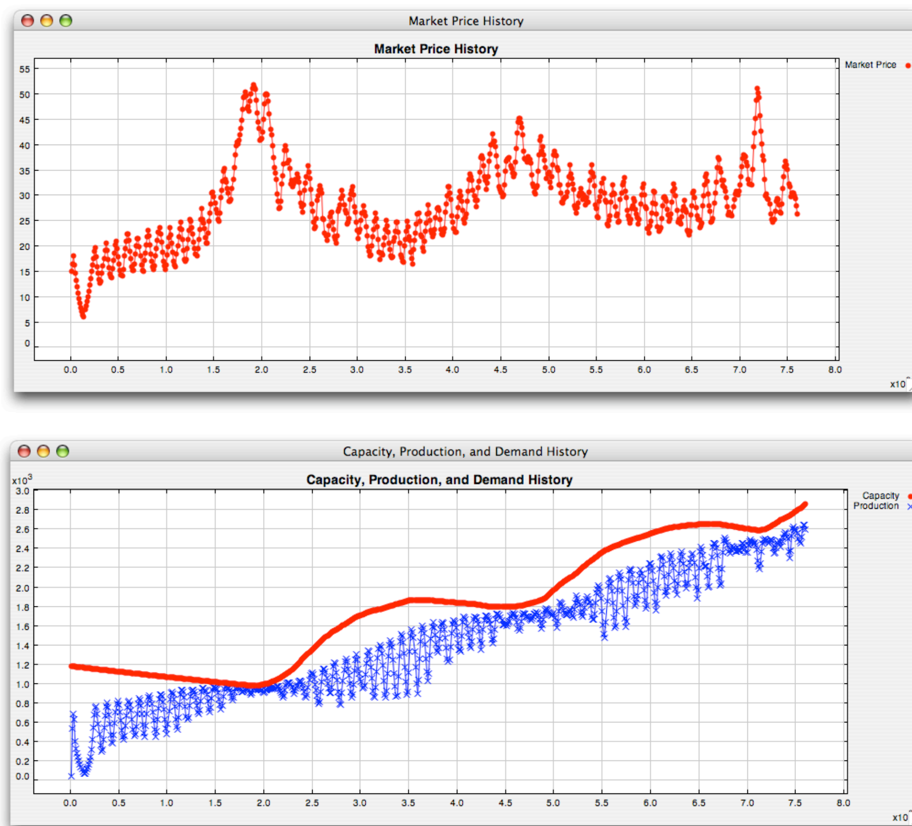

Figure 10. Screenshot of Repast output drawn in real time.

Conclusion

The success of implementing an agent-based model to simulate the theoretical commodity market found in John D. Sterman's book "Business Dynamics: Systems Thinking and Modeling for a Complex World" suggests that the application of this modeling paradigm onto real world scenarios is worth further investigation, to be detailed below.

Future Work

This project will continue by taking the model developed and modifying it to represent the specific supply/demand markets of chemicals used by The Dow Chemical Company. The final deliverable will be an Agent Based Model created using Repast software that represents real Dow data.

There are still many features of the Repast framework that may be implemented: for instance, instead of randomly sorting the consumer list before the trading round, small-world networks may be designed to create persistent trading partners. Such small-world networks better represent Dow's trading patterns in the commodity market. Libraries exist to rapidly implement such networking capabilities.

In a Dow-specific model, the trade algorithm must be reconsidered. For a given chemical commodity, Dow is both a buyer and a seller. In fact, most production is not sold on the open market, but rather utilized by Dow itself. The production aspect of a firm may also be treated in an agent-based setting. For example, a firm may have multiple agents whose sole function is to produce goods for trade. The firm itself collects the production from each of these agents before entering the trade round.

As of December 3, 2007, the release of Repast Simphony has integrated the Repast libraries into a point-and-click interface, further speeding the development of agent-based models. This new software will be utilized going forward.

References

Collier, Nick, Tom Howe and Michael North. "Onward and Upward: The Transition to Repast 2.0." North American Association for Computational Social and Organizational Science. Pittsburgh, 2003.

Govindu, Ramakrishna. Multi-Agent Systems for Supply Chain Modeling: Methodological Frameworks. Diss. Wayne State Univ, 2006.

Macal, Charles and Michael North. Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation. New York: Oxford University Press, 2007.

—. "Tutorial on Agent-Based Modeling and Simulation." Winter Simulation. Orlando, 2005.

Shnerb, Nadev. "The Importance of Being Discrete: Life Always Wins on the Surface." Proceedings of the National Academy of Sciences. 2000. 10322-10324.

Sterman, John D. Business Dynamics: Systems Thinking and Modeling for a Complex World. Boston: Irwin/McGraw-Hill, 2001.

Appendix 1 – DowModel Source Code

```
package dow.phase1;

//Import necessary Repast and Java classes.
import uchicago.src.sim.analysis.DataSource;
import uchicago.src.sim.analysis.OpenSequenceGraph;
import uchicago.src.sim.analysis.Sequence;
import uchicago.src.sim.engine.BasicAction;
import uchicago.src.sim.engine.Schedule;
import uchicago.src.sim.engine.SimInit;
import uchicago.src.sim.engine.SimModelImpl;
import java.util.ArrayList;
import java.util.Collections;

public class DowModel extends SimModelImpl {

        // Model variables with defaults from parameter box.
        private int numConsumers = 650;
        private int numProducers = 20;
        private double startPrice = 15;
        private double cDemand = 1;
        private double cDemandWidth = .01;
        private double pSupply = 2;
        private double pSupplyWidth = .02;
        private double pCapacity = 60;
        private double pCapacityWidth = 15;
        private double pCapacityAdd = 1;
        private double pCost = 8;
        private double slopeLoss = .3;
        private double slopeProfit = 1;
        private double slopeBreak = .2;
        private double utilRatio = .9;
        private double demandGrowth = .16;

        // Schedule object: this is required by Repast.
        private Schedule schedule;

        // Lists that Contain agent objects.
        private ArrayList<Consumer> consumerList;
        private ArrayList<Producer> producerList;

        // List that contains trade objects.
        // These objects track all trades in a given step.
        private ArrayList<Trader> traderList;

        // Lists to store data for the sequence graphs.
        // These are market wide observations.
        private ArrayList<Double> priceList;
        private ArrayList<Double> capacityList;
        private ArrayList<Double> inventoryList;
        private ArrayList<Double> demandList;
        private ArrayList<Double> productionList;

        // Sequence graphs.
        private OpenSequenceGraph graph1;
        private OpenSequenceGraph graph2;
        private OpenSequenceGraph graph3;

        public String getName(){
```

```
        // Name of the Model.
        return "Dow Phase 1";
}

public void setup(){
        // This is another Repast inherited method.
        // Each time the setup button is pressed the
        // following is executed.

        // Create needed lists.
        consumerList = new ArrayList<Consumer>();
        producerList = new ArrayList<Producer>();
        traderList = new ArrayList<Trader>();
        priceList = new ArrayList<Double>();
        capacityList = new ArrayList<Double>();
        inventoryList = new ArrayList<Double>();
        productionList = new ArrayList<Double>();
        demandList = new ArrayList<Double>();

        // Create the schedule object.
        schedule = new Schedule(1);

        // Tear down old displays before building new ones.
        if (graph1 != null){
                graph1.dispose();
        }
        graph1 = null;

        if (graph2 != null){
                graph2.dispose();
        }
        graph2 = null;

        if (graph3 != null){
                graph3.dispose();
        }
        graph3 = null;

        // Create displays for the three sequence graphs.
        graph1 = new OpenSequenceGraph("Market Price History", this);
        graph2 = new OpenSequenceGraph("Capacity, Production, and Demand History", this);
        graph3 = new OpenSequenceGraph("Inventory History", this);

        // Register the above displays.
        this.registerMediaProducer("Plot", graph1);
        this.registerMediaProducer("Plot", graph2);
        this.registerMediaProducer("Plot", graph3);
}

public void begin(){
        // This method is executed when the start button is pushed.

        // Required Repast methods.
        // Rather than writing all the needed code here,
        // we call the following methods.  This allows
        // for a greater level of organization in the code.
        buildModel();
        buildSchedule();
        buildDisplay();

        // Create the graph windows.
        graph1.display();
```

```
		graph2.display();
		graph3.display();
	}

	public void buildModel(){
		// This method calls the constructors for
		// both types of agents.
		addNewConsumers();
		addNewProducers();
	}

	public void buildSchedule(){
		// This method creates an inner class that defines
		// the appropriate actions to be completed at each step.

		class DowModelStep extends BasicAction{
			// This inner class provides the code to be
			// executed at each step.

			public void execute(){
				// The following methods are unique to this project and
				// must be executed in order.

				// These methods gather market wide information
				// to be displayed at each step.
				setTotalCapacity();
				setTotalInventory();
				setTotalProduction();
				setTotalDemand();

				// This method conducts all the trading actions for a step.
				trade();

				// This method calculates and set the market price
				// for the current trading step.  This information is then
				// gathered to be displayed.
				setMarketPrice();

				// These methods update the respective graphs
				// and are provided by Repast.
				graph1.step();
				graph2.step();
				graph3.step();

				// This loop prepares all the consumers
				// for the next round of trading.
				for (int i = 0; i < consumerList.size(); i++){
					Consumer tempConsumer = consumerList.get(i);
					tempConsumer.step(getTime(), priceList.get(getTime()),
demandGrowth);
				}

				// This loop prepares all the producers
				// for the next round of trading.
				for (int i = 0; i < producerList.size(); i++){
					Producer tempProducer = producerList.get(i);
					tempProducer.step(getTime(), priceList.get(getTime()),
demandList.get(getTime()),
								numProducers, slopeLoss, slopeProfit,
slopeBreak, utilRatio, pCapacityAdd);
				}
			}
```

```
        }

        // Tells how often and when to execute the above code
        schedule.scheduleActionBeginning(0, new DowModelStep());
}

public void buildDisplay(){
        // This method calls Repast methods in order to display the graphs.
        graph1.addSequence("Market Price", new marketPriceHistory());
        graph2.addSequence("Capacity", new capacityHistory());
        graph3.addSequence("Demand", new demandHistory());
        graph2.addSequence("Production", new productionHistory());
        graph3.addSequence("Inventory", new inventoryHistory());
}

class marketPriceHistory implements DataSource, Sequence{
        // This inner class is needed for the corresponding sequence graph.
        // This fetches the data for the market sequence.

        public Object execute(){
                return new Double(getSValue());
        }
        public double getSValue(){
                return priceList.get(getTime());
        }
}

class capacityHistory implements DataSource, Sequence{
        // This inner class is needed for the corresponding sequence graph.
        // This fetches the data for the capacity sequence.

        public Object execute(){
                return new Double(getSValue());
        }
        public double getSValue(){
                return capacityList.get(getTime());
        }
}

class inventoryHistory implements DataSource, Sequence{
        // This inner class is needed for the corresponding sequence graph.
        // This fetches the data for the inventory sequence.

        public Object execute(){
                return new Double(getSValue());
        }
        public double getSValue(){
                return inventoryList.get(getTime());
        }
}

class productionHistory implements DataSource, Sequence{
        // This inner class is needed for the corresponding sequence graph.
        // This fetches the data for the production sequence.

        public Object execute(){
                return new Double(getSValue());
        }
        public double getSValue(){
                return productionList.get(getTime());
        }
}
```

```
	class demandHistory implements DataSource, Sequence{
		// This inner class is needed for the corresponding sequence graph.
		// This fetches the data for the demand sequence.

		public Object execute(){
			return new Double(getSValue());
		}
		public double getSValue(){
			return demandList.get(getTime());
		}
	}

	private void trade(){
		// Create a trade object for each trading round and add
		// the object to the traderList. Each trader object keeps
		// track of all the trades in a trading round.  A trade
		// consists of a quantity and a price; these are recorded in lists.
		// The indices correspond to a pair
		Trader trades = new Trader(getTime());
		traderList.add(trades);

		// Shuffle the consumer list.
		Collections.shuffle(consumerList);

		// This is the trade loop.  This iterates through
		// the shuffled consumer list.  For each consumer
		// the producer list is shuffled and the consumer
		// seeks to exhaust their demand for any price
		// lower than their expected price.
		for (int i = 0; i < consumerList.size(); i++){
			// Selects a particular consumer.
			Consumer tempConsumer = consumerList.get(i);

			// This shuffles the producer list.
			Collections.shuffle(producerList);

			// This loop iterates through the producer list while
			// the conumsumer's demand is still greater than zero.
			while (tempConsumer.getFindingTrade()){
				// This iterates through the producer list.
				for (int j = 0; j < producerList.size(); j++){
					Producer tempProducer = producerList.get(j);

					// This checks whether the producer's price is acceptable
					// and whether they still have inventory to sell.
					if((tempProducer.getTempQuantity(getTime()) > 0) &&
(tempConsumer.getPrice(getTime()) >= tempProducer.getPrice(getTime()))){
						// This method adds the trade (producer price) price
to Trader object.
						trades.nextPrice(tempProducer.getPrice(getTime()));

						// This keeps track of the last price a consumer
paid for goods.

	tempConsumer.setTradePrice(tempProducer.getPrice(getTime()));

						// This determines and adds the trade amount to the
Trader object.
						double consumerAmount =
tempConsumer.getTempQuantity(getTime());
```

```
						double producerAmount = 
tempProducer.getTempQuantity(getTime());
						double tradeAmount = Math.min(consumerAmount, 
producerAmount);
						trades.nextQuantity(tradeAmount);

						// These methods update the remaining quantities for 
both producers and consumers.
						tempConsumer.adjustTempQuantity(getTime(), 
consumerAmount - tradeAmount);
						tempProducer.adjustTempQuantity(getTime(), 
producerAmount - tradeAmount);
					}

					// This determines whether or not the consumer needs to
					// continue searching for another trade.
					if ((tempConsumer.getTempQuantity(getTime()) <= 0) || (j == 
producerList.size()-1)){
						tempConsumer.setFindingTrade(false);
						break;
					}
				}
			}
		}
	}

	private void addNewConsumers(){
		// Create consumers and add them to the consumer list.

		for (int i = 0; i < numConsumers; i++){
			Consumer a = new Consumer(i, startPrice, cDemand, cDemandWidth);
			consumerList.add(a);
		}
	}

	private void addNewProducers(){
		// Create producers and add them to the producer list

		for (int i = 0; i < numProducers; i++){
			Producer a = new Producer(i, startPrice, pSupply, pSupplyWidth,
					pCapacity, pCost, pCapacityWidth);
			producerList.add(a);
		}
	}

	private void setMarketPrice(){
		// Calculate the Market Price for the current time step.

		double numer = 0;
		double denom = 0;
		Trader trades = traderList.get(getTime());
		int numTrades = trades.numTrades();
		if (numTrades > 0){
			for (int i = 0; i < numTrades; i++){
				numer += trades.getPrice(i) * trades.getQuantity(i);
				denom += trades.getQuantity(i);
			}
			priceList.add(numer/denom);
		}else{
			priceList.add(priceList.get(getTime() - 1));
		}
	}
```

```
private void setTotalCapacity(){
        // Calculate the Market Capacity for the current time step.

        double capTotal = 0;
        for (int i = 0; i < producerList.size(); i++){
                Producer tempProducer = producerList.get(i);
                capTotal += tempProducer.getCapacity(getTime());
        }
        capacityList.add(capTotal);
}

private void setTotalInventory(){
        // Calculate the Market Inventory for the current time step.

        double invTotal = 0;
        for (int i = 0; i < producerList.size(); i++){
                Producer tempProducer = producerList.get(i);
                invTotal += tempProducer.getQuantity(getTime());
        }
        inventoryList.add(invTotal);
}

private void setTotalProduction(){
        // Calculate the Market Production for the current time step.

        double proTotal = 0;
        for (int i = 0; i < producerList.size(); i++){
                Producer tempProducer = producerList.get(i);
                proTotal += tempProducer.getProduction(getTime());
        }
        productionList.add(proTotal);
}

private void setTotalDemand(){
        // Calculate the Market Demand for the current time step.

        double demTotal = 0;
        for (int i = 0; i < consumerList.size(); i++){
                Consumer tempConsumer = consumerList.get(i);
                demTotal += tempConsumer.getQuantity(getTime());
        }
        demandList.add(demTotal);
}

public Schedule getSchedule(){
        // Get method required by Repast.
        return schedule;
}

public int getTime(){
        // Returns the current Repast defined time.
        // This time starts at one.
        return (int)schedule.getCurrentTime()-1;
}

public String[] getInitParam(){
        // This method customizes the parameter box.
        // Each entry must have a get method and a set method as well.

        String[] initParams = {"Consumer_Amount", "Producer_Amount", "Price",
                        "Consumer_Demand", "Consumer_Demand_Width", "Producer_Supply",
```

```
                              "Producer_Supply_Width", "Producer_Capacity",
"Producer_Capacity_Width",
                              "Cost", "Slope_Loss", "Slope_Break", "Slope_Profit",
"Utilization_Ratio",
                              "Capacity_Increase", "Demand_Annual_Rate"};
               return initParams;
       }

       // The following are the necessary get and set methods
       // needed to modify the class variables using the parameter box.
       public int getConsumer_Amount(){
               return numConsumers;
       }

       public void setConsumer_Amount(int cons){
               numConsumers = cons;
       }

       public int getProducer_Amount(){
               return numProducers;
       }

       public void setProducer_Amount(int prods){
               numProducers = prods;
       }

       public double getPrice(){
               return startPrice;
       }

       public void setPrice(double price){
               startPrice = price;
       }

       public double getConsumer_Demand(){
               return cDemand;
       }

       public void setConsumer_Demand(double demand){
               cDemand = demand;
       }

       public double getConsumer_Demand_Width(){
               return cDemandWidth;
       }

       public void setConsumer_Demand_Width(double width){
               cDemandWidth = width;
       }

       public double getProducer_Supply(){
               return pSupply;
       }

       public void setProducer_Supply(double production){
               pSupply = production;
       }

       public double getProducer_Supply_Width(){
               return pSupplyWidth;
       }
```

```
public void setProducer_Supply_Width(double width){
	pSupplyWidth = width;
}

public double getProducer_Capacity(){
	return pCapacity;
}

public void setProducer_Capacity(double capacity){
	pCapacity = capacity;
}

public double getProducer_Capacity_Width(){
	return pCapacityWidth;
}

public void setProducer_Capacity_Width(double width){
	pCapacityWidth = width;
}

public double getCost(){
	return pCost;
}

public void setCost(double cost){
	pCost = cost;
}

public double getSlope_Break(){
	return slopeBreak;
}

public void setSlope_Break(double point){
	slopeBreak = point;
}

public double getSlope_Loss(){
	return slopeLoss;
}

public void setSlope_Loss(double slope){
	slopeLoss = slope;
}

public double getSlope_Profit(){
	return slopeProfit;
}

public void setSlope_Profit(double slope){
	slopeProfit = slope;
}

public double getUtilization_Ratio(){
	return utilRatio;
}

public void setUtilization_Ratio(double ratio){
	utilRatio = ratio;
}

public double getCapacity_Increase(){
	return pCapacityAdd;
```

```
	}

	public void setCapacity_Increase(double amount){
		pCapacityAdd = amount;
	}

	public double getDemand_Annual_Rate(){
		return demandGrowth;
	}

	public void setDemand_Annual_Rate(double ratio){
		demandGrowth = ratio;
	}


	/**
	 * @param args
	 * This is the main method.  This calls methods
	 * provided by Repast in order to start the program.
	 */
	public static void main(String[] args) {
		SimInit init = new SimInit();
		DowModel model = new DowModel();
		init.loadModel(model, "", false);

	}

}
```

Appendix 2 – Producer Source Code

```
package dow.phase1;

import java.util.ArrayList;
import java.util.Collections;

import uchicago.src.sim.util.Random;

public class Producer {

        // These are the variables and lists unique to every consumer agent.
        // The idNumber currently is not used at all.
        private int idNumber;
        private ArrayList<Double> price = new ArrayList<Double>();
        private ArrayList<Double> quantity = new ArrayList<Double>();
        private ArrayList<Double> tempQuantity = new ArrayList<Double>();
        private ArrayList<Double> weightList = new ArrayList<Double>();
        private ArrayList<Double> production = new ArrayList<Double>();
        private ArrayList<Double> capacity = new ArrayList<Double>();
        private double cost;

        public Producer(int idNum, double producerPrice, double pSupply, double pSupplyWidth,
double midCapacity, double pCost, double capWidth){
                // This is the producer constructor.

                // Agent Id
                idNumber = idNum;

                // Sets the constant random cost for a producer
                cost = Random.uniform.nextDoubleFromTo(pCost, pCost + .1);

                // Initial quantity amount (Inventory)
                // Use tempQuantity for the trading algorithm.
                // Use quantity to keep track of total quantities.
                double amount = Random.uniform.nextDoubleFromTo(pSupply - pSupplyWidth, pSupply +
pSupplyWidth);
                quantity.add(amount);
                tempQuantity.add(amount);

                // Production list keeps track of how much new product a producer makes each step.
                // This is arbitrarily initialized to the initial inventory amount.
                production.add(amount);

                // Set the initial random capacity for a producer.
                capacity.add(Random.uniform.nextDoubleFromTo(midCapacity - capWidth, midCapacity +
capWidth));

                // Set the initial desired price for trade.
                price.add(Random.uniform.nextDoubleFromTo(producerPrice - .3, producerPrice +
.3));


                // Create weightList for price averaging
                // The weights below correspond to prices for time n
                // 0: n-3
                // 1: n-2
                // 2: n-1
```

```
		// 3: Current Price
		// 4: Market

		// Create three random weights.
		weightList.add(Random.uniform.nextDoubleFromTo(0, .4));
		weightList.add(Random.uniform.nextDoubleFromTo(0, .3));
		weightList.add(Random.uniform.nextDoubleFromTo(0, .2));

		// Sum the three random weights in order to determine the fourth weight.
		double threeWeightSum = 0.;
		for (int i = 0; i < weightList.size(); i++){
			threeWeightSum += weightList.get(i);
		}

		// Create the fourth weight.
		weightList.add(1 - threeWeightSum);

		// Sort the weight list in ascending order by default.
		Collections.sort(weightList);

		// Create fifth weight, which is the market weight.
		weightList.add(.0005 + Random.uniform.nextDoubleFromTo(0, .001));
	}

	public void step(int currentTime, double marketPrice, double marketDemand, int
numProducers, double slopeLoss, double slopeProfit, double slopeBreak, double utilRatio, double
pCapAdd){
		// This is done after each trade and before the next time step.
		priceStep(currentTime, marketPrice);
		quantityStep(currentTime, marketDemand, numProducers, slopeLoss, slopeProfit,
slopeBreak);
		capacityStep(currentTime, utilRatio, pCapAdd);
	}

	public void priceStep(int time, double mPrice){
		// This method uses the weight list to determine the
		// expected price of purchase for the next trading round.

		// In this method we first determine a desired price based on inventory levels.
		// Then we smooth this price using current price and the market price
		double naturalPrice = 0;
		double logicalPrice = 0;
		double desiredPrice = 700;
		double newPrice = 0;

		naturalPrice = Math.pow(capacity.get(time) / (.001 + tempQuantity.get(time)), .7);
		logicalPrice = Math.max(0, naturalPrice);
		desiredPrice = Math.min(700, logicalPrice);

		double marketAdjust = weightList.get(4)* mPrice;
		double otherAdjust = 1 - marketAdjust;

		// This calculates the new expected price based on what time it is.
		switch (time){
		case 0:
			newPrice = marketAdjust + otherAdjust * (weightList.get(3) * desiredPrice
+ (1 - weightList.get(3)) * price.get(0));
			break;
		case 1:
			newPrice = marketAdjust + otherAdjust * (weightList.get(3) * desiredPrice
+ weightList.get(2) * price.get(1) + (1 - weightList.get(3) - weightList.get(2)) * price.get(0));
			break;
```

```
		default:
			newPrice = marketAdjust + otherAdjust * (weightList.get(3) * desiredPrice
+ weightList.get(2) * price.get(time) + weightList.get(1) * price.get(time - 1) +
weightList.get(0) * price.get(time - 2));
			break;
		}

		// This prevents the price from changing by more than ten percent.
		if (newPrice < .9 * price.get(time)){
			newPrice = .9 * price.get(time);
		}else if(newPrice > 1.1 * price.get(time)){
			newPrice = 1.1 * price.get(time);
		}
		price.add(newPrice);
	}

	public void quantityStep(int time, double mDemand, int numProducers, double slopeLoss,
double slopeProfit, double slopeBreak){
		// This adjusts the production levels by deciding
		// how much production is needed for the next trade round.

		// This is the leftover production not sold from
		// the current trade round.
		double remainingAmount = tempQuantity.get(time);

		// Calculate the ratio of capacity to produce for next round.
		double ratio = 0.;
		if (price.get(time) > cost){
			ratio = Math.min(1, slopeProfit * ((price.get(time) - cost) / cost +
slopeBreak));
		}else{
			ratio = Math.max(0, slopeLoss * (price.get(time) - cost) / cost +
slopeBreak);
		}

		// Calculate new production quantity based on the ratio of current capacity.
		double additionalAmount = ratio * capacity.get(time);
		double differentAdditionalAmount = (additionalAmount + (mDemand / numProducers)) /
2;
		additionalAmount = Math.min(differentAdditionalAmount, additionalAmount);

		// Calculate the new inventory amount
		double newAmount = additionalAmount + remainingAmount;
		production.add(newAmount - remainingAmount);
		quantity.add(newAmount);
		tempQuantity.add(newAmount);
	}

	public void capacityStep(int time, double utilRatio, double pCapAdd){
		// This adjusts the capacity levels based on length of high utilization.

		double old = capacity.get(time);

		if ((time > 70) && utilization(time, utilRatio)){

			capacity.add(.999 * old + pCapAdd);
		}else{
			capacity.add(.999 * old);
		}
	}

	// Average utilization calculation
```

```
	// Takes in a time and averages over 12 steps
	// If over utilRatio utilization then return a true value
	public boolean utilization(int time, double utilRatio){
		double sumProduction = 0;
		double sumCapacity = 0;

		for (int i = 0; i < 12; i++){
			sumProduction += production.get(time - 62 + i);
			sumCapacity += capacity.get(time - 62 + i);
		}

		double ratio = sumProduction / sumCapacity;

		if (ratio > utilRatio){
			return true;
		}else{
			return false;
		}
	}

	//
	// Get and set methods
	//

	// Expected Price variable
	// Just need a get
	public double getPrice(int time){
		return price.get(time);
	}

	// Capacity variable
	// Need a get
	public double getCapacity(int time){
		return capacity.get(time);
	}

	// Production variable
	// Need a get
	public double getProduction(int time){
		return production.get(time);
	}

	// Quantity variable represents starting inventory at each step for a producer
	// Need a get
	public double getQuantity(int time){
		return quantity.get(time);
	}

	// Temporary Quantity variable
	// Use this to adjust remaining demand after trades
	// Need a get and a set
	public double getTempQuantity(int time){
		return tempQuantity.get(time);
	}

	public void adjustTempQuantity(int time, double remainingQuantity){
		tempQuantity.set(time, remainingQuantity);
	}
}
```

Appendix 3 – Consumer Source Code

```
package dow.phase1;

// Import needed java classes.
import java.util.ArrayList;
import java.util.Collections;

import uchicago.src.sim.util.Random;

public class Consumer {

	// These are the variables and lists unique to every consumer agent.
	// The idNumber currently is not used at all.
	private int idNumber;
	private boolean findingTrade;
	private double lastTradePrice;
	private ArrayList<Double> price = new ArrayList<Double>();
	private ArrayList<Double> quantity = new ArrayList<Double>();
	private ArrayList<Double> tempQuantity = new ArrayList<Double>();
	private ArrayList<Double> weightList = new ArrayList<Double>();

	public Consumer(int idNum, double consumerPrice, double cDemand, double cDemandWidth){
		// This is the consumer constructor.

		// Agent Id Number
		idNumber = idNum;

		// Trading Status
		findingTrade = true;

		// Last price that a consumer bought at.
		lastTradePrice = 0;

		// Initializes quantity amounts.  Use tempQuantity for the trading algorithm
		// and use the quantity to keep track of the demand at the start of each
		// trading round.
		double amount = Random.uniform.nextDoubleFromTo(cDemand - cDemandWidth, cDemand +
cDemandWidth);
		quantity.add(amount);
		tempQuantity.add(amount);

		// Set random initial desired price for trade
		price.add(Random.uniform.nextDoubleFromTo(consumerPrice - .01 * consumerPrice,
consumerPrice + .01 * consumerPrice));

		// This creates the weightList for price averaging.
		// The weights below correspond to prices for time n
		// 0: n-3
		// 1: n-2
		// 2: n-1
		// 3: Current Price
		// 4: Market

		// Create three random weights.
		weightList.add(Random.uniform.nextDoubleFromTo(0, .4));
		weightList.add(Random.uniform.nextDoubleFromTo(0, .3));
		weightList.add(Random.uniform.nextDoubleFromTo(0, .2));
```

```
		// Sum the three random weights in order to determine the fourth weight.
		double threeWeightSum = 0.;
		for (int i = 0; i < weightList.size(); i++){
			threeWeightSum += weightList.get(i);
		}

		// Create the fourth weight.
		weightList.add(1 - threeWeightSum);

		// Sort the weight list in ascending order by default.
		Collections.sort(weightList);

		// Create fifth weight, which is the market weight.
		weightList.add(.0005 + Random.uniform.nextDoubleFromTo(0, .001));
	}

	public void step(int currentTime, double marketPrice, double demandGrowth){
		// This is done after each trade and before the next time step.
		priceStep(currentTime, marketPrice, demandGrowth);
		quantityStep(currentTime, demandGrowth);
		findingTrade = true;
	}

	public void priceStep(int time, double mPrice, double demandGrowth){
		// This method uses the weight list to determine the
		// expected price of purchase for the next trading round.

		// This is the leftover demand that was not satisfied
		// during the current trading round.
		double remainingDemand = tempQuantity.get(time);

		// This is the additional demand based on an adjustable demand
		// inflation rate. Each step is considered to represent a week.
		double additionalDemand = Math.pow(1 + demandGrowth / 52, time + 1) *
quantity.get(0);

		// This calculates the ratio to be used in the sensitivity calculation.
		double ratio = ((remainingDemand + additionalDemand) / additionalDemand);

		// This calculates the sensitivity factor for a new expected price.
		double sensitivity = Math.pow(ratio, 0.7);

		double newPrice = 0;
		double marketAdjust = weightList.get(4)* mPrice;
		double otherAdjust = 1 - marketAdjust;

		// This calculates the new expected price based on what time it is.
		switch (time){
		case 0:
			newPrice = marketAdjust + otherAdjust * price.get(0);
			price.add(sensitivity * newPrice);
			break;
		case 1:
			newPrice = marketAdjust + otherAdjust * (weightList.get(3) * price.get(1)
+ (1 - weightList.get(3)) * price.get(0));
			price.add(sensitivity * newPrice);
			break;
		case 2:
			newPrice = marketAdjust + otherAdjust * (weightList.get(3) * price.get(2)
+ weightList.get(2) * price.get(1) + (1 - weightList.get(3) - weightList.get(2)) * price.get(0));
			price.add(sensitivity * newPrice);
```

```
                        break;
                default:
                        newPrice = marketAdjust + otherAdjust * (weightList.get(3) *
price.get(time) + weightList.get(2) * price.get(time - 1) + weightList.get(1) * price.get(time -
2) + weightList.get(0) * price.get(time - 3));
                        price.add(sensitivity * newPrice);
                        break;
                }
        }

        public void quantityStep(int time, double demandGrowth){
                // This adjusts the demand levels by deciding
                // how much new demand to add for the next trade round.

                // This is the leftover demand not satisfied from
                // the last trade round.
                double remainingDemand = tempQuantity.get(time);

                // This ratio uses the consumer's calculated price and
                // last trade price.
                double ratio = (price.get(time) - lastTradePrice)/price.get(time);
                ratio = Math.max(0,Math.min(1, ratio));

                // This is the additional demand based on an adjustable demand
                // inflation rate and the ratio above. Each step is considered
                // to represent a week.
                double additionalDemand = ratio * Math.pow(1 + demandGrowth / 52, time + 1) *
quantity.get(0);

                // This sets a consumer's demand needed for the next trade round.
                double newAmount = Math.min( 100, remainingDemand + additionalDemand);
                quantity.add(newAmount);
                tempQuantity.add(newAmount);
        }

        //
        // The following are get methods and set methods for the consumer objects.
        //

        // Trade status variable
        // Need a get and a set
        public boolean getFindingTrade(){
                return findingTrade;
        }

        public void setFindingTrade(boolean state){
                findingTrade = state;
        }

        // Trade price variable
        // Need a get and a set
        public double getTradePrice(){
                return lastTradePrice;
        }

        public void setTradePrice(double tPrice){
                lastTradePrice = tPrice;
        }

        // Expected Price variable
        // Just need a get
        public double getPrice(int time){
```

```
		return price.get(time);
	}

	// Quantity (Demand) variable
	// This is the starting demand at each step for a consumer.
	// Need a get
	public double getQuantity(int time){
		return quantity.get(time);
	}

	// Temporary Quantity variable
	// Use this to adjust remaining demand after each trade.
	// Need a get and a set
	public double getTempQuantity(int time){
		return tempQuantity.get(time);
	}

	public void adjustTempQuantity(int time, double remainingQuantity){
		tempQuantity.set(time, remainingQuantity);
	}
}
```

Appendix 5 – Trade Engine Source Code

```
package dow.phase1;

import java.util.ArrayList;

public class Trader {
	// This object keeps track of all the trades
	// conducted in a given trading round.

	private int idNumber;
	private ArrayList<Double> price = new ArrayList<Double>();
	private ArrayList<Double> quantity = new ArrayList<Double>();

	public Trader(int idNum){
		// This is the single argument trade constructor.
		idNumber = idNum;
	}

	// The id number refers to the trading round,
	// which is also corresponds to the current time.
	// The id numbers start from one.
	public int getIdNumber(){
		return idNumber;
	}

	// For the following methods,
	// tradeNum references the index number for a given trade
	// while tradeAmount references the amount of a new trade.
	public double getPrice(int tradeNum){
		return price.get(tradeNum);
	}

	public void nextPrice(double tradePrice){
		price.add(tradePrice);
	}

	public double getQuantity(int tradeNum){
		return quantity.get(tradeNum);
	}

	public void nextQuantity(double tradeAmount){
		quantity.add(tradeAmount);
	}

	// This returns the quantity of trades for a given trader object.
	public int numTrades(){
		return price.size();
	}
}
```