EOS: Developing an General Agent-Based Economic Simulation

Chris Rucinski ‘10

Submitted for spring semester independent work in the COS Department at Princeton University

Faculty Advisor:
Kenneth Steiglitz

Undergraduate Collaborators:
Ye “Cody” Wang ‘10
Michael Adelson ‘11

May 4th 2009

Abstract

Economics has always been an important field of study for the simple reason that, at its idealistic heart, it is the study of how to influence the trade and utilization of limited resources in order to make people as happy as possible. However, economics is also a field in which it is very difficult to prove assertions to be absolutely true because economics concerns itself with the behavior of people, and people all too often don't know how they themselves will behave in certain situations, never mind predicting how other people will behave. This difficulty motivates a certain skepticism about how successful current economic thought actually is in terms of making people happier. Of course a contemporary example of one of economics' failures is the current recession, the cause of which is a contentious issue agreed upon by few but has been blamed on everyone from the speculators on Wall Street to mortgage holders to government officials who perhaps failed to regulate the interactions of the preceding parties. The complex human element that predicated this recession is often overlooked by economic theory, which instead tends to focus on aggregate behavior that fits nicely into elegant equations. However, the elegance of these equations is lost in the chaos that ensues when they fail to accurately predict and prevent economic downturns.

In this paper we assert that economics lends itself more to analysis by simulation rather than analysis by equations, which tend to abstract away too many relevant details to be valid. More specifically, we assert that a simulation that models the interactions of individual agents in an economy is more useful than equations that aggregate individual behavior away into averages. Furthermore, we discuss the design and implementation of a new framework for such a simulation, EOS, which we argue is an improved version of

previous similar projects that informed its development. Finally, we discuss a few experiments with the current implementation that both demonstrate valid economic conclusions and bring to light a few of its limitations, which in turn motivate possible improvements for the future.

Background to EOS

Before we proceed much farther, it would be wise to define the general structure of any agent-based economic simulation. In any of these simulations, time is discretized into timesteps during which the agents in the simulation interact with each other in ways defined by the simulation. Such actions may include eating, trading, processing or creating resources, reproducing, dying, etc. Each simulation is given certain parameters that may affect the amount of resources agents begin with, the behavior of agents in specific situations, etc. Once the simulation begins, it is autonomous in the sense that the researcher running it can't intervene at an arbitrary timestep and save an economy from collapsing or agents from dying. The hope is then that interesting behavior, in terms of global prices of goods for instance, will present itself, and then the parameters of the simulation can be tweaked to determine what parameters affect the outcome in what way in order to build more realistic models. For a complicated enough simulation the discovered causal relationship between the parameters and the result may then inform economic policy decisions.

An example of such a simulaiton was MinSim, the predecessor to EOS on which EOS is based. MinSim's initial implementation was predicated on what came to be known as the "gold-food" model, and was ported from C to Java by Chris Chan for his

senior independent work at Princeton in 2008 [1]. The idea was that there were two types of agents in the model that I'll call laborers and traders. On each timestep both laborers and traders had to consume a unit of food and could trade food for gold or gold for food. Laborers had the additional ability to produce a certain fixed amount of either gold or food on each turn. The need for trade arose from the fact that traders and some laborers couldn't produce enough food to feed themselves each turn, and so had to trade gold for food in order to survive.

In this initial implementation, Minsim appeared to be successful in showing that traders could stabilize absolute prices of food by essentially amplifying price signals due to their "buy low, sell high" behavior [2]. However, this is a dubious achievement because one would expect the price of food to rise for the following reason: every timestep laborers were producing gold, the currency in which the price of food was measured, which means that inflation was occurring, but the price of food was staying relatively constant!

MinSim also appeared to be flawed in other ways, and these flaws eventually convinced us to start over with EOS. Chris Chan expanded upon the gold-food model by adding farms, banks, and tractor factories in the second semester of his project [3]. What he failed to note in his paper is that stable runs of the expanded economy only occurred in about 16% of the simulations for this expanded model (this was noted by both Daniel Hayes-Peterson [4] and Eric Vreeland [5] in their papers on the topic) for an unknown reason. Vreeland's paper is almost entirely dedicated to tracking down why so many runs failed, and though he doesn't narrow down the reason exactly, he does make some damning observations about the MinSim framework.

Vreeland's eventual conclusion was that so many runs were failing because money was inadvertently being injected into the simulation through shoddy bookkeeping within the framework [6]. Equally disheartening is that he noticed that while laborers were in theory required to consume food each timestep, if they had no food then this requirement was not enforced [7]. In other words, agents didn't die. Furthermore, there was no way to add the ability to die to the MinSim framework without rewriting large parts of the framework [8].

At this point it was clear that MinSim was broken rather severely. Not only was there a bug in a basic mechanism in the program, but also the framework wasn't flexible enough to be extended in a very elementary way. We also took into account MinSim's utilization of the RepastJ framework. RepastJ is a library for implementing just this sort of simulation, but we found that using it demanded too much overhead for such a small model and that we would likely gain more flexibility (for instance, in terms of how output was displayed) if we were to begin again from scratch. Having learned from the mistakes of MinSim, we set out to create a framework for a simulation that was flexible enough to be extensible in every direction by anyone who took up the project after us.

Design Decisions and Implementation

Whereas most economic simulations are designed to model a particular economy that exists in the world (Electricity Market Complex Adaptive System (EMCAS) [9], a model of the electricity economy in Illinois is one example of many), our goal in EOS was to develop an extensible model that could be used to study economics in the most general sense. We looked to Leigh Tesfatsion's work (her website [10] was a great

general resource to get started) for guidance, as she appears to be the current authority on agent-based computational economics (ACE), which is just another name for the type of simulation we concern ourselves with in this paper. In particular we reaffirmed that Java was a very suitable language in which to develop because its object-oriented nature and support for class inheritance provided a natural structure on which we could impose a taxonomy of agents (following the advice of Tesfatsion [11]). In fact we called the result of our labor EOS, an acronym for Economic, Object-oriented Simulation, to emphasize just that feature. Organizing the simulation into a suitable hierarchical taxonomy would allow it to be easily extensible in the sense that the amount and locality of coding necessary scales with the complexity of whatever new mechanism is being added to the simulation. That is, conceptually small changes should only require minimal recoding in only a few different files. Also, to aid both ourselves and our successors to the project, we aspired to impose strict error checking so as to catch invalid parameters to methods as early as possible in order to speed up the process of debugging, which can be time-consuming with unfamiliar projects. Finally, in the context of this high-quality model, we hoped to recreate some of the results of MinSim, or at least give some demonstration that our simulation actually produced meaningful results.

The three high-level classes on which we chose to organize our new framework were Agent, Good, and Market (note that there is a very good description of the EOS structure in the Ajent (a previous name for our creation) Documentation [12] that Michael Adelson put together, which the next few paragraphs reproduce in part). Conceptually, these were the three entities that seem necessary to have a minimal working economy. Different agents could own goods of various types and trade them

according to the rules defined by implementations of markets. Agents could be anything from laborers to firms to banks; the only things that agents must be able to do is possess goods and be added and removed from the simulation (i.e. because of death in the case of a laborer and because of bankruptcy in the case of a firm). Every other class in our simulation inherits from one of these three, except for the Economy class, which essentially is the driver of the simulation itself.

The Economy class is responsible for running the simulation by calling the public act() method of every agent in the simulation (so that each agent could interact with the simulation) and calling the public clear() method of every market in the simulation (so that bids could be resolved) on every timestep. We tried to keep the economy class as uncluttered as possible. For instance, in our baseline implementation of an economy, we have laborer agents who must eat every timestep, and the natural question that arose was where this eating would be enforced. Michael and I toyed around with the idea of enforcing this in the Economy whereby a hypothetical eat() method could be called every timestep as well. However, if many similar mechanisms had to be enforced for future agents, it would make more intuitive sense to associate those mechanisms with the agents themselves rather than having them all bunch up at the Economy class. Therefore, Michael and I eventually decided it would be better to enforce eating in a subclass of Agent that was also an abstract superclass of the actual laborer implementation. In the end we had Agents that act()ed, Laborers that extended Agent and eat()ed in the act() method as well as calling bid(), which was implemented by, for instance, a SimpleLaborer that extended Laborer and implemented bid() according to some simple rules. This effectively separates the enforcement of eating and other behavior; if an Agent

is a Laborer, then one can be sure that it would eat every timestep, but behavior outside of that depends on each specific implementation.

The other major design decision that influenced the simulation a fair amount was deciding how objects would communicate in order to pass on information that was necessary for the simulation to work properly. We decided to err on the side of caution and include a way for every object in the simulation to communicate with every other object, or at least, if two objects couldn't communicate (for instance, to prohibit collusion between laborers when selling their labor), that functionality could be easily added if desired. The Economy object has a reference to every object that needs to potentially perform an action on every timestep: agents, markets, etc. Because of this, any objects in the simulation just needs a reference to the economy itself in order to communicate with other objects in the simulation. For instance, the way that agents access markets to submit bids is to call a method (getMarketFor()) from their instance of the economy that returns the relevant market.

Finally, we wanted to create a way to selectively and generally output information about each run of the simulation, and the natural place to do this was of course the Economy class since it has a reference to every other object in the simulation. Michael was responsible for implementing the code that essentially allowed the person running the simulation to select between a few different "printers" which outputted information with some frequency to a CSV spreadsheet that thereby chronicled the events of the simulation. For instance, there are currently printers that allow the price of food and labor to be printed, and since the economy has access to every object in the simulation, there is essentially limitless potential for further printers to be added.

We now examine the motivation behind the design of the three high level classes and the classes that extend them, beginning with markets. Again, we strove for simplicity. Markets have only two goods which are both traded for each other within the market; one good is designated the currency, and the other the product. To submit bids to buy or sell goods, agents pass references to their goods into the relevant market (i.e. in order to sell food for money, agents pass food and money into the market where food is the product and money is the good) along with their offer of how much to buy or sell and at what price. The methods to buy and sell are two distinct methods, and agents always buy the product with the currency and sell the product in return for the currency. When markets clear(), they use the references to the goods passed to them in order to increase and decrease the quantities of corresponding goods owned by two agents who have agreed to trade via the bids.

In the decision to designate one good as the currency, it may appear as though we sacrificed some generality. For instance, it doesn't seem as though our model could support an economy based entirely or barter, where the goods being traded are both products with inherent value (i.e. neither is just money). However, this situation can be handled in our model by simply creating a market that has one of the goods to barter as its product and the other as its currency. This is a bit unintuitive since it introduces a needless asymmetry between the goods, whereby if an agent wanted to trade wool for food, it would have to know which was the designated currency in the market in order to decide whether to call the buy() or sell() method of the market. For this reason, our implementation of markets has methods to identify which goods are designated as products and currency.

A viable alternative implementation of markets is to have no distinction between the two goods associated with it and a single method called something like submitOffer(). Agents would pass references to their goods in a particular order that designated that they wanted to trade a quantity of the first good in return for the receipt of a second good. While elegant, this implementation is a bit unintuitive since most people think of buying and selling as two distinct actions. In this implementation we felt it would be difficult to constantly switch between mindsets where for instance in order to sell food, one is really buying money with food, and in order to buy food, one is really selling money for food. People don't usually conceptualize trade as the buying and selling of money. In order to mitigate this potential confusion and any errors it would generate, we opted against this implementation, realizing too that most economic simulations would probably have a set currency, so clarity in the majority of implementations outweighed the cost of a loss of elegance.

Goods were much more straightforward to implement than the markets that organized their trade. In our current implementation, Good objects have a single field for the good's quantity and methods for accessing that quantity, increasing it, and decreasing it. The quantity may be either continuous or discrete, and this is indicated both by the subclasses DiscreteGood and ContinuousGood, and by a method in Good, isDiscrete(). An alternative implementation that we quickly decided against was to have each instance of a Good represent a discrete unit of a good, but of course this left no way to deal with continuous goods, and furthermore it would have led to a deluge of instances of Goods, which would not have scaled very well at all. Structuring the representation of goods into

single objects also allowed quantities of goods to be grouped nicely where an instance of a Good represented the quantity of a good owned by a single agent.

Agents themselves on a high level were also not very complicated. Besides a reference to the economy that they're in, the only other important field is really a map of goods that maps a String to an instance of a good possessed by that agent (i.e. "Food" maps to an instance of a Food object). As already mentioned, Agents have an act() method, and they also have an isAlive() method which lets the economy know if an Agent has died. Agents also have a die() method, but this method is protected. die() could have been made public, with the onus of killing Agents then resting on the economy, but just like the discussion of eat() earlier, it would have led to too much clutter in the Economy class, so we gave the responsibility of killing Agents to the subclasses of Agents, which we'll discuss next.

We implemented Laborer as a subclasss of Agent and built the specific attributes of Laborers into the act() method. Within the Laborer act() method, the laborer is forced to eat and is granted a timestep's worth of labor to sell on the labor market. Within the eating subroutine, the laborer also calls die() if it doesn't have enough food to eat on that timestep. At the end of act() the Laborer calls an unimplemented method called bid(), which needs to be implemented by classes that extend Laborer in order to interact with the simulation. This structure separates all the required behavior of laborers into the Laborer class, and leaves implementations of interesting behavior to subclasses. This seems to be ideal for a variety of circumstances; one we had in mind was using this model as a classroom assignment to create Laborers to compete in an economy populated by each student's Laborer implementation. In order to interface with the simulation

properly, all implementations would have to extend Laborer, so they would be guaranteed to eat every turn or die and receive a set amount of labor every turn. Now, since agents have access to their list of goods, it is possible for them to increase them directly by calling deposit() for the relevant good. However, since this is the only way to increase the quantity of a good, it would be very easy to check in this hypothetical assignment whether or not laborers were cheating by examining the source code for any instances of calls to deposit().

Firms and the Farms that extend them were implemented in a very similar way to Laborers. The only behavior that Firms are required to implement is that at the end of every timestep, any labor that they bought must be discarded. The way that firms would use labor productively is by calling an unimplemented produce() method that converts labor into another good according to some function. Firms don't need to eat as Laborers do, but they can die if they have no money on hand and no products to sell either. As in the Laborer class, this check on dying is made in the act() method of Firms, but the implementation of the check is left to subclasses like the Farm.

Farms extend Firms and specify Food as the good that they sell. Farms implement a conversion from Labor to Food as well as a declaration of bankruptcy if the farm has no money or food (to sell) or labor (to convert to food). Also just like laborers, Farms have an unimplemented bid() method which subclasses are required to implement. By organizing things in this way, subclasses of Farms convert labor to food in a way enforced by the farm, separating the required and optional behavior of the subclasses. By separating Firms and Farms, we leave open the possibility of adding Firms that produce other goods from labor while simultaneously enforcing the purge of unused labor as high

up in the inheritance hierarchy as possible. As with this decision, the driving factor in all of the decisions of the EOS framework have been to be as simple and as extensible as possible.

The Ajent Documentation [13] has a fair amount of examples of how the model might be extended, namely by creating agents that reproduce, adding tractors as capital for farms, and adding the support for contracts between agents (see the documentation for a more thorough discussion about how these might be added). Another addition to the model that we discussed could be the addition of firms that are owned by laborers or groups of laborers. As it is now, firms and the farms that extend them have no owners, and this became a bit of an obstacle when we tried to get a baseline version of an economy running (this will be discussed later). It wouldn't be terribly difficult to create a new type of Firm called OwnedFarm that had a reference to an Agent that owned it; the owning Agent would extend a new class OwnerAgent and would also have a reference to the OwnedFarm. In the OwnerAgent's act(), the agent could call methods in the OwnedFarm that dictated bidding on food and labor. OwnedFarms would still enforce the dissolution of all their labor through their own act() method, which would not do any bidding on food or gold, and their act() could not be overwritten by subclasses. Since managing a farm would take time, we might stipulate that OwnerAgents that own a farm can't use their labor to work on other farm for instance. If an OwnerAgent deemed that a farm they own is no longer making sufficient money, they could abandon it in favor of working on another farm or founding a different type of firm. As a final note, the way that agents would start a farm is buy calling a method that already exists, addAgents(), in their reference to the economy. There are many possibilities for extensions like this one,

but once we had the basic parts of an economy implemented, we wanted to create a baseline working model as a proof of concept.

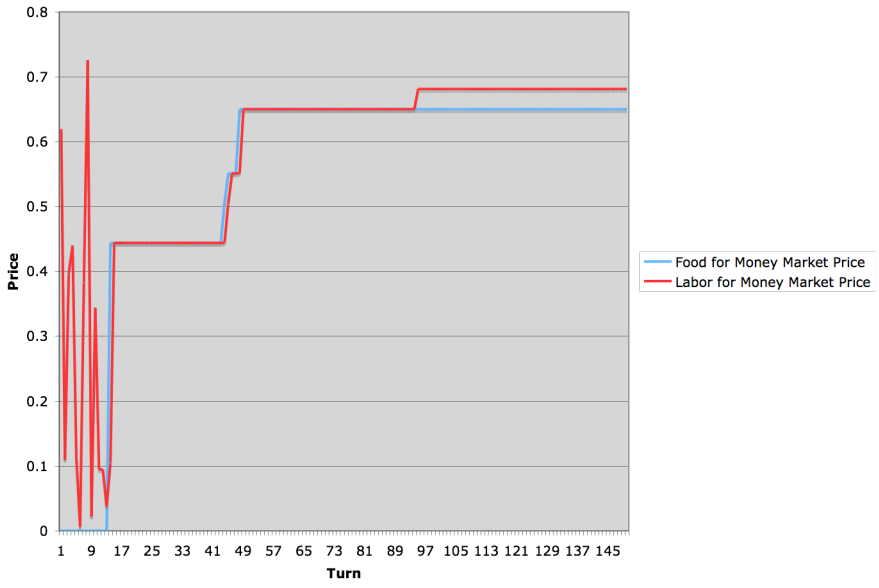
A Baseline Implementation

In theory, we believed our organization of the model was sound, but a theory is no good until its been verified empirically, and of course if our simulation didn't produce economic results that made any sense, it wasn't a very useful model. Therefore, we took a step back and laid out a few requirements that, if met, would mean that our simulation worked in a useful way. We started out by assuming one farm and around a dozen laborers. The first requirement is that most of the laborers need to survive for a reasonable amount of time, assuming that the single farm could support them all in theory. After all, nothing interesting would ever happen in our simulation after all the laborers die. The second is that the prices of one day's worth of food and one day's worth of labor for laborers should oscillate around some fixed value; the values needn't be the same, and it was alright if the oscillations were fairly wild (this was illustrated by the initial implementation of MinSim [14]). Finally, the price of one day's worth of labor should be greater than or equal to the price of one day's worth of food. This is more of a corollary to the previous requirements; that is, if the laborers don't die, and their only way to get money is to work, then they must be getting paid enough for one day's labor to eat one day's worth of food.

The first thing we needed to do was decide upon a market to use in the simulation, and for this we looked to MinSim for guidance. Like MinSim, we used a call auction market [15] called, fittingly enough, CallAuctionMarket that Michael coded up. The

basic idea is that sell bids and buy bids each form their own curve in the price-quantity plane, and the market price and quantity traded on that timestep are set at the intersection of the two curves. Then, bids are cleared at that price such that the highest buy offers and lowest sell offers are cleared first. An important note about this market is that on each turn, each agent can submit as many bids to the market as they like since each bid is cleared independently of every other bid. This allows agents to effectively submit a supply or demand curve for the product they are selling or buying, something that was not supported in MinSim but turned out to be very useful when we went about implementing our baseline simulation.

After the three of us played around with ideas, I implemented a very rudimentary SimpleFarm and SimpleLaborer that exhibited a few of the requirements. In the simulations that followed, most of the time 3 SimpleLaborers survived indefinitely with a single SimpleFarm, and this effect scaled, so 4 SimpleFarms could support 12 SimpleLaborers. A representative simulation is below:



Note that the above graph only displays 150 timesteps to show the initial random behavior before things settle down; after the period displayed by the graph, prices remain relatively fixed. As the graph makes clear, prices remain pretty stable and labor usually costs more than food, but oscillations in these prices are not present. This behavior, while not ideal, did at least demonstrate that the code for the interactions between agents was functioning, and the lack of oscillations makes a lot of sense once one examines how SimpleFarm and SimpleLaborer make their bidding decisions.

SimpleFarm bids on buying labor by bidding the expected revenue for the next unit of labor until the expected revenue falls below a certain amount or the farm runs out of money (when a bid is submitted, the market deducts the appropriate amount of money from the agent that submitted the bid). That is, for the next unit of labor, the farm examines how much food that labor would produce according to its production function, multiplies that by the last price of food, and bids that amount for the labor. SimpleFarm sells food at the last price of food or a random number in some range if it's the first timestep.

SimpleLaborer buys food according to a linear function determined by its food threshold. The idea is that SimpleLaborers would like to have a comfortable stockpile of food for the near future and are willing to pay more for a day's worth of food if they are closer to having no food left to eat (this is another idea we borrowed from MinSim). Therefore, SimpleLaborers don't bid on food if their food on hand is greater than or equal to their food threshold, and they bid linearly increasing amounts of money the less food they have, up until they bid all of their money on hand in the event that they have a timestep's worth of food left. SimpleLaborers sell their labor at the last price of labor.

Now, of course with two of the four strategies for bidding coming directly from previous prices, it should come as no surprise that once prices reach a value that can support the remaining agents, prices fluctuate very little thereafter. However, this is also very artificial since neither farms nor laborers attempt to deviate from the previous market prices in a way that gives them an advantage. For instance, one thing that we noticed with the previous simulation was that farms were accumulating a lot of food, more than enough to have fed laborers that had died. With this in mind, we set out to create a more realistic simulation that exhibited all the properties we previously mentioned.

All three of us spent a fair amount of time trying to tweak the implementations of farms and laborers to achieve a working model, but it was eventually Michael who got a working pair with BudgetLaborer and DemandCurveFarm. One thing we noticed as we experimented was that our agents tended to amplify price swings by basing their bidding decisions only on the prices of the last timestep; often the economy couldn't recover from these drastic swings and the laborers died. Therefore, just as in MinSim model did in its traders, we tried using exponential smoothing to allow the agents to take advantage of the history of the simulation they were in; both BudgetLaborer and DemandCurveFarm ended up using this method of incorporating history into their bidding strategies.

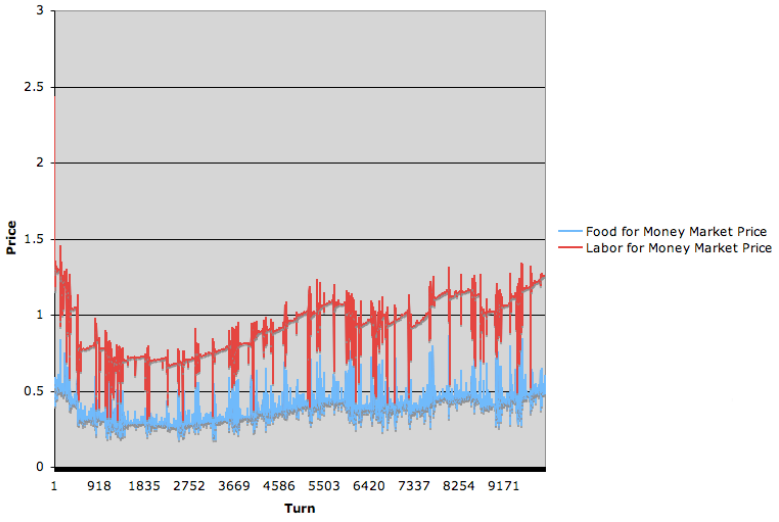
DemandCurveFarm ended up being not so different from SimpleFarm, but the differences solved the problem of farms accumulating large amounts of food that they would never sell. The heart of the problem is that farms with no owners have no incentive to hire laborers and produce food. Whereas laborers must eat and will die if they don't work to get money to buy food, farms could sit on their money and exist forever while

the laborers around them died en masse. Therefore, while the strategy for buying labor was unchanged from SimpleLaborer (except that it now implemented exponential smoothing) and was therefore just as competitive, the new strategy for selling food was very benevolent. DemandCurveFarms sell all their food at a price equal to (exponentially smoothed price of food) * (constant) / (current food on hand), which translates to an inverse variation between food on hand and bidding price, meaning that these farms sell food at very low prices as soon as their food on hand approaches the constant above. With these changes, DemandCurveFarms generously sell at very low prices any food that they would otherwise hoard until the end of the simulation.

BudgetLaborers differ markedly from SimpleLaborers, and these differences allow them to stay alive indefinitely in simulations with enough farms to produce food. When selling their labor, BudgetLaborers expand upon the knowledge that they should get paid enough to eat for at least one additional timestep. Therefore, if their food on hand is greater than their food threshold, they sell their labor at a price equal to (exponentially smoothed price of food) * (food on hand) / (food threshold) since they don't need the money that badly and would rather take a chance on a higher bid. If the amount of food on hand is less than half their food threshold, they bid 75% of the exponentially smoothed market price of labor, choosing to undercut the current market price in an effort to be hired independent of the current cost of food. Finally, in the case where their food on hand is more than half the food threshold, but less than the food threshold, they bid slightly more than the current price of food, hoping to make enough money to eat another day and a little extra just to be safe.

The way that BudgetLaborers bid on buying food is based on the same sort of tiered bidding scheme that depends on the current amount of food the BudgetLaborer has. The key insight that Michael had about laborers is that in our previous attempts, laborers would often bid far too much money for food when they began to get desperate, and they would then have no way to rebound and achieve a reasonable stockpile of food since they had no money left. Therefore, the bidding scheme had to provide a way for laborers to rebound from this desperation and attain a comfortable level of food once again whilst preventing them from spending all their money on just a few days worth of food. To achieve this, BudgetLaborers set a budget at the beginning of every timestep that increases as their food reserve decreases, but the budget never goes above four times the price of food or below half the price of labor. If their food on hand is below half their food threshold, then they submit two high bids for food (within the budget) in an attempt to get enough food for the next two timesteps. Otherwise, they submit linearly decreasing bids for less money until they've exhausted their budget.

Below is a representative simulation with 30 BudgetLaborers and 5 DemandCurveFarms; all the laborers survived after 10,000 timesteps:



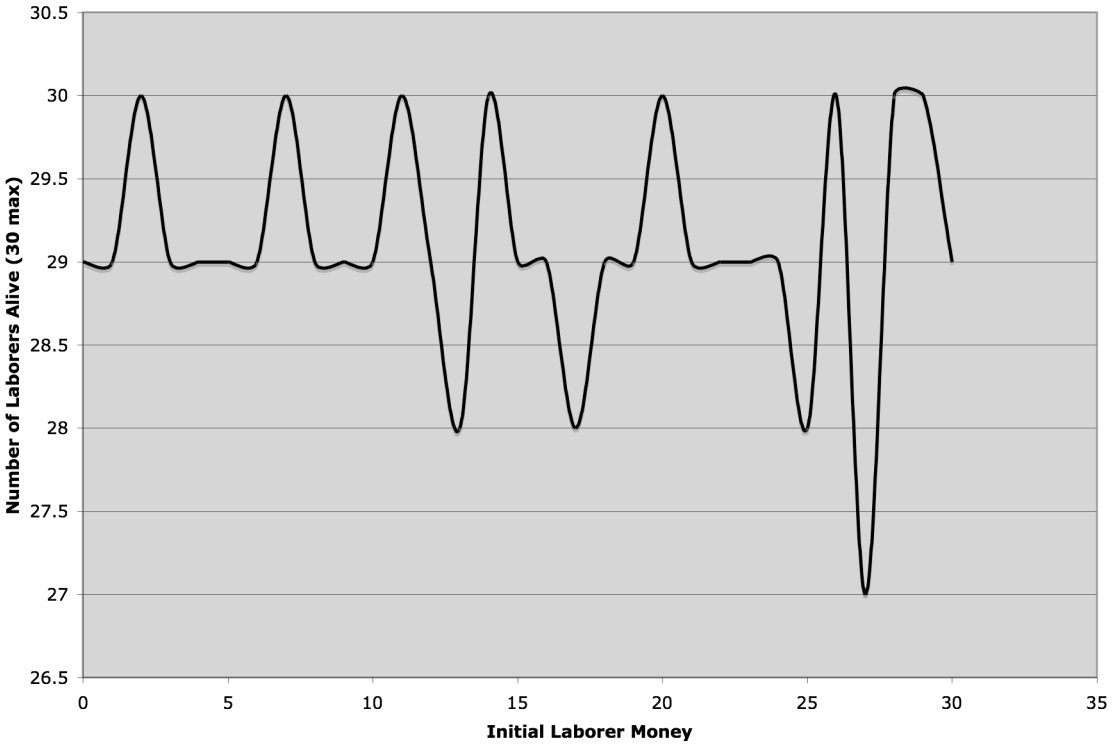
As can be seen clearly there are fairly wild oscillations, but the prices of both food and labor seem to remain stable, and most importantly, all the laborers survived through the whole simulation. Having achieved our intermediate goal of surviving laborers in a relatively stable economy, it was time to test the limits of how stable the simulation was when parameters given to it were altered.

Experimentation

There are many parameters implicit in the previous discussion, and before we start tweaking them, it will be useful to state fully the initial set of parameters from which we are deviating. In the previous instance of the most recent baseline, there were 30 BudgetLaborers and 5 DemandCurveFarms that each started off with 15 units of money and 15 units of food. The BudgetLaborers felt comfortable if they had 15 food on hand or more, so they each started with a comfortable amount of food but were motivated to buy food almost as soon as the simulation started. The exponential smoothing for both the laborers and farms was such that (new predicted price) = (alpha) * (current market price) + (1 - alpha) * (old predicted value), where alpha was 0.1. The production function for the farm was (food) = min(2.5 * labor, 40). All other strategies for bidding for both the laborers and farms is the same as discussed before and for the proceeding experiments, only BudgetLaborers and DemandCurveFarms are used unless otherwise noted.

Probably the most obvious question to ask about a simulation such as this is how changing the initial distribution of money affects the outcome of the simulation. Ideally, it should be the case that a wide array of initial distributions of money should produce similar results, since a simulation that collapses on all but only a few specific sets of

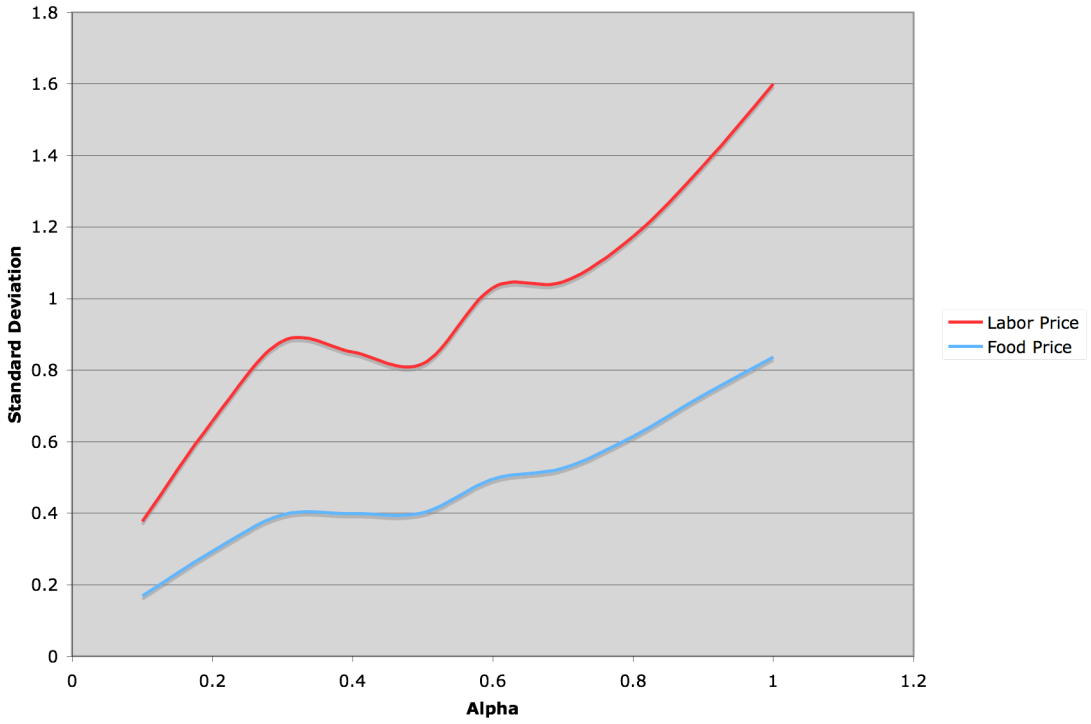
parameters is not terribly useful. To test how our simulation would fare, I altered the starting values of money gradually such that the sum of the money that each laborer and each farm began with always summed to 30. So, if we represent starting values for money as (money for each farm, money for each laborer), I tested all the cases in the series (0, 30), (1, 29), … , (30, 0) over a series of 10,000 timesteps each. The results are below for the number of agents alive at the end of each simulation:



As the graph shows, the initial starting values for money have little effect on the final outcome of the simulation. This may be a little surprising, but it's important to remember that even when laborers began with no money, they still had 15 food and could work on farms to earn money quickly. A few laborers did die every now and then, but two or three agents dying over the course of 10,000 timesteps doesn't seem symptomatic of any deep problem and is more likely the consequence of the random guessing at prices that occurs

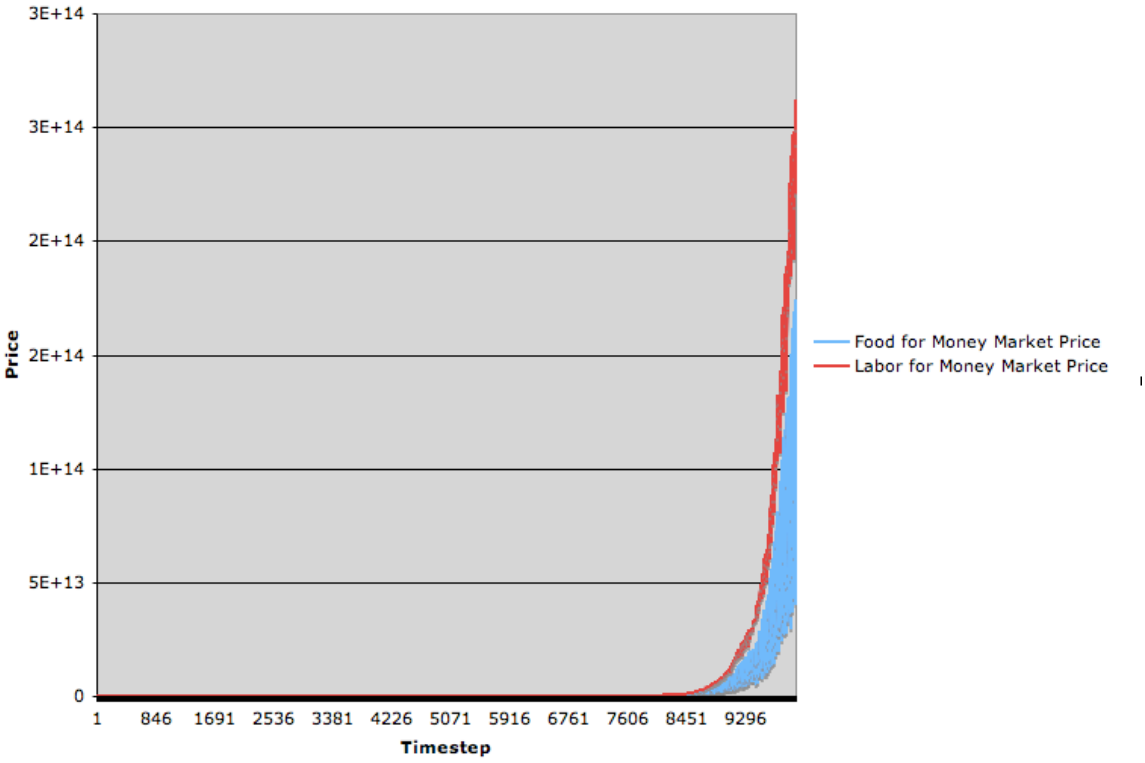
at the beginning of the simulation before any market prices are available. Therefore, this experiment yielded positive results overall in terms of our simulation behaving as it would be expected to.

Another interesting parameter to tweak is the alpha used in the exponential smoothing for the agents in the simulation. Conceptually alpha is the weight that the agent places on the most recent market price when trying to estimate the market price for the next timestep. In this sense agents with a low value of alpha attribute less weight to the current market price and therefore "remember" more of the past market values. As alluded to earlier, we expect agents who remember more to have a stabilizing effect on prices, even to the point of saving the economy from collapsing and all the laborers dying. To test this, I ranged alpha for all the laborers in the simulation from 0.1 to 1.0 in increments of 0.1 for 100,000 timesteps and noted the stabilizing effects of each value. The results are below:



Clearly, laborers with a longer memory tend to stabilize prices better since in simulations with laborers with lower values of alpha, the prices of both goods tend to vary less. In fact, when alpha was 1, all the laborers in the simulation ended up dying at around the $70{,}000^{th}$ timestep (the standard deviation for that data point was taken over the time when at least one laborer was alive). This experiment quantitatively showed that granting agents a memory of previous prices tends to stabilize prices, thereby keeping agents alive that otherwise would have died.

Having examined two experiments that present positive, expected results from our model, we now examine two experiments that reveal a few unintentional consequences of how our baseline model was implemented, beginning with a look at inflation. In our baseline model, there was always a fixed amount of money that was the sum of all money granted to each agent at the beginning of the simulation. I wanted to see what would happen if we granted the farms additional money (this could be thought of as an unsolicited farm bailout funded by a government printing money) on each timestep; would the laborers die because of their disadvantaged economic simulation? At the very least, we would expect prices to rise. This last observation may seem obvious, but recall that this is something that did not happen in MinSim, and definitely should have. I altered the simulation so that every turn, the money that each farm possessed *doubled*. The results of 10,000 timesteps are below:

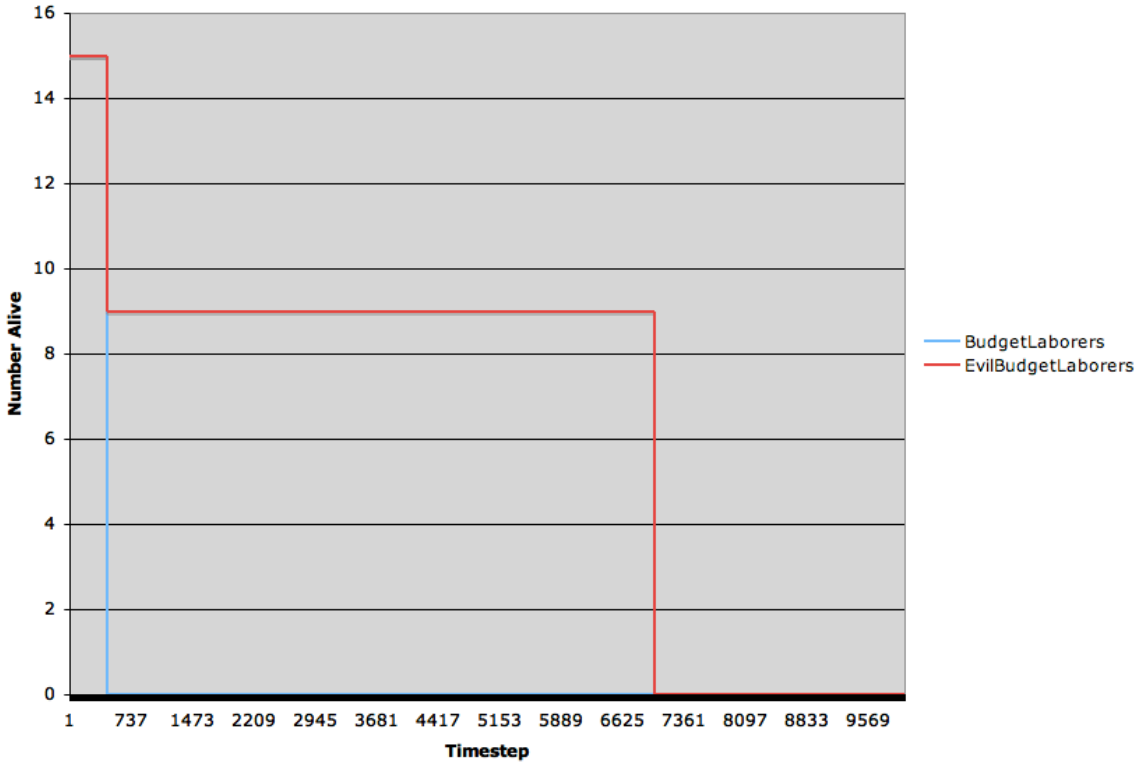


As expected, the prices follow an exponential increase, following the roughly exponential increase of money in the simulation every timestep. What is not shown in the graph and what was not expected at all is that *every laborer survived*.

My first inclination was that the laborers survived because although money was being injected into the economy at an obscene rate, the exponential smoothing that both the farms and laborers were using would dampen the effects to the point that it would be negligible. However, as the graph above makes clear, the effects of the inflation are not dampened, but merely delayed, so the reason for the indestructible agents must lie somewhere else. Therefore, I examined more closely the behavior of our agents in the model to try and explain these results.

The one major change we made to stabilize our baseline simulation was to make both farms and laborers benevolent to laborers. Farms had sales on food, and laborers

would sell their labor for more if they didn't need money, allowing more desperate laborers to be hired first. I wanted to test just how much of an impact this had on the stability of our baseline simulation, and to test it I created a variation on BudgetLaborer named EvilBudgetLaborer, which was the same as the old BudgetLaborer except for one line of code. When an EvilBudgetLaborer had more than a comfortable amount of food, it *decreased* the price at which it sold its labor, undercutting potentially desperate other laborers. My prediction was that if I put these two types of laborers into a simulation together, then eventually all the BudgetLaborers would die at the hands of the predatory EvilBudgetLaborers. I ran a simulation with 15 of each type of laborer and 5 farms; the results are below:



As predicted, the BudgetLaborers eventually all die. What is surprising though is that so many laborers all die over the course of just a few timesteps both when all the

BudgetLaborers die and when the EvilBudgetLaborers die a few thousand timesteps later. We noted similar behavior in our previous unsuccessful models. What happens is that since all the laborers behave in a similar manner, they each slowly become poorer at about the same rate, and by the time a few of them die, the others are so poor that they cannot recover and die as well. In this particular simulation, it appears that the EvilBudgetLaborers' predatory pricing undercut all laborers in the simulation, first eliminating the benevolent BudgetLaborers and a few unfortunate EvilBudgetlaborers and finally driving the remaining EvilBudgetLaborers to die as well.

The preceding experiments validate certain aspects of our baseline implementation and raise a few questions as well. Our baseline simulation is robust against a fairly wide range of initial conditions. Exponential smoothing works as intended and is a fair way to grant agents the ability to remember. However, the inherent benevolence of our agents perhaps plays too big a role in the stability of our simulation (for instance, in helping to weather absurd rates of inflation). On the other hand, should we really expect a simulation populated by uncooperative agents to flourish? In the real world, people definitely look out for themselves, but they also look for opportunities to mutually benefit from cooperation. It is a complicated question, but it is worth noting that if a simulation produces only expected results, then there is really no need for the simulation in the first place. In the end, the only way to say for certain whether unexpected results are incorrect is to examine the assumptions that the model makes in terms of behavior of the agents and see if they agree with the behavior of people in the microcosm modeled in the simulation. Since we weren't modeling any particular situation in the real world with our baseline model, it is unclear how correct our baseline

model was, but it proved what it set out to prove, that our modeling framework can support an economy with fairly complicated behavior among multiple agents.

Conclusions

The main purpose of this project was to improve upon the MinSim modeling framework in terms of simplicity, extensibility, and correctness. Where MinSim fell short in terms of providing easy avenues for extensions, EOS seems to excel. Whereas MinSim began with the gold-food model and was shoehorned around the idea of expanding in other directions, EOS was designed from the beginning with that explicit end in mind. It's hierarchical taxonomy of agents is designed to make changes as easy as possible, and the ease with which I was able to run the experiments at the end is a testament to this. The results of these experiments show that the EOS framework is capable of supporting an interesting economy that generates questions about the interactions between agents within that economy, and the possibility of extending the model to incorporate more complex institutions and economic agreements is readily supported. Finally, it is worth noting that while EOS purports to eventually be a very general framework on which most economies can be modeled, no model is a perfect model of life. In the words of George E. P. Box, "All models are wrong, but some models are useful." It is our hope that EOS will continue to be a member of the latter group as it is expanded and revised in the future.

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

Works Cited

Adelson, Michael. "Ajent Documentation." Unpublished Documentation, 1 May 2009.

Chan, Christopher K. "An Agent-Based Model of a Minimal Economy." Paper, 5 May 2008. 4 May 2009 <http://minsim.cs.princeton.edu/resources/ChrisChan_IW_2008_Part2.pdf>.

- - -. "A Java Library Implementation of the Gold-Food Economic Model in Repast and MASON." Paper, 7 Jan. 2008. 4 May 2009 <http://minsim.cs.princeton.edu/resources/ChrisChan_IW_2008_Part1.pdf>.

Hayes-Patterson, Daniel. "Introducing Traders to an Agent-Based Minimal Economy: Creating a Platform for Collaborative Research in Economic Agent-Based Simulation." Paper, 5 Jan. 2009. 4 May 2009 <http://minsim.cs.princeton.edu/resources/dhayesThesisFINAL.doc>.

LeBaron, Blake, and Leigh Tesfatsion. "Modeling Macroeconomies as Open-Ended Dynamic Systems of Interacting Agents." Paper, May 2008. 4 May 2009 <http://www.econ.iastate.edu/tesfatsi/AEAPP2008.LeBaronTesfatsion.ACEMacroModeling.Final.pdf>.

Macal, Charles M., and Michael J. North. "Tutorial on Agent-Based Modeling and Simulation Part 2: How to Model with Agents." Paper, 2006. 4 May 2009 <http://www.informs-sim.org/wsc06papers/008.pdf>.

Tesfatsion, Leigh. Agent-Based Computational Economics. 4 May 2009 <http://www.econ.iastate.edu/tesfatsi/ace.htm>.

Vreeland, Eric. "Stabilization of an Agent-Based Minimal Economy: Lifecycles?!?...Why Won't They Just Die!" Paper, 13 Jan. 2009.

Notes

1) “A Java Library Implementation of the Gold-Food Economic Model”

2) “A Java Library Implementation of the Gold-Food Economic Model”, p. 14

3) “An Agent-Based Model of a Minimal Economy”

4) “Introducing Traders to an Agent-Based Minimal Economy”

5) “Stabilization of an Agent-Based Minimal Economy”

6) “Stabilization of an Agent-Based Minimal Economy”, p. 5

7) “Stabilization of an Agent-Based Minimal Economy”, p. 7

8) “Stabilization of an Agent-Based Minimal Economy”, p. 7

9) “Tutorial on Agent-Based Modeling and Simulation Part 2”, p. 5

10) “Agent-Based Computational Economics”

11) “Modeling Macroeconomies as Open-Ended Dynamic Systems of Interacting Agents”, p. 4

12) “Ajent Documentation”

13) “Ajent Documentation”

14) “A Java Library Implementation of the Gold-Food Economic Model”, p. 14

15) “A Java Library Implementation of the Gold-Food Economic Model”, p. 7

Appendix: An Breakdown of Individual Work Done

For the explicit purpose of grading this paper as a submission for independent work, it will be useful to break down which parts of the project were done by which people, at least to the extent to which the work is divisible. I will note that this account is of course from my perspective, and it is entirely possible that I've omitted some of the work of the other members of the group that didn't make it into the final version of the code.

All three of us met with Professor Steiglitz to come up with a basic framework for EOS. Cody then created a partial mock up in Java of classes we decided on with the intent of creating a working baseline very quickly. This proved to be rather hasty, overly optimistic evaluation of the difficulty of the task. When it was clear that we would have to spend more time fleshing out the model before implementing it, Michael and I conceptually refined the previous model; in particular we made the decisions about where to enforce eat()ing and the like in agents (at this point Cody had not yet returned from spring break). Then, Michael wrote up the Simulation, Economy, and Market framework classes while I wrote up the Agent, Firm, Farm, and Laborer framework classes. After that, Michael wrote up the implementations for the CallAuctionMarket, the baselineSimulation and the baselineEconomy while Cody and I tried to put together agents that would function correctly in the simulation.

After I got SimpleLaborer and SimpleFarm working, the three of us tried to come up with more sophisticated implementations that provided similar results. A lot of time was spent in meetings discussing the best approaches, and I can't really speak for what specific code people wrote besides myself and the final BudgetLaborer and

DemandFarmCurve that Michael implemented. In my search for stability, I wrote SimplePredictiveFarm, PredictiveSalesFarm, SimpleSaleLaborer, TheoryLaborer, and the following two classes: TieredRucinskiLaborer and TieredRucinskiExponentialLaborer, which were based on an implementation that Michael had at one point that I've renamed TieredAdelsonLaborer to avoid confusion. Aside from that I know that Cody wrote up TheoryFarm. All these classes can be found on the CD that accompanies this paper. After we settled on BudgetLaborer and DemandCurveFarm, I know that Cody and Michael experimented a little bit with traders, though I'm not sure how complicated those got. Also, throughout this process, Cody had been maintaining an SVN repository to organize all these different files as well.

There were many more informal trials that never made it into formal classes because the changes from previous implementations were so minute or the changes didn't help the behavior of the simulation in any tangible way. Also, most if not all of the files on the CD associated with this paper should have comments in the initial comment section denoting who worked on which files. Finally I will note that a very sizeable portion of the work spent on this project was done on blackboards in meetings, and it's very difficult to break that sort of work down since there were no persistent, tangible, direct results from those discussions.