Genetic Algorithms in Agent Based Computational Economics

by

Nathan Forczyk

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 3, 2014

Keywords: Genetic Algorithms, Agent Based Modeling, Strategic Information Transmission, Trade Topography

Approved by

Henry Thompson, Chair, Professor of Economics
Richard Beil, Associate Professor of Economics
Tannista Banerjee, Assistant Professor of Economics
Al Wilhite, Professor of Economics

Abstract

Genetic algorithms are commonly used in agent based computational economics. In this dissertation they are examined in the context of agent based modeling and economic theory and applications. The drawbacks and strengths of this approach are discussed and two applications are presented. Each application makes use of genetic programming in a different way and the benefits of each are examined so practitioners can decide which is preferred.

Acknowledgments

None of my graduate studies would have been possible without the help Auburn Economics Department. The faculty has been nothing but supportive. I have always been a gambler and coming to Auburn was one of the biggest gambles of my life. Every gambler knows that the outcome is left to chance. This gamble mainly worked out because of the faculty at Auburn.

In particular Henry Thompson and Richard Beil have been the difference between success and failure. Al Wilhite and Tannista Banerjee have also taken chances to help me out at critical moments that have pushed this over the top. Life is full of people that default to no and to find helpful people is rare.

I also am grateful for the faculty at Auburn University Montgomery for my first full time teaching job. I had little knowledge of the other aspects of academia until they gave me my chance for two and a half years.

Table of Contents

List of Figures

List of Abbreviations

ABM	Agent Based Modeling
ACE	Agent-based Computational Economics
ESS	Evolutionary Stable State
GA	Genetic Algorithm
GP	Genetic Programming
MABS	Multi-Agent Based Simulation
OOP	Object-Oriented Programming
OLG	Overlapping Generation
RPS	Rock Paper Scissors

Chapter 1

Introduction

Social science lacks controlled experiments that the hard sciences depend on. To explore some of the issues in economics would be cost prohibitive. No one would willingly submit to living a life of poverty to see if in fact it affects their children's educational attainment. Voters of an experimental country would not destroy their institutions to see how it affects long run growth compared to a control. One method of controlled simulated experiments that has gained some traction is agent based modeling.

Agent basd modeling attempts to reconstruct a larger picture by setting the rules on how individual actors interact. The common way to do this is to create a computer program, but the beginnings of this field involved coin flipping. As it is a relative new comer to the social science toolset, the rules and methods are evolving. Chapter 2 summarizes the major points of research on ABM, the types of learning, and how they relate to economic theory.

The emphasis is genetic algorithms. Issues, problems, and some commentary on where the field is heading is in the first essay. Technical aspects such as computer programming issues are also discussed.

An issue in the field of agent based modeling is the "black box." It is hard for outsiders to see what is going on internally. Algebraic solving of regression equations is standard for economists so there is widespread understanding. Agent based modeling problems cannot be solved algebraically and involve a probabilistic element.

Chapters 3 and 4 contain two vastly different models each with a genetic algorithm. Each is programmed with a different programming language. Chapter 3 is a model exploring the evolutionary game aspects of lying, honesty, and punishment. It is programmed in R, a common

statistical language. A major benefit of this approach is that the code is easier to read and more known to economists as it is often used for empirical work. It is easier to make graphs and movies of the output with a language like R that contains abilities for mathematical operations that would be prohibitive for an academic to reinvent in another language. A drawback is the execution time of R is significantly slower as it is not a compiled language.

Chapter 4 builds upon prior work in trading between agents in different topographies. Where this essay differs is its usage of a genetic algorithm. In prior work, the number of trading partners is predetermined, but with a genetic algorithm the model determines the optimal number of trading partners. The implementation of a genetic algorithm is done with a customized application framework written in Objective-C. A major advantage of this method is the speed of execution. However, it is more difficult to verify that the algorithm is working and to make the results accessible to others.

Extensive debugging is necessary in both instances. Construction of any software involves finding errors. To my best knowledge, there are no errors in the computer code. The technical appendices contain snippets of code and output to assure that the genetic algorithm in each worked properly. During testing, the model was spot checked by hand to ensure proper functioning.

R has never been used for genetic algorithms, and it could bridge a significant gap in software approaches to agent based modeling. The rule of thumb in existing software packages is that ease of programming means giving up some power. For instance some learning packages are easy to learn but limit the modeler in some form or another. With R the power of MATLAB like abilities is combined with its free price and a more sandbox like environment.

The last chapter summarizes the benefits and cost of each approach. A lengthy technical appendix contains all the code in R and Objective-C that was used to create these models. It also contains a more technical discussion of the details of the models and the programming involved.

Chapter 2

A Literature Review of Agent Based Modeling

"(A theory) is not a collection of assertions about the behavior of the actual economy but rather and explicit set of instructions for building a parallel or analog system" – Robert Lucas (Methods and problems in business cycle theory. In Lucas, R.E. Jr., ed., *Studies in Business-Cycle Theory* (pp. 271-296). Cambridge, MA: The MIT Press.

Economic theorists think in terms of individuals being "agents" in an economy, governed by internal rules of behavior in their interactions with others in society. Adam Smith wrote of economic growth coming from individual actors specializing and dividing labor tasks. The "invisible hand" was the net result of agents acting based upon their own rules, resulting in societal wide improvement.

The first mathematical models of economics came in terms of explicit individual rules of behavior. Utility maximization is agent behavior, and the net effect of an economy with agents guided by this principle is the market demand function. Agent rules of behavior form the structure of markets, rather than a top down imposed rule.

Economic modeling built on this approach is *agent based modeling* (ABM). In short, ABM models involve a population of "agents" that populate an environment. Rules describe how they interact with their environment and with each other. In each cycle, agent traits and the environment is updated.

While ABM is performed with computers, an early simulation was done by hand. Thomas Schelling (1969) showed how spatial racial housing patterns could arise out of different preferences for being among their own or another race. His method involved flipping coins and assigning probabilities of different agent preferences for being among their own race. By experimenting with different probabilities, he offered an alternative explanation to segregated neighborhoods than mere top down imposition of racial separation. In fact, one possible outcome is that minorities may be "rationed" among a majority in that there would be small clusters of minorities.

ABM is similar to "cellular automata," a field in computer science. Cellular automata involves simple rules of patterns propagation at a small level to show how over time it can lead to complex structures with fractal characteristics. Complexity can arise out of simple rules.

However, what is different from simple cellular automata is the probabilistic element or Markov chain element of agent based modeling. A simulation is necessary because the state of the environment and population at any time *t* is unpredictable. In Schelling (1969) the probabilistic element was the different chances of relocating and staying based upon the racial makeup of a neighborhood.

These models may appear to be of little real world consequence, but automated agents that can act economically are now populating the internet (Kephart, Hanson, and Greenwald, 2000). Many financial decisions require quick action that would be slowed by human action, calling for automated agents. A mistake can cost millions of dollars if an automated agent makes a wrong buying or selling decision on the internet.

Agent based methods and game theory become intertwined in areas when the agents are set

in a competitive environment and are allowed to adjust strategies. Many techniques allow virtual agents to learn over the course of a simulation. These methods were developed when computing power was expensive but over time ABM will be limited less by hardware and more by modeler ability.

In other fields, computer simulations have filled areas left by hard evidence. For instance, in biology a simulation on evolution of the eye showed it could evolve in 300,000 years (Nilsson 1994). In astronomy, simulations of globular clusters are a cottage industry that shows traits unobservable with other scientific instruments.

What follows is a review of the history, methods, and directions of agent based methodology and findings in economics and closely related social sciences. What started as several independent researchers occasionally working on a model has turned into a interconnected field that is building rules and standards.

2.1 The Model

Economists put forth equations such as the idea of an equilibrium, but admit that no one believes the equilibrium instantaneously arrives. The market price and output decisions are not found *a priori* through producers knowing the demand function and others' output decisions, but is found through a repeated series of trial and error with limited information. These repeated trials naturally lend themselves to agent based models.

A *single agent simulation* seeks to model an environment with one actor. Economic and social systems, however are defined by interacting agents. Single agent approaches are used in areas like engineering, where for instance the behavior of a jet airliner in turbulence is simulated in a computer.

Multi Agent Based Simulation (MABS) involves multiple agents, seeking to determine the behavior of each. Every ABM in economics is a MABS. When learning is introduced care must be used in simulation design so that the modeler is aware of the type of population. The *genetic algorithm* method of learning can appear to have hundreds of agents in a simulation, but it may be solving only one agent's optimal behavior.

The beginnings of these methods cannot be traced to one pivotal development, but rather to a series of incremental advances to the methodology. While Schelling's (1969) neighborhood simulation was done by hand, future experiments were done with computers. A convenient beginning could be placed at the time of Axelrod and Hamilton (1981).

In the late 1970s Axelrod and Hamilton invited anyone to submit a program to play an iterated prisoners dilemma game with other submitted programs. The result was that "tit-for-tat" won two tournaments. Tit for tat involves beginning with cooperation and then picking the opponent's last move in following rounds. This tournament caused some discussion, and eventually a strategy was found that could beat tit-for-tat (Beaufils, Delahaye and Mathieu 1998).

The 1970s also saw the development of "genetic programming" (Holland 1975). The genetic algorithm is a special case of genetic programming. Its application to economic agent based simulations would take some time. Genetic programming mimics natural systems and market systems as well.

In biological systems, the environment consists of nature and other "agents" or organisms. Each organism seeks to find an optimal inventory and distribution of physical and mental attributes. However, the agents have an informational problem in that they know nothing about

the environment in which they will be placed or how the environment will evolve.

An optimal or near optimal set of characteristics in any environment is more likely to be passed on to the next generation while the more detrimental sets will result in little to no reproduction. In this manner given all else equal, an organism can find an optimal solution. This optimizing approach forms the basis of genetic programming.

In addition to the mechanism of reproduction, other traits of sexual reproduction can be included. Crossover involves switching pieces of genetic code between organisms to create new instructions. Mutation randomly switches a small portion of some agent's codes. Some practitioners also introduce agents with random traits.

The net effect is that genetic programming is a simple yet powerful method for searching for optimal solutions. If the population arrives at a local but not global optimal, mutation and crossover can push it towards the global optimal. The difficulty comes from framing a problem that can be solved by this method. Successful applications of ABMs with and without a genetic algorithm have explained things in economics that other approaches could not.

Gintis (2007) showed an interesting difference between agent based and traditional approaches. Scarf economies (Scarf 1960) involve models based on Walsrian equilibrium in which there are multiple goods and the excess demand in the system sums to zero. Two goods can have non zero excess demand. Prices between goods are determined by relative excess demands.

Prior work on the global stability of prices in such a system found multiple equilibria. With the use of an agent based approach, Gintis (2007) found that with private prices the global equilibrium is stable. Private prices refers to agents not announcing their reservation prices and

updating them privately over the course of the simulation. Relaxing the private price aspect even slightly results in more chaotic characteristics of equilibrium prices.

Gintis (2007) did not focus on formation of prices alone in an economy but characteristics of the entire economy as a whole. Capital and labor interact to form demand functions for each product in addition to measures of efficiency. Macroeconomic shocks such as increases in labor supply or optimal firm size affect the unemployment rate and prices in the economy.

In some markets, non price considerations affect outcomes. For example, the Marseilles fish market exhibits high price dispersion that under arbitrage assumptions should disappear. Kirman and Vriend (2000, 2001) explore the possible explanations for empirical facts gathered from a detailed data set of this market. It is a unique market in that strategic decisions are limited by the perishability of the product, its organization, and the time it is open (2 am to 6 am daily). Buyers and sellers also meet face to face, and because of the limited number of buyers and sellers, interpersonal relationships matter in a way that is hard to capture.

Prices in the Marseilles fish market vary between sellers. In addition, a seller may offer a different price to different buyers for the same type of fish. Seller offers are final and there is no bargaining. Each morning the market clears in that there is not much fish thrown away. The persistent price dispersion is an odd trait given that the switching and search costs are negligible.

Standard theory explains this through information asymmetry. If buyers are not able to determine quality before purchase, there is an incentive to deliver good quality to maintain loyal customers. However, with the small size of the market Kirman and Vriend (2001) note that every buyer is a potential loyal customer, but they cannot afford to sell high quality to everyone. Either loyal buyers are inherently loyal, which would result in no incentive to offer them high

quality, or buyers are inherently disloyal, which again would result in no incentive to offer high quality. Another theory is that the buyers and sellers enter into an implicit contract, but the buyer demand and supply fluctuate too much for this to be the case. Complicating the analysis of the data set they examined was the inability to determine quality differences.

Instead of empirical statistical analysis, Kirman and Vriend (2001) opted for an ABM approach with a classifier system that fully encompassed the specific decisions that each buyer and seller agent has to make in this market. They find that the determining factor of buyer and seller loyalty is that loyal buyers have a lower rejection rate when visiting previous selling partners. This leads sellers to offer better deals to loyal buyers which causes the cycle to repeat.

The importance of social contacts in market decisions is also a theme in Rouchier, Bousquet, Requier-Desjardins, and Antona (2001) similar to Kirman and Vriend (2001) except that it deals with cattle herders, farmers, and villagers in North Cameroon. Another difference is they use a more simplistic rule based method of agent learning instead of a classifier system.

Each dry season, cattle herders must leave their homes to find water for their herd. These herdsmen have to travel on others' property. The farmers and villagers that decide whether to allow them to cross face a decision. The farmers and villagers can focus on "cost priority" or "friend priority."

Cost priority involves agents making decisions based solely on lowest cost provider of watering hole services. Friend priority means that herders will revisit farmers and villages that accepted most of their propositions. The results of the model were focused on global herd size, the number of encounters, and the difference between the largest and smallest herd. The friend priority model resulted in larger difference between the largest and smallest herd and a larger number of animals overall.

Another contradiction between traditional economic results and ABM results is nonbinding price controls. In a Walrasian equilibrium, setting a price ceiling above the equilibrium does not affect allocative efficiency. However, Gode and Sunder (2004) showed that in auction dynamics with non binding price controls allocative efficiency is aversely effected. It causes a small 1-3% decrease in efficiency because in the early rounds of trading, the buyers that are willing to pay more cannot meet up with sellers that will only sell at a higher price.

2.2 Areas Where ABM is Appropriate

Much of the literature on agent based methods is focused on the mechanisms of simulation and not the issue of when to use agent based models. The general rule of thumb in the field is that the models should be used when other models fail or are inferior. Whether an agent based model is valid is different from whether or not it is a significant contribution to economics.

A particular area where ABM's have a significant advantage over other approaches is when the distribution of the variable of interest plays a major role as in Borill and Tesfatsion, (2011). Econometric techniques do well with normally distributed variables, an assumption of OLS. Theory is more general. Agent based methods make no assumptions and can be modified to accommodate irregular distributions.

Major drawbacks of ABM's are the lack of data and the perceived notion that there is little connection between the model and reality. Successful models will strive to design a model that as closely resembles reality as much as possible as described by prior work and data gathering. They will strive to minimize the part that can not be verified, and often try to make that the research question.

For example, we know the statistical characteristics of stock market returns (Pagan 1996). What is not known is the individual behavior of traders that lead to the statistical traits of these returns. ABM experiments by Lettau (1997) and LeBaron et al (1999) showed possible individual behavior of traders that yield those asset return characteristics.

2.3 The Methodology

An agent based model starts with a written description of how the researcher thinks an area of interest works. It begins with the agents, the environment, and how they will interact. They each begin with an information set, which may or may not be updated, and a set of traits and behaviors. At each time period, called a tick, the simulation is in state S. At the end of the time period, the simulation is updated based upon programmed rules.

If the simulation involves probability, it will be a Markov process (Gintis 2012). Learning also implies a Bayesian element with priors (Hommes 2006). Each agent begins with a random rule set and beliefs about its environment. These rules can change over the course of the model as more information about the environment and other agents is found. However, the choice of learning and which process of change it uses needs to be rooted in economic theory. Some methods of learning violate basic economic assumptions.

For instance, economic models that are founded on zero information (ZI) actors need to use methods of updating and learning that do not violate this assumption. In this analogy, if we assume two chess players with no knowledge of the rules of chess, employing a super computer that can use perfect foresight to model the players moves would be meaningless. Similarly assuming a stock trader has knowledge of the other traders' characteristics and optimal moves would not yield realistic results. However, ABM can accommodate models where different

agents have different levels of information.

Articles in the field have extensive theory and mathematical manipulation exactly like any other econometrics based paper. For example the models dealing with stock markets and foreign exchange markets (Ehrentreich 2006, Lux and Schornstein 2005, Yeh and Yang 2010, Lettau 1997) are explicit about the theoretical optimal behavior of individual traders in the market, and seek to explore if individual artificial agents can reach that behavior through individual utility maximization and the proposed rules of agent interaction.

2.4 Methods of Learning

There is no "one correct way" to implement learning in adaptive agents. However, often the choice of methods are limited by economic theory. As mentioned, ZI actors cannot learn based upon information that economic theory prohibits them from knowing. Additional, computational limits have historically made methods such as "Q Learning" practical in only limited situations.

Much has been done with simple rules. Schelling's (1969) model explored variations of simple rules on neighborhood racial makeup outcomes. One rule was that people prefer to live amongst their own racial group, and this led to one outcome - segregated neighborhoods An adjustment to this rule was to see what would happen if there was a slight preference for a minority group amongst the majority, and this led to "rationing" of the minority group amongst the majority.

Another example of simple rule based behavior is Wilhite (2001) and Wilhite and Allen (2008). The former studied the effects of trade topography on transaction cost, and at its core was a decision was whether or not an agent would trade with another agent based upon differing

MRS. The latter involved a virtual urban environment in which agents made decisions about switching criminal status, changing level of protection (both private and public protection), moving, or doing nothing.

Both of these models are rule based, but vary in the different types of rules. Wilhite (2001) isolated the effect of different trade topographies on search costs and trades in each type. Agents were randomly endowed with two goods and had identical utility functions. A search algorithm that depended on the type of trade network – globally interconnected, local, or local connected – determined possible trading partners. The outcome of interest was dispersion prices and how long it took to reach price stability. The global trade network in which all agents could trade reached equilibrium fastest but at a high search cost. The local network in which only a few small groups of agents were connected had the highest price dispersion but least searches.

Wilhite and Allen (2008) is another rule based model but is more complex. The ABM there explores crime prevention and neighborhood characteristics. It makes usage of "tags" that describe neighborhood and agent characteristics. Agents have a choice to contribute more or less to crime protection, become a criminal, or move to a different neighborhood. The chossen parameters result in a crime participation rate comparable to several US cities. Results from the model were analyzed with econometric techniques showing the effects of the model on criminal participation and agent well being. An implication of that study is that putting more people into prison may lead to a long run increase in crime rates.

A simple rule based agent model does not mean simplicity however. Rouchier et al (2001) explore the complex social relationships between nomadic herdsmen, villagers, and farmers. At issue is whether herdsmen give preference for "friends" or merely seek to keep costs low. In

effect this is similar to Kirman and Vriend (2000) in exploring the social aspect of markets. Both found that participants giving priority to friends and being loyal customers despite short term inefficiencies resulted in longer term gains.

Genetic programming (GP) is a broad term that involves a mechanism of reproduction of successful rules, mutating existing ones, and crossing over with other agents to form new rules. There is no hard rule as to how to implement; the agents in Wilhite and Allen (2008) have some implicit genetic components such as randomness and allowing the least satisfied agents to change their status.

Use of a "classifier system" (Booker 1989) with genetic programming is a good example of a less commonly used GP technique in economics. For example, an agent may have a rule set that may be encoded as a "trinary string" of length four. An example could be "01##" Each position of the string could indicate a behavior action encoded in the program, and whether or not it is undertaken. A zero would indicate no, a one would mean yes, and # could mean it does not matter.

In economics, however, it is the "genetic algorithm" (GA), a special case of GP that is more popular. In effect the GA is a search algorithm limited by the modeler and the length of the string size. A GA is binary system of 0's and 1's. Instead of representing a complex set of rules, it only represents the value of a parameter. For instance, a GA could be used to find the optimal mix of stocks and bonds where the GA would represent the percentage of stocks and bonds.

Each GA string is a string of length *l,* and if the percentage of bonds were represented as a string of length 5, interpretation would be as follows. In the decimal system, each position in a number represents the values of ten to a certain power. So, the number 950 means zero values at

10^0, 5 10^1's, and 9 10^2's. A binary system follows the same pattern except each position is base 2. For example, the following is a binary string "10011" is translated from right to left as: $1*2^0 + 1* 2^1 + 0*2^2 + 0*2^3 + 1*2^4 = 19$.

Each string of length n has $2^n - 1$ possible values. The use of binary strings makes the genetic processes easier and quicker. However, a drawback is the number "19" by itself means little. For instance if the string is encoding an agent strategy in a game, it is unlikely 19 can directly relate to anything. To make it have meaning, the value has to be normalized.

Normalization is accomplished by dividing by the maximum value of the string. A string can be any length, and for any binary string of length N, the maximum number of combinations is 2^N. However, one of those values is a zero, so the largest number the string can represent is thus $2^N - 1$. The above string is length 5, so the maximum value is $2^5 - 1$, or 31. Normalization would lead to 19/31 ~ .61. This could now represent spending 61% on bonds.

Genetic programming works best when the agent rules are not fixed but can adapt. Each string (whether a binary, trinary, or so on) has a *fitness* associated with it. Each turn of a model is called a *tick*, and after a specified number of ticks, genetic learning is applied. Again there are no accepted specific rules, but all GP models must have certain characteristics.

Upon learning, the agents in a *population* are ranked from most fit to least fit. If the agents are stock and bond traders, the most fit would be the ones that made the most money. Each population is made up of agents that learn from each other, and will trend to one optimized value. Thus if a model has vastly different types of agents, it is necessary to create multiple populations of similarly typed agents that will learn from only each other and arrive at different parameter values. However, most GA's in publication fall under the single or dual population variety.

After ranking, the most fit agents have to be more likely to survive. This can result from a simple rule allowing the top few percent to go on unscathed to the next generation. Another way is to have a probabilistic model where the most fit are more likely to reproduce.

Crossover and mutation are methods to introduce new rules into the system. Two "parent" genomes are selected, and a random point in the string is picked to exchange genetic information beyond that point. For example, two parents with strings of 01110 and 11000 would yield a child of 11110 if the cross over point was the third position. Mutation is implemented as a percentage chance that an agent's code will be mutated, and a random bit position is flipped.

There are may modifications to this general approach. Arifovic (1991) uses an "election" operation in addition to the previous methods. This operation involves testing the child's fitness against its parents', and if one of the parent's fitness is higher than the child, that parent is retained in the population. If the child's fitness is higher than that of both parents then both are replaced by the child.

The genetic algorithm is effectively a search algorithm that properly formulated can seek the optimal solution in X^n possibilities, with X being the number of possibilities that each bit can encode, and n being the length of the string. For instance a binary string of length 8 would have 2^8 possibilities, while a trinary string of the same length would have 3^8 possibilities. The trick is phrasing the research question and defining the problem such that the optimal solution is in one of the possibilities.

An indirect consequence of the genetic algorithm is that every genetic algorithm is also an evolutionary game (Riechmann 2001). The individual agent's fitness depends on its own strategy and the strategies of others. Agents in a genetic algorithm population also "tend to converge towards a Nash equilibrium" (Riechman 2001). However, the clear line only exists for

simple, single population genetic algorithms. Because nearly every implementation of a GA is a single population - only two studies, Atrifovic (1995b) and Birchenhall (1995) are of the two population type - this characterization applies to the bulk the publications in the field.

GA's also tend to go towards an "evolutionary stable state" (ESS), first outlined in Smith (1982). In this state, new strategies will not do as well in the game as the stable state. Generalized evolutionary game theory has no clear cut process for adoption of mutant strategies or rejection of inferior ones (Riechmann 2001), but a GA does. However, the theoretical work of Friedman (1991) is of use in the terminology of evolutionary games. Of particular interest are the relationships of the ESS to the Nash Equilibria (NE) - the NE are always a superset of the stable states. It should be noted that Friedman (1991) provides mathematical solutions to some two or three dimensional games with two players, whereas a agent based model with a GA has no theoretical limit to the dimensionality or the number of players.

A genetically programmed agent has no foresight, and for some purposes, such as agents that make automated bids on auction sites, this could be a problem. Q-learning is one such method that does have foresight. Kephart, Hanson, and Greenwald (2000) demonstrated that in a simulated automated auction environment, a Q-learning agent can outperform myopic agents. However, significant drawbacks make Q-learning merely a novelty for now.

Q-learning works similar to a chess playing artificial intelligence. In the Q model in Kephart et al (2000), the "chess match" was determining how to set prices in a market with another competitor. The real world value of this simulation involves "bots" on the internet that make bids at auction sites. The Q method is to determine the future income stream generated by a pricing decision, given the possibilities it could charge and the competitors current price. A price that maximizes this net present value is the chosen price.

While not difficult to explain, in execution the Q method begins to take prohibitively long as the number of competitors increases. In Kephart, Hanson, and Greenwald (2000), they note that their experiments had only two sellers, and they were beginning to experiment with three. If auction agents are to be used, they will likely come across hundreds, if not thousands of sellers and customers. This would make Q learning unworkable for the time being. Similarly Kutschinski et al (2003) confirmed that Q learning's advantages are mitigated by the time it takes.

To see why, expand the chess metaphor further. Currently computers can beat the top chess players because they can account for all the possibilities in a two player game. But if the game were expanded to a 100 players, the possibilities would explode and if the game were timed, movement choice would take prohibitively long.

Other forms of learning exist as well. Least squares learning applies regression techniques to help influence the next decision. This would model situations in which real life agents take into consideration past values. An example of this is in Marcet and Sargent (1989), in which they used agents with this learning type to model inflation.

Rational expectations modeling of inflation in prior work (Sargent and Wallace, 1981) had shown that rational expectations leads to predictions of stability only at high inflation rates. An agent based model with least squares updating led to different results. With this approach, a low inflation equilibrium was the attractor, and the other state was chaotic.

Finally it must be emphasized that the choice of learning method can alter the outcomes. A good illustration of how different types of learning can change the results is in Vriend (2000). It is not merely the type of learning, but the "level" of learning that also affects the outcome. If

agents learn from the population as a whole, socially optimal outcomes are possible - but population learning can lead to "spite effects." These are actions that hurt the agent, but hurt other agents more. This is because learning methods' fitness is often measured by relative fitness to others. If learning is individual, spite effects are not present.

2.5 Main Avenues of Agent Based Modeling

Agent based modeling has an advantage in models with a Markov process and where hetereogeneity of a constituent population becomes an issue. Some (Borril and Tesfatsion 2010) believed agent based modeling is the best way to approach social sciences in general. Social systems have little top down design and instead rely on agents making decisions in their best interest.

The first priority in this field has been confirming that it in fact have the ability to model real world situations better than other methods, and that the computer agents can mimic humans. One way of confirming this is to do a simultaneous experiment with human and computerized agents. The other is to make a model based upon prior empirical findings. Arthur (1991) discusses the foundation of using both to calibrate adaptive agents in order to better mimic reality. Ideally the agents should pass a Turing test (Turing, 1950), but is only necessary for agents to explain some stylized fact about the particular subject it is modeling.

Duffy (2000) and Pingle (2001) are two studies that use an agent based model and human subjects to strengthen and improve each others' results. Duffy (2000) explores a Kiyotaki Wright (1989) setting to shed light on the "speculation failure" of individuals to hold on to money despite higher returns elsewhere. Already this subject had been studied in a strictly agent based model by Marimon, McGrattan and Sargent (1990), and Başçi (1999), in addition to

human experiments (Duffy 2001).

By having a simultaneous agent based model in addition to designing a human experiment, Duffy (2001) argues that this is possibly a better way to design both experiments. The strictly agent based studies had more inherent coordination than exists in human subjects, and human experiments for their part are costly and do not go for as long. This paper concluded that agent based methods can in fact mimic human behavior, with the implication being that computational agents are cheaper and therefore could be used alone.

Pingle and Tesfatsion (2001) show a slightly different outcome of simultaneous human and agent based experiments. Their focus was on labor markets. The human experiments involved university students playing an iterated prisoner's dilemma game in a computer lab. In each round, a Gale-Shapley (1962) matching process paired potential employers with employees. If employed, both the worker and the employer had a choice to "cooperate" or "defect," with the combined results determining payoffs. If unemployed (or had no workers), there was an unemployment payoff.

Computational agents programmed to play the same game with each other had some similar outcomes. Raising the unemployment payoff increased unemployment in both experiments. However, in the computational experiments, there was far more stability and workers were more likely to end up continually employed with a single employer. The provided explanation is that the human experiments were smaller (had 3 employers and 3 employees as opposed to 12 for the computational agents) and thus the potential cost of rejection was larger (each rejection incurred an "offer cost"). This provided some guide to future human experiments in this area - raising the offer costs may increase long term employment relationships.

Inflation and exchange rates have been studied often in an agent based framework by using overlapping generations (OLG). Lucas (1986) argued for an experiment using OLG to explore the adaptive behavior of such a model and its implications for macro economic indicators. An inherent problem though with an adaptive model of OLG with an infinite time horizon is that it would be difficult to find human subjects willing to participate. This naturally led to models based upon computerized agents.

Arifovic (1995, 1996), Arifovic, Bullard and Duffy (1997), and Marimon et al (1993) are all examples of OLG type agent based experiments. They differ in what macro economic indicator they are seeking to explain. However at the core of each is a computer agent population each living two periods, with some sort of decision to be made at the first period.

Arifovic, Bullard, and Duffy (1997) attempt to explain long run economic growth patterns in human societies. They note that there appear to be two equilibria in long run growth patterns - one in which growth is consistent, and the other in which growth is consistently zero. Using a OLG type agent based model in which agents had to decide how much to invest in education and how much to invest in physical capital, they were able to show the two observed steady states of long run growth patterns. Levels of growth depended primarily on initial levels of education.

Inflation and exchange rates are the subjects of Arifovic (1995, 1996). In the former, the purpose was to compare a genetic algorithm with a least squares learning agent in an OLG context. The genetic algorithm better followed real world fluctuations and human subject experiments. In a similar vein, the latter compared the GA agents with other types of learning in the context of exchange rate dynamics. Genetic agents were better able to shift investments and take advantage of arbitrage, resulting in the observed fluctuation of exchange rates.

Agent based approaches have been better able to explain the dynamics and statistical properties of stock markets as well. Lux and Marchesi (2000) sought to explain the "stylized facts" about the statistical properties of financial markets. These facts came from Pagan (1996), in which financial markets were found to have a unit root, not be a random walk, have stationary returns, have "fat tails," and have volatility clustering.

Instead of beginning with an equilibrium approach, Lux and Marchesi (2000) built an agent model where the trading agents act on rules that are more behavior oriented. Some traders are "chartists," and some are "fundamentalists." The fundamentalists value an asset on its "underlying truths," and act as if the price of an asset will return to its value at some point. Chartists on the other hand believe the activities of others reveal more information and act on market dynamics more than fundamental value. This can lead to a cascade effect. Chartists are also further divided into optimists and pessimists.

The model was able to have its agents decide to switch between chartists and fundamentalists over the course of its duration. The volatility clustering was high when the fraction of chartists was high, and stability returned when the fundamentalists' fraction grew.

Combining the evolutionary game aspect of agent based GA's with auctions have also yielded some important insights into the effects of option rules on outcomes. The Santa Fe Institute did an on going virtual stock market (Arthur et al 1996, LeBaron 2002) and from those experiments, the first commercial product for agent based methods arose. eSnipe is a commercialized auto bidder that came from these early auction experiments, and it places bids at the optimal time - at the last moment.

2.6 Existing Toolsets

Numerous tools exist for agent based modeling, each with their own drawbacks and strengths. There is a tradeoff between a smaller learning curve and power as well. A trend has been for specific toolsets called frameworks, such as the "Trade Network Game" (TNG) that studies formation of trading networks (McFadzean and Tesfatsion 1999). That one requires significant C++ programming knowledge that often is a barrier to entry for most.

In addition to lower level frameworks, there also exists other tools that have a lower barrier to entry and more build in graphical capabilities. Swarm (Minar, Burkhart, Langton, and Askenazi 1996) is a generalized tool for complex adaptive systems that tries to strike a balance between power and built in graphical capabilities. It uses Objective-C. Both the TNG and Swarm make heavy use of object oriented programming (OOP) methods so anyone attempting to learn these will have to understand those techniques.

Both of these are for complex models, but numerous simpler tools for smaller models exist. These are especially useful for instructional purposes in the agent based concepts. For example, StarLogo (Resnick 1996) is a platform specifically designed for educational purposes.

For larger scale purposes, and one that was used in Deissenberg, Van der Hoog, and Dawid (2008) is FLAME - which stands for "Flexible Large-scale Agent Modeling Environment." It was used in that papers 3-year EURACE project which attempted to model the European economy. It is not designed for the typical academic and was intended to be used on a supercomputer. The agent based models of the European economy had millions of interacting agents, each doing their own calculations and exchanging data. An important indication of the

time that takes is the "time per iteration," and on one of their super computers with 9 processors, each iteration took several minutes.

2.7 Future Directions

Borril and Tesfatsion (2011) believe that "Agent-Based Modeling is an alternative and potentially more appropriate form of mathematics for the social sciences." They view that the future of social sciences will be to run computer experiments of proposed theories, and then compare these results with empirical data.

In other fields simulation and computerized experiments are an accepted and well used tool for exploration of theory. A major reason why this may be in its infancy in economics is twofold. In physics, the need to model was seen as a national priority with the simulation of nuclear weapons. Physics modelers were able to use supercomputers at the governments expense. Academic economists cannot afford supercomputers and for most of the development of agent based modeling, computational power was a significant limit. In the 1980s desktop computers rarely had more than a few megabytes of memory and often did not have hard disk drives. This severely limited capabilities. In comparison, a top of the line desktop computer as of 2013 that is within an academic social scientists budget would be the fastest computer in the world in 1990.

A second reason that social science has not adopted these methods is that the computational power required for social systems is enormous. As Deissenberg, Van der Hoog, and Dawid (2008) showed, attempts at modeling macro behavior with micro rules are time consuming even for a super computer of today. Comparatively, many science simulations are not as complex. Humans, firms, and governments have rules and behaviors that bring a

dimensionality to modeling that most sciences do not have. In addition the ability of these agents to make decisions instead of following a preset path adds to this. This dimensionality is why a computer can help design an aircraft but not an economy.

Once computational power increases, new abilities will be added to agent based modeling. In addition to new types of learning such as neural networks and Q - learning, the process of modeling may be reversed. As of today, all models in social sciences are *forward simulations* in computational science jargon. However, in engineering and aerospace, a growing approach is the "inverse" approach (Murray-Smith 2000, Kurahashi et al 1999). The inverse approach has the potential to increase the abilities of ABM.

A forward simulation requires setting rules and parameters, and seeing what happens. The inverse approach is the opposite. A modeler sets the goal of what will happen, and the model adjusts its parameters to find a solution set that satisfies that goal. Helicopter autopilot simulations use this to find the set of flight control inputs to determine which sets of inputs leads to stability (Murray-Smith 2000).

There are two methods for setting up an inverse simulation (Murray-Smith 2000). The first approach involves numerical differential techniques that would not apply to agent based methodology. The second is effectively running the model with slightly different parameters each time in an attempt to find parameters that match the desired outcome. This can take a long time computationally, as it is a search of all the possible parameters. If a simulation had 500 agents, each with 4 decisions to make each cycle, each possible combination of inputs would be tried. The major barriers to inverse simulation is significant run times, and that each inverse simulation has to be specifically tailored.

This could be paired very well with empirical modeling. A big problem in economics is the "correlation does not mean causation" problem. Another is the unobserved factors. In engineering they are often used to explore the possible causes of a single observed time series (Murray-Smith 2000). An empirical modeler could team up with a computational economist to build an inverse model that asks "what possible parameters of theory lead to these observed empirical findings."

Another aspect of agent based modeling is that it is the only area of economics whereby an economist can create an artificial agent that can make real world decisions that affect actual profit and loss. The explosion of the internet has large potential for a new class of economist, who can put their game theory and econometrics in automated agent form. Kephart et al (2000) envisioned that by 2010 much of the worlds economic decisions will be made by such automated agents.

Already, Kephart, Hanson, and Greenwald (2000) noted, there are automated economic agents in operation. "Shopbots" compare prices, and things such as eSnipe make use of the optimal bidding behavior, placing a bid at the last moment. They also experimented with how such things as an automated stock market would affect volatility. In a pure automated stock market with automated agents using optimal strategies that adapt, much of the volatility disappears and markets return to pure random walk (Kephart, Hanson, and Greenwald (2000).

2.8 Conclusion

With increasing computational power and the widespread growth of the internet, there are growth opportunities in the nonacademic world. These new techniques could have academic merit as well.

Given the nature of social systems, however, agent based approaches are one way to model economic theory from the bottom up. The multi-dimensionality and complexity of social systems does not lend it self to closed algebraic solutions. Computational methods are suited for these types of problems. The two essays that follow are implementations of a framework for agent based modeling, and explore how agent based modeling can model dynamics in novel ways.

Chapter 3

An Evolutionary Investigation into Dishonesty

The self interested nature of human beings often leads to "inflated communication" or dishonesty. In some cases, dishonesty can be a crime. In others, there may only be a reputation penalty. Lying is distasteful but most of us at some point will make use of the tactic. No one wants to live in a society where trust is rare and dishonesty is expected.

There are two ways to enforce honesty that are vastly different. One is to have external enforcement, such as a court system that can punish offenders. This external punishment could also come from society in the form of social norms. The other is moral enforcement. Some people prefer to be honest even when it is anonymous and devoid of any external punishment.

One way of studying dishonesty is games with volunteers. However, it can be difficult to know for certain the motivations of the volunteers. The present experiment involves computer agents replacing humans playing "Rock, Paper, Scissors" (RPS). Strategies are the chance of choosing each. However, there is a modification. In each game, one player is required to tell the other their strategy, that will be put in terms of R/P/S[1]. The other has a choice whether to believe it, and to make a pick that best matches the stated strategy of the opponent.

The underlying RPS strategies for each player are fixed but a genetic algorithm is used to evolve the chance that a player lies or believes. In comparison with experimental games involving human players, evolutionary games with a genetic algorithm mimic human behavior often better than any other methods. An added bonus is there is no mystery as to the chance a player will lie and every detail of the experiment can be examined.

[1] A 50% chance of choosing rock, 30% chance of choosing paper, and a 20% chance of choosing scissors is 50/30/20.

This game is modified further by then allowing players to "litigate" or reveal if the opponent is lying. If the opponent is lying, the litigator receives a payoff and the liar gets a penalty. However, if litigation reveals the opponent to be honest, the litigator is punished and a payoff goes to the falsely accused.

The importance of understanding the dynamic patterns of dishonesty, belief, and litigation is the perceived patterns across many social and market systems. In some countries and markets, no one is trusted. In others trust is persistent and dishonesty rare. Most may be somewhere in between the extremes.

A third possibility is a cyclical pattern that would be difficult to observe and measure. An example could be financial markets where judicial investigations appear to be clustered. It is not known if this is the result of more public scrutiny, political cycles, or temporary cycles of dishonesty.

3.1 Literature Review

Lying in the economic literature is called "strategic information transmission" (Crawford and Sobel 1982) and is an example of asymmetric information (Akerlof 1970). The consequences are highly sensitive to the situation and agent, however.

If lying is a crime, then the decision to lie also falls under the umbrella of criminal cost benefit analysis (Becker 1968). The more crime that is committed, the more presumed social harm and interest in preventing and punishing those crimes. Enforcement depends on the harm of the behavior. Given costs of enforcement, it is inefficient to set the goal of zero criminal activity.

The theoretical framework for lying and believing is set forth in Crawford and Sobel (1982). The side with more information than the other is the "sender" while the other side that

can choose to believe or not is the "receiver." Honesty and belief become prevalent only when they both have the same goals. Deviations far enough from a common goal result in a breakdown of communication and the optimal result is no messaging.

Not all lies are equal. Gneezy (2005) sets out four categories of lies. There are lies that may benefit both sender and receiver, such as complimenting appearance. Another type is the "altruistic lie," if people want higher total surplus instead of higher utility for themselves. Third is the spiteful lie that harms sender and receiver. The fourth type of lie benefits the sender at the receiver's expense. This fourth type is the assumption of the present simulation.

Lying is difficult to measure for many reasons. Lies have different payoffs and costs. Gneezy (2005) undertook an experimental game approach with human participants in a non zero sum game in which one player could choose to tell the other whether to pick choice A or B. The assumption is that the sender was telling the receiver what was in their best interests. The fraction that lies is dependent on the payoff matrix. Participants are less likely to lie when the loss to the other player is increased, and more likely when their own benefit is increased. The result shows individuals are a mixture of altruistic and self interested.

Another internal limit on lying comes from Mazar, On, and Ariely (2008). In addition to altruistic an moral code standards, people may practice "self concept maintenance." Actors wish to maintain their belief of their own general "goodness" but still lie to some degree. To avoid these lies shattering their false perception, they rationalize their lies.

Such rationalization may explain why insurance and tax fraud are judged differently from other lies (Tennyson 1997). The penalty of committing fraud is especially low because prosecution is so rare. Consumer attitudes towards the acceptability of fraud depend on the

circumstance. A major determinant is whether the person is in a state where more people find it justifiable, along with acceptability of misreporting to the tax authority.

A genetic algorithm (GA) is a method of learning by automated agents developed by Holland (1975). An advantage of GA's are that the agents' behaviors and motivations are not in doubt. Also there is no need to compensate the players. In addition, experimental games with human participants to avoid negative payments in order to help attract volunteers.

3.2 Methodology

Rock, Paper, Scissors was chosen as the underlying game for two reasons. First, there is no dominant strategy in RPS. In fact, if the strategies are evenly distributed randomly, as they are in this experiment, each strategy is identical in terms of expected wins. Second, it is easy to group strategies in respect to distance from 33/33/33. In these terms, 100/0/0 and 0/100/0 are the same distance.

Distance matters because once players have to tell the other their strategy, a 100/0/0 player will lose every time. This is because an extreme strategy that is broadcast to an opponent will result in that opponent picking the perfect counter. However, even a 33/33/33 player wins more if they lie and get away with it as they can tell their opponent they will always choose rock while their true strategy is something else.

Another benefit of RPS is that the payoff matrix can be changed to approximate the situations put forth in Crawford and Sobel (1982). By rewarding ties, and giving agents the ability to go for ties, the effects on lying and belief can be determined.

The game is composed of 500 players and 2,000 rounds. In each round, players are randomly paired. Player 2 makes a choice to lie or be honest about their strategy. If player 2 lies, it broadcasts in 100/0/0 form, telling the other they have a 100% chance of picking one

move. If player 2 lies, they will pick the lie that makes their underlying strategy most likely to succeed. For instance if they have a 75% chance of picking paper, they will tell a lie in order to trick the opponent into picking rock. If the player decides to be honest, then the underlying strategy is sent.

If the other player (*player1*) believes what the opponent says, then they make a choice that counters. Disbelief results in falling back on their underlying strategy. After the picks are made, the match is evaluated and the winner gets +1, ties are 0 for both, and losing results in -1. Each of these is added to their total winnings. The winnings are how fitness of a strategy is evaluated.

Each player is assumed to have zero internal cost of lying, strictly following Becker's (1968) cost/benefit assumption. The decision to litigate follows this assumption also. The only outcome in which one player does not lose points is in a tie. One modification allows for players to go for a tie that are rewarded.

Every 20 rounds, the winnings are reset. Learning is done based on the most fit strategies, and only strategies that work out best over many rounds reproduce. Before the winnings are reset, a genetic algorithm evolves the strategies. Then the next round begins.

Lying, belief chance, and the chance of litigation are evolved, while RPS strategies are fixed. To make them evolve with a genetic algorithm, they must be in a binary string.

A binary string is a set of 0's and 1's of length *l* such that the decimal equivalent is found as $\sum_{k=1}^{l} \alpha_{i,t}^{k}$ with α being the value at the position k[2]. If it is a percentage that is represented, this

[2] The string 1011, reading from the rightmost value is 1, so this is the 2^0, position, like how in decimal this is the 10^0 position. So, there is 1 2^0, 1 2^1, 0 2^2, and 1 2^3. Adding these up gets the decimal value of 11.

value must be scaled so that it falls in between 0 and 1. A way to do this is to divide by $2^l - 1$. This means dividing by the maximum value of any string of length l.

A length of 7 was chosen because the decimal equivalent maximum value of this is closest to 100. This is to minimize the equivalent strings because percentages are rounded. Large strings would result in some binary strings being effectively equivalent. In simulations with more than one evolved trait, the strings are lumped. When this is done the simulation makes use of a string of size $l = 21$, with each segment of 7 encoding a different part of the strategy read and scaled separately.

Once the strings are encoded, the standard practice is replication, crossover, and mutation. Replication is intended to mimic "survival of the fittest." Crossover is a way of generating new strategies. Mutation is to model random innovation. Fitness is determined by number of wins in 20 games.

Every 20 games, strings are ranked and replication is accomplished by creating a pool of strategies. Strings that come from players with more wins are more likely to be picked. The bottom 30% of the players' strategies are not included in the pool. Each player then gets a strategy from the pool for the next 20 games.

Crossover involves pairing players up randomly. A spot in the string is picked, and the portion of each string at that point is exchanged. As the strategies converge crossover has a diminishing effect.

Mutation is vital in the process. If the players converge on an unstable equilibrium, crossover will not yield a challenging strategy. However, with mutation even a temporary settling will be challenged by random strategies that, if win out, will eventually be copied.

Practically it involves a chance of a player's strategy being mutated slightly by switching a part of a string from 0 to 1 or vice versa. For this simulation, the mutation rate is 5%.

Each player will be in the *player1* (the "receiver") and *player2* (the "sender") position half the time on average. In the *player1* position, they have a chance η_i of believing what *player2* broadcasts. If the player does not believe, they have chance δ_i of litigating. Success depends on the average lie chance of the population, κ. If they do not believe, the outcome depends on their own RPS strategy, θ_i, and the population average strategy, θ.

$$E(player1) = \eta_i * ((1-\kappa) * RPS(\theta_i, \theta) + \kappa*(-1)) + (1-\eta_i) * (\delta_i * ((\kappa*(2)) + (1-\kappa) * (-2)) + (1- \delta_i) * RPS(\theta_i, \theta))$$

For *player2*, the player in the position to lie or be honest, the outcome depends on their chance of lying, κ_i, the population average chance of believing, η, and the population average chance of litigation, δ.

$$E(player2) = \kappa_i * ((\eta* (1)) + ((1- \eta) * (\delta* (-2)) + ((1-\delta) *RPS(\theta))) + ((1-\kappa_i) * (\eta_i * RPS(\theta_i, \theta)) + (\delta* (2)))$$

Players have the expected unconditional outcome in each round of:

$$E(player) = .5*E(player1) + .5*E(player2)$$

At the mean, the strategies of the population are roughly 33/33/33[3]. Strategies are evenly distributed so a 100/0/0 is as likely to occur as 34/33/33. The expected outcome of any strategy paired with an opponent is 0. In the initial turns, the probability of lying, litigating, and believing is also evenly distributed, with mean 0.5.

[3] Decimals are avoided so there is no 33/33/33, the closest possible is 34/33/33, 33/34/33, or 33/33/34. Nevertheless, the average of all players is roughly $33^{1/3}/33^{1/3}/33^{1/3}$.

The expected outcome of telling the truth is 0 at the mean, as broadcasting a 33/33/33 is not detrimental to the outcome. However, the benefits of lying increase as the distance away from 33/33/33 increases. A 100/0/0 player always loses if they tell the truth.

For any given set of average strategies κ, δ, and η, each player has a linear set of strategies that have the same expected zero value as the standard RPS game. However, the nature of the genetic algorithm will tend to eliminate strategies that have higher variance. This property should result in strategies that are grouped together, but may deviate on a linear path.

This model is programmed using the statistical language R, which has never been used in publication for an agent based model. The advantage of using R is more transparency, ease of debugging, and quicker implementation of changes. In addition the built in graphical functions of R make output simpler to do and more informative.

Using R's capabilities for graphing, the program will output each frame of some key elements of the model in each time frame into a plot. Each time frame's plot is saved into a file such that an external program can read them in and make a movie out of the output. Some stills from this movie appear as figures below.

In addition to the graphical capabilities, R comes with the standard statistical and mathematical analysis abilities that are in programs such as MATLAB. Because it is free and open source, researchers can make their functionality available in publically available libraries. These features combine to make R a great academic tool for agent based modeling.

A drawback is the runtime of each of the model presented here. With relatively small strings and population size compared to some in the literature, the program takes about 15 minutes to run. With the requirement that agent based models have to be run hundreds if not

thousands of times, much of the development time is spent waiting. However this can be alleviated by making use of more than one computer.

3.3 Results

Results are presented with a "Lying vs Belief" scatter plot with each quadrant relating to the prevalence of a set of strategies. Figure 1 shows an empty graph with no data points on it.

Figure 1

Anything with low belief is regarded as communication breakdown. At the beginning, players are randomly and uniformly distributed over this space.

Figure 2 shows where the game ends up when there is no punishment for lying. The result is an equilibrium where lying is prevalent and belief is low.

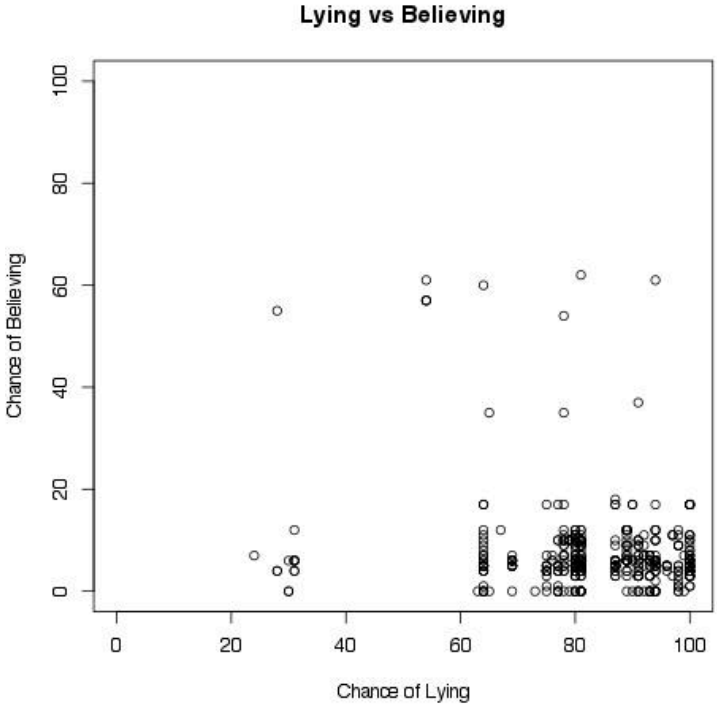

Figure 2

Adding the ability to litigate and to punish liars changes the outcome but does not increase belief. Instead, lying all but disappears. Another graph shows the three traits – lying, belief, and litigation chance – on the same graph. Receivers only sue enough to keep dishonesty in check, but a small number continue to lie. However, belief is low. Figure 3 shows the end points for these parameters. Figure 4 shows the dynamic time paths of the average strategies.

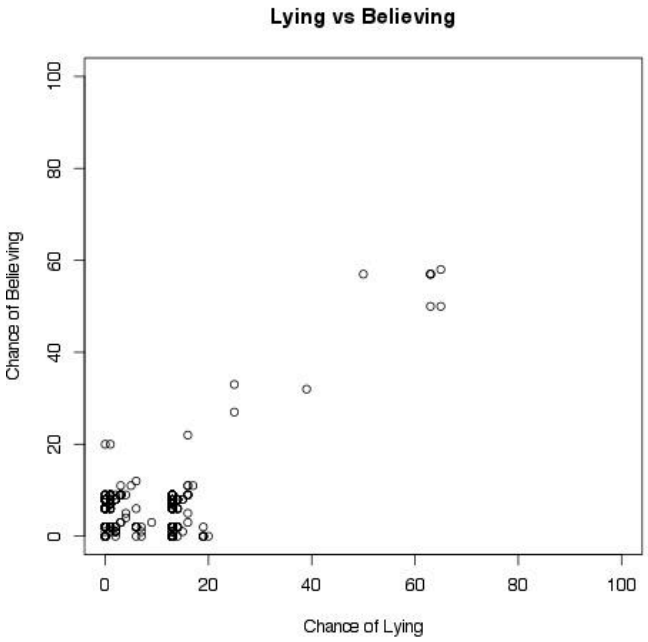

Figure 3

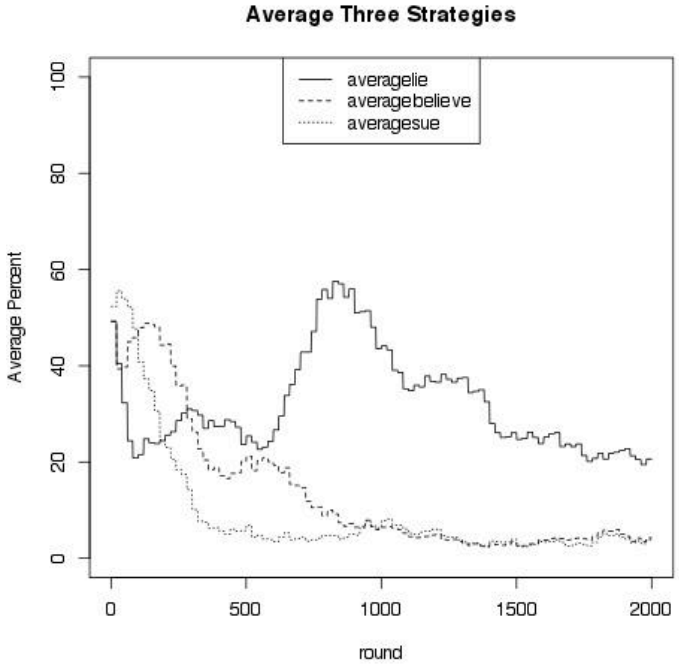

Figure 4

With divergent goals, there is no incentive for the players to be honest. By changing the payoffs to reward ties with 2 points and allowing both to go for ties, the players have an interest

in being honest. The decision to go for a tie is contingent on the sender not lying, the receiver believing, and both deciding to go for a tie.

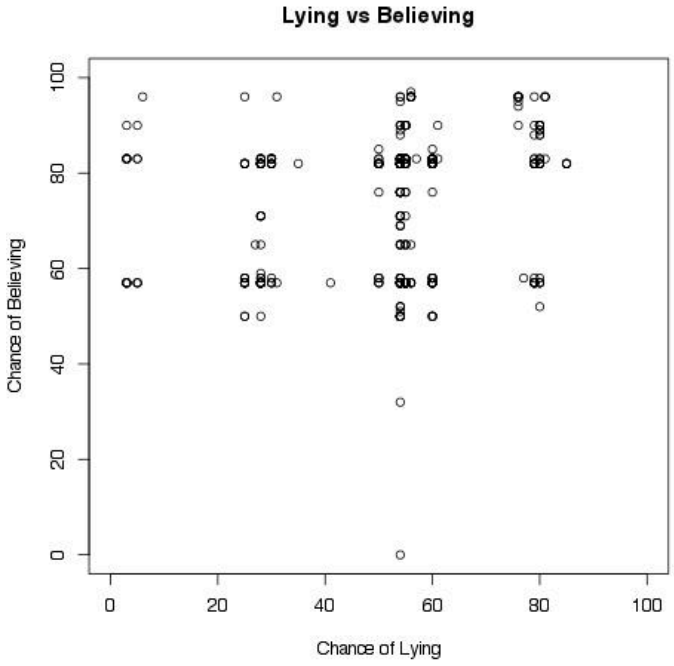

Figure 5

This outcome is not clear cut. A momentary snapshot in Figure 5 of the end does not capture the dynamics of this game. A look at the time paths of all the strategies is telling. Players are cycling through the quadrants, and stability is not attained. A full video of this run is available on the internet[4].

As soon as high levels of belief are reached, the incentive for being dishonest increases. This in turn causes the belief levels to fall and the levels of litigations to rise. Once the litigations rises enough, lying decreases and honesty resumes. The result is a cycle of honesty/lying/litigation/honesty that persists throughout the 2000 rounds.

[4] www.youtube.com/watch?v=D3IF8JlNAQM

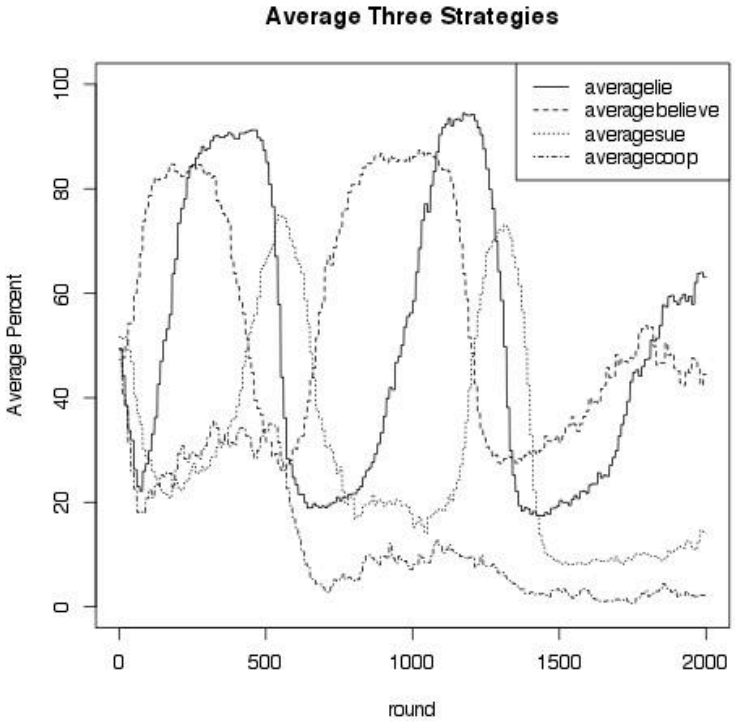

Figure 6

To get the players to end up in a honest/belief stable equilibrium, ties must be rewarded even more. Rewarding ties at 5 points ensures the honest/belief outcome.

3.4 Conclusion

The dynamic patterns of dishonesty, belief, and punishment are difficult to observe by data gathering. For example the 2007 financial crisis brought to the surface many dishonest practices in the financial sector. This in turn caused a litigation backlash by the authorities. It is unknown if the level of dishonesty in this sector is constant and a crisis merely causes the authorities to uncover it, or if levels of unethical behavior rise leading to a crisis.

Patterns in the evolutionary games presented here suggest that a potential pattern is low levels of dishonesty followed by spikes in dishonest behavior if agents are allowed to play a non zero sum game if cooperation is not rewarded enough. Rising litigation follows. We can

observe litigation in court systems, but other patterns of honesty and dishonesty at the individual level do not register in a data set.

A genetic algorithm in a game where agents could lie and punish liars results in a fairly honest outcome. However, belief of the signal dropped to near zero. Effectively, communication ceased. This conforms to the possible outcome of strategic communication in Crawford and Sobel (1982). In a world where trust and cooperation can benefit, this is not a desirable outcome.

Results here suggest that it is not enough for cooperation to be rewarded more than going alone. Below a critical point of reward, the system becomes highly variable. In markets and economic environments where cooperation is not so beneficial, this model could predict the possible patterns. The expected value of cooperation depends on the chance of others cooperating. Any drop in this chance causes a downward race to the bottom that is only reversible by a punishment mechanism.

In the zero sum game with the ability to punish liars, lying still occurs at a stable rate. Litigation settles at a low level, similar to real world insurance fraud and prosecution rates. With no reward for cooperation this situation proves stable.

It would be difficult to empirically confirm these results with empirical data. The true rate of dishonesty in a stable environment would be a fraction of what is observed. In a dynamic environment, the true rate of dishonesty and the observed rate would depend on enough factors that determination with certainty would be impossible.

Experimental games with human subjects would have the best chance of determining if these patterns exist. However, the number of rounds in the present is in the thousands. Human

experiments are expensive and would require many rounds for the subjects to react to changing environments. Also, each set of human subjects would be unique.

The present agents represent zero information participants. Ethics formation and value transmission across generations are not modeled. This approach models an environment in which new agents take the place of old, and merely find the optimal strategy to the new situation over the course of a the game. An ethical agent would be ousted in an environment in which ethics did not pay off, and a lying agent would be ousted in an environment in which lying is punished. In this sense the present approach does not model changing individual behavior in a market but models which behaviors are likely to be rewarded and become prevalent.

Future research may focus more on social pressures of agents within the model. Humans tend to look at those around them to help determine their acceptable range of behavior. This coud be incorporated by having agents examine societal "norms" which would come through as average values of past rounds. This would introduce a type of inertia in the system in which things would be hesistant to change. However, if that was included it may not stop the dynamics but may only make the peaks more persistent. Upon reaching a peak the societal norms would plateua but the nature of genetic algorithms would produce some agents that would challenge these norms and still cause dynamics. Different core games other than RPS could show different patterns as well. Non zero sum games or games such could change the strategies for cooperation and litigation.

Chapter 4

Trade Network Formation with a Genetic Algorithm

4.1 Introduction

Trade networks conceptually encompass both international and intranational trade. Any exchange of goods or services that results in Pareto improvement results in a trade network. The present chapter attempts to improve on prior work that describes types of networks by adding the ability of automated trade agents to decide how many networks they would like to form. The trade agents also have the ability to sever networks.

An agent based model with genetic algorithm components forms the core of the present analytical method. Objectively the model will attempt to mirror "stylized" facts about the real world. Namely, trade is a constant part of human society. Despite the volume of Pareto improvements, Pareto "finished" with no incentive to trade has yet to be reached.

Trading has been a major avenue of agent based modeling, from workers and employers trading time and money (Tesfatsion 1998), stock market trading (Lebaron 2002, Ehrentreich 2006), trading partner loyalty (Kirman and Vriend, 2000), and formation of exchange rates (Marimon, McGrattan, Sargent, 1990). Resulting inequality differences amongst a population of artificial agents is explored in Epstein and Axtell (1996) in their "sugarscape" models. The present focus is allowing agents to determine how many trading partners to have via a genetic algorithm.

Jackson and Wolinsky (1996) forms the theoretical basis for the strategy of when to form a new connection with another "node" or trading agent. A connection is only formed if both agents see trade as advantageous. In addition, the decision to sever the link can come from either

of the nodes in the link. This ability to sever or make new connections on the fly is the main innovation of the present chapter.

Others have studied the characteristics of different types of trade networks, falling under the umbrella of "trade topography." The significance of trade topography is easy to overlook in trade theory, but its importance was summed up by Schelling (1978). He noted that if everyone needs a 100 watts of light to read while we all only have a 60 watt light bulb, reading will cease if we are arranged in a line but continue if we are in a circle.

Allowing agents to decide the number of trading partners was first suggested in Ioannides (1997) who details different trade topologies. Prior work in topologies assumes a given set of connections. Examples include the "star" pattern (Bell 1998) and variations of the "circle" pattern (Bell 1998, Wilhite 2001). The circle pattern is one in which agents arranged in a circle can trade with their neighbors, while the star pattern connects all agents through a centralized hub. Allowing topologies to change based upon agent decisions is a "random graph" model first articulated by Kirman (1983). Prior work in the random graph approach uses fixed probabilities of interaction between agents.

The agent based model in the present chapter closely follows Wilhite (2001) except in a few key areas. In that study, the focal point is the number of trades and the standard deviation of prices among different types of fixed trade networks. Agents are given a random allotment of two goods and all have the same utility function. Several types of fixed trade networks are tested to see how each topography changes the standard deviation of prices among trade groups, the number of trades, and the number of searches.

For instance one type of network is the "local network." The local network has subsets of agents which only trade with each other. This network has the highest standard deviation of

prices. Comparatively, the "global network" in which every agent was connected to each other has the lowest standard deviation but the highest search costs. In the middle are the two intermediate types in which a variation of the circle network where two agents in each subgroup are connected to the next group, and another where a cross over connection between two distant groups is allowed. The local connected network results in benefits of both extremes with relatively fewer trades and lower standard deviation of prices.

Agent based models (ABM) is a subset of computational economics. Fundamentally ABM excels at non homogeneity and situations where dynamic properties are important (Tesfatsion 2002). In addition, agent based models with probabilistic elements are Markov processes (Gintis 2012).

The addition of genetic algorithms (GA) to the trade topography literature is unique. Genetic algorithms are a method of searching through possible optimal solutions to complex problems. They involve representing a strategy with a "string" of 0's and 1's, and associating each string with a "fitness" level that determines its chance of continuing. The process is intended to pattern biological evolution and was first proposed by Holland (1975).

In each "tick" or measure of time, agents trade based upon rules. After a specified number of ticks (allowing more ticks prevents random elements from causing wide fluctuations) the agent genetic information is sorted by fitness, the least fit strategies are dropped, while the most fit engage in "cross over" mimicking reproduction. The top 5% of the strategies are allowed to continue on unscathed in the present simulation. Random mutations also occur, and are combined with crossovers as a way of introducing new strategies.

Genetic algorithms are also described by the number of populations. Each population learns from only its members. The model presented here is a single population GA. A

population will tend to converge to a Nash equilibrium and is also an evolutionary game (Riechmann 2001). "Winning" is determined by having the largest fitness. In this chapter fitness is determined by highest utility.

4.2 Methodology

As a basis for comparison and a convenient starting off point, Wilhite's (2001) trade network model is the foundation for much of this work. At the beginning of the model, each of the *N* agents are given a random allotment of two goods - g_1 and g_2. There is no production, or a price formation process. Trades are only undertaken if there is a Pareto improvement for both agents. Otherwise the negotiating stops.

A baseline utility that is the same as Wilhite (2001) is used to compare with a utility function that is more like Bell (1998). Both use a Cobb Douglas, but in the former the utility function is identical for all agents, while the latter has a penalty for the cost of connections. The utility hybrid utility function for these agents is $u_i = g_1^i g_2^i - \delta_i$, with δ signifying the number of searches. A high number of searches resulting in no trades will decrease utility.

The first term is the baseline utility, the next is the effect of trading, and the last is a search cost term. In this study the search cost is the square of the number of negotiating partners. The cost is present even if no accepted price is reach or trades performed.

Once the allocations are set, every agent is given a turn to conduct searches and trades. This is done without replacement in that if agent 1 trades with agent 2, when agent 2 gets a turn it can trade with agent 1. Agents rank trading partners on potential prices, which are set at the midpoint of each agent's MRS, which depends on its randomized utility function. Once a partner is found, trading continues until one of the partners determines an additional trade at that

price will not increase its utility. The selected agent then continues on to the next potential trading partner.

For comparison, Wilhite's (2001) model is not only a comparison point but also serves as a subset of the computer program used to make this one. All of the outcomes in that paper are a subset of the parameters asked upon opening the program - in this case written in Objective - C (instead of C++). At the beginning, the program asks how many groups there are, and how many agents in each group. It also asks some details about the connectedness of agents, and whether utility is random.

The "small world network" from Wilhite (2001) is achieved by inputting a number of groups, telling it how many in each group, and specifying that they are not to interact with other groups. Global network is achieved by specifying one group with all the agents, and the intermediaries are achieved by another detail.

The innovation here comes from an additional possible characteristic - allowing agents to determine their own trade networks. Also, they use a genetic algorithm to determine the number of partners. This parameter ranges from 0-100, with 100% meaning a global network in which each agent seeks out possible trades with the entire population of agents, and 0% meaning complete autarky. Over the course of the simulation the dynamics change.

Finally, an overlapping generations element is added by randomly "killing off" a percentage of the agents each cycle. Each death results in a new agent that inherits the previous agent's stock of goods, but has a random utility.

4.3 Results

Below is a graph showing the results of a 1000 runs, with the utility functions of the agents fixed and no penalty for the number of trading partners. This is the baseline scenario

most similar to Wilhite's (2001) trade networks and the results from this are in Figure 1. The purpose of this graph is to demonstrate that the model, while different in implementation, starts with similar results.

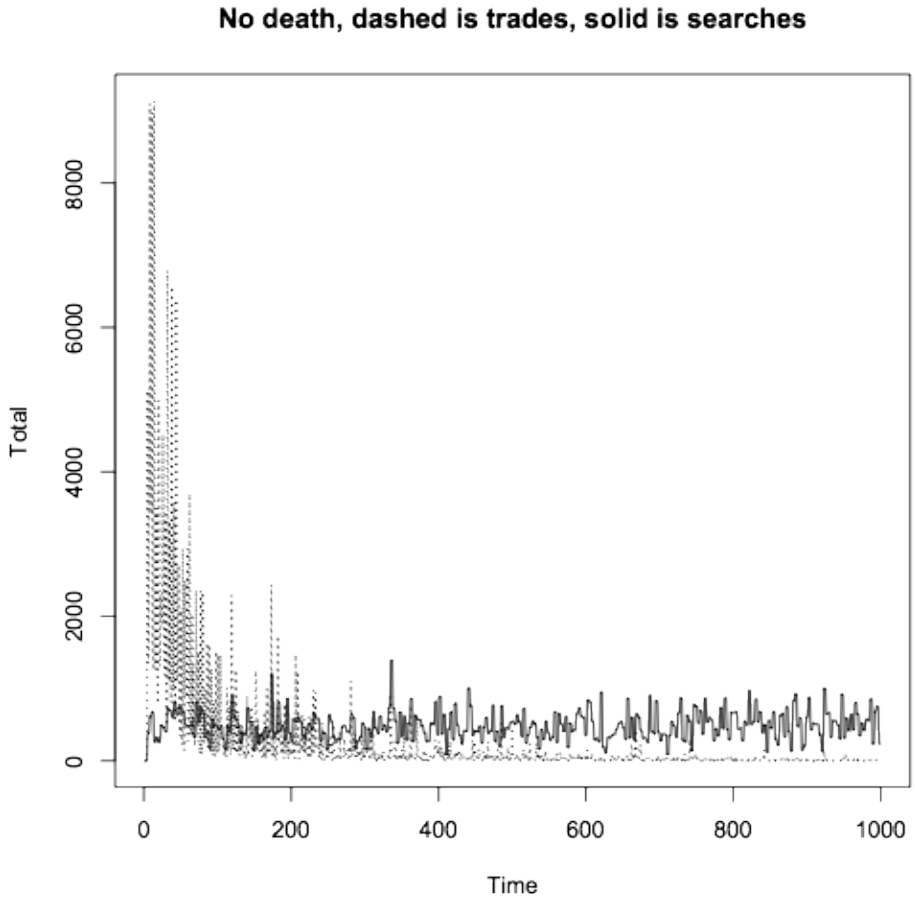

Figure 7

This outcome does not make sense given costly search. If there are no Pareto improvements, it does not make sense to continue searching. In order to better mirror reality, search costs must be added.

The addition of search costs and allowing agents to determine how many trading partners to have causes a different pattern of trade than in Wilhite (2001). At the beginning, all agents start out with zero trading partners. However, mutation causes some agents to figure out that

trading increases their utility. Trading increases initially, and then decreases to near zero in the model with no death. These results are shown in Figure 2.

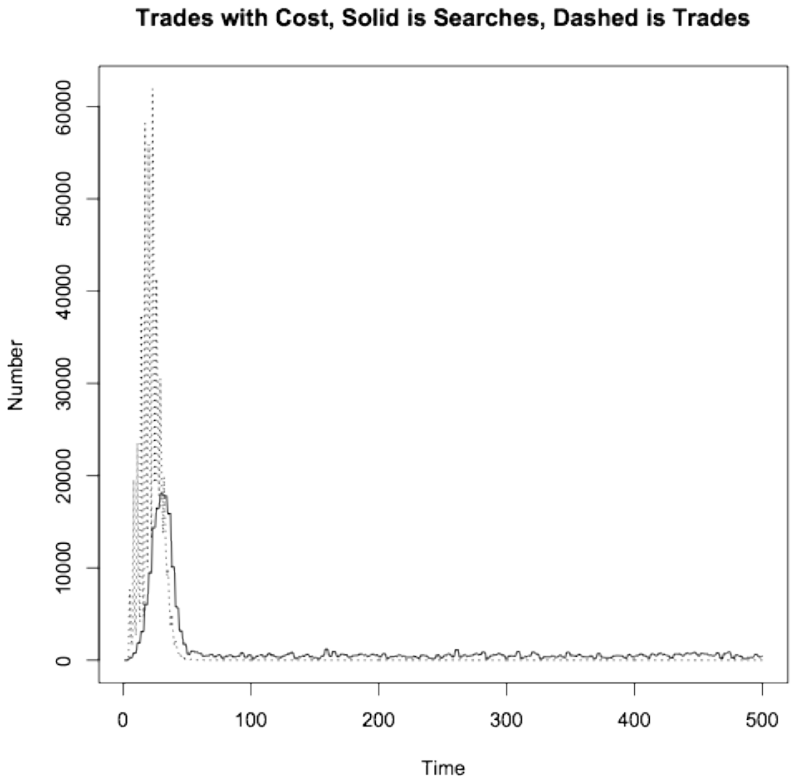

Figure 8

While this may explain a static situation in which once agents become Pareto optimized, clearly in real life trade has yet to cease on its own. A more dynamic approach must be adopted to model the constant flow of goods.

With OLG type attributes, and replacing of some established utility functions with random ones, trade continues ("death"). Although it does decrease after the initial burst of trading, it exhibits a persistent quality that mirrors real world trading patterns. What level of trading is determined by the "death" rate and the random mutation rate.

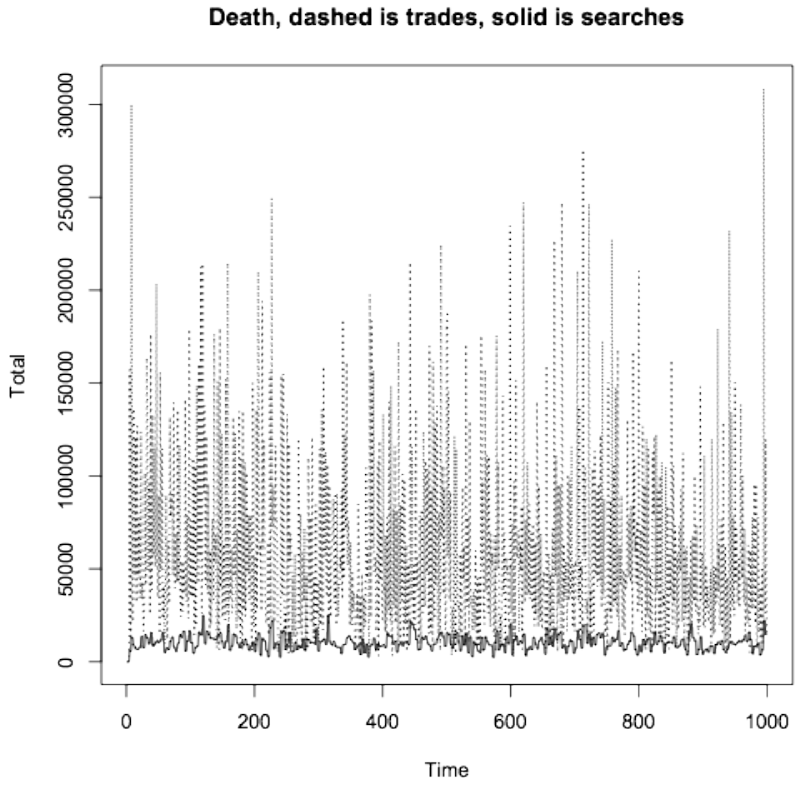

Figure 9

Figure 9 shows an average over 20 runs of the model. Every time period 10% of the agents die and are randomly replaced. This percentage is enough to keep the number of searches and trades continuing on. The number of goods at the start is the same as in the end, so these agents are constantly sending the same goods back and forth.

4.4 Conclusion

Trade networks are dynamic. The present model explains some of the dynamic properties of trade networks. Even without production, birth and death are enough to keep trade networks shifting and continuous even in the presence of search costs.

Search costs in some arenas may in fact be near zero. On the Internet, the marginal cost of an automated trader contacting another automated trader is next to zero in most contexts. Other trade networks such as grocery shopping have search costs.

In order to better model trade network formation, production can be included. At this time there is no approach to incorporating trade network formation and production. Trade networks may also involve borders. There are also no social effects such as preferring a previous trading partner over another.

The addition of genetic learning and dynamic trade network formation with search costs has illustrated dynamics of trade networks that may explain real world patterns. Even without production of new goods, the process of replacing old agents with new ones with different utility functions is enough to keep trade going.

Chapter 5

Conclusion

After the 2007 recession a hearing in congress investigated existing models in macroeconomics and their failures to predict the crisis. An article in *The Economist* (June 22nd 2010) asked if agent based modeling could succede where others failed. Agent based modeling explained the volatility of stock markets and the tendency for crashes. The same article wrote of a brief flury of workshops in which the top experts presented agent based models that explained the crash. However, explaining something that already happened is not only easy, it is what agent based modeling is designed to do.

Furthermore, agent based modeling like other models can be adjusted to include and prove a wide range of possible outcomes, so can agent based modeling (ABM). While the internals of an ABM appear to be ad hoc, there is an empirical qualitative or quantitatve aspect. The larger picture it paints must conform to an observation about a market or society. As the models get more complex the importantce of mirroring an empirical or stylized fact increases because otherwise the entire model has little purpose.

The usage of a genetic algorithms allows for more possibilities – a genetic algorithm with a string size of eight can search over 65000 possible solutions. Adding more individuals that can arrive at different solutions multiplies the number of possibilities. A genetic algorithm model of even a small population with each citizen treated as an agent would be both creatively and computationally difficult but not out of the realm of the possible.

For smaller research questions on microeconomics practicioners face the uphill climb of getting their work accepted. Many economists will assume modeling of this sort is ad hoc. But

the successes of prior work in explaining phenomena that other tools cannot should give the area some traction.

The first limitation is ability to program, debug, and manage code. Because of the prevalence and ease of making mistakes, clarity, transparency, and reproducibility are paramount. As econometricians use statistics packages in basics such as matrix inversion, agent based modeling practitioners should rely on a minimum of common code that has been thoroughly debugged.

The existing tools give non programmers an ease of entry but give up some power. Customization is difficult. If a modeler wishes to create some populations that learn from other populations for instance, a custom framework is necessary. That custom framework should contain the minimum code for gentic algorithm routines but not much else to both limit learning time and errors in the code. For instance, the Santa Fe Artificial Stock Market was found to have a minor error years after many of the papers it produced were published (Ehrentreich 2004). Any error found after publication spreads distrust of the models.

The present Chapter 2 shows how a programming language many economists know can be adapted for agent based modeling. A language designed for statistics allows adaptive agents to play a game and learn genetically. The dynamic patterns of lying and trust are displayed with existing graphing capabilities.

Chapter 3 is an agent based model that used a genetic algorithm to learn. However, it is different in that its goal of examining trade partners and agents did not explicitly play a game against themselves. However, the collective attributes affected individual payoffs. Its lower level language is faster, but the cost is developing and learning the code.

Ultimately, anyone using agent based modeling and genetic algorithms will have to choose an approach that maximizes effectiveness at a reasonable cost of effort. For instance, the "sugarscape" model that was the centerpiece of Epstein and Axtell (1996) was 20,000 lines of code. Much of the code had to deal with graphical capabilities. If every ABM had a customized graphics functionality they would be cost prohibitive. Languages such as R can make use of already existing graphical capabilities. For other applications where execution speed is a concern, a lower level language such as C will be chosen.

References

Akerlof, G.A., 1970. The market for“ lemons: Quality uncertainty and the market mechanism. The quarterly journal of economics 488–500.

Albin, P., Foley, D.K., 1992. Decentralized, dispersed exchange without an auctioneer: A simulation study. Journal of Economic Behavior & Organization 18, 27–51.

Andreoni, J., Miller, J.H., 1995. Auctions with Artificial Adaptive Agents. Games and Economic Behavior 10, 39–64.

Arifovic, J., 1995. Genetic algorithms and inflationary economies. Journal of Monetary Economics 36, 219–243.

Arifovic, J., 1996. The Behavior of the Exchange Rate in the Genetic Algorithm and Experimental Economies. Journal of Political Economy 104, 510–41.

Arifovic, J., Bullard, J., Duffy, J., 1997. The Transition from Stagnation to Growth: An Adaptive Learning Approach. Journal of Economic Growth 2, 185–209.

Arthur, W.B., Holland, J.H., LeBaron, B., Palmer, R., Tayler, P., 1996. Asset pricing under endogenous expectations in an artificial stock market.

Arthur, W.B., 1991. Designing Economic Agents that Act Like Human Agents : A Behavioral Approach to Bounded Rationality. The American Economic Review 81, 353–359.

Axelrod, R., Hamilton, W.D., 1981. The evolution of cooperation. Science 211, 1390–1396.

Başçı, E., 1999. Learning by imitation. Journal of Economic Dynamics and Control 23, 1569–1585.

Beaufils, B., Delahaye, J.-P., Mathieu, P., 1998. Complete classes of strategies for the Classical Iterated Prisoner’s Dilemma, in: Porto, V.W., Saravanan, N., Waagen, D., Eiben, A.E. (Eds.), Evolutionary Programming VII, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 33–41.

Becker, G.S., 1974. Crime and Punishment: An Economic Approach (NBER Chapters). National Bureau of Economic Research, Inc.

Bell, A. M. (1998). Bilateral trading on a network: A simulation study. Working Notes: Artificial Societies and Computational Markets.

Birchenhall, C., 1995. Modular technical change and genetic algorithms. Comput Econ 8, 233–253.

Booker, L.B., Goldberg, D.E., Holland, J.H., 1989. Classifier systems and genetic algorithms. Artificial Intelligence 40, 235–282.

Borrill, P.L., Tesfatsion, L.S., 2010. Agent-Based Modeling: The Right Mathematics for the Social Sciences? (Staff General Research Paper). Iowa State University, Department of Economics.

Crawford, V., Sobel, J., 1982. Strategic Information Transmission. Econometrica 50, 1431–1451.

Deissenberg, C., van der Hoog, S., Dawid, H., 2008. EURACE: A massively parallel agent-based model of the European economy. Applied Mathematics and Computation 204, 541–552.

Ehrentreich, N., 2006. Technical trading in the Santa Fe Institute Artificial Stock Market revisited. Journal of Economic Behavior & Organization 61, 599–616.

Epstein, J.M., Axtell, R., 1996. Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institution Press.

Friedman, D., 1991. Evolutionary Games in Economics. Econometrica 59, 637–66.

Gale, D., Shapley, L.S., 1962. College Admissions and the Stability of Marriage. The American Mathematical Monthly 69, 9.

Gintis, H., 2007. The Dynamics of General Equilibrium. The Economic Journal 117, 1280–1309.

Gintis, H., 2013. Markov Models of Social Dynamics: Theory and Applications. ACM Trans. Intell. Syst. Technol. 4, 53:1–53:19.

Gneezy, U., 2005. Deception: The Role of Consequences. The American Economic Review 95, 384–394.

Gode, D. (Dan) K., Sunder, S., 2004. Double auction dynamics: structural effects of non-binding price controls. Journal of Economic Dynamics and Control 28, 1707–1731.

Holland, J.H., 1975. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, Oxford, England.

Holland, J.H., 1992. Adaptation in natural and artificial systems. 1975. Ann Arbor, MI: University of Michigan Press and.

Holland, J.H., Miller, J.H., 1991. Artificial Adaptive Agents in Economic Theory. American Economic Review 81, 365–71.

Hommes, C.H., 2006. Chapter 23 Heterogeneous Agent Models in Economics and Finance, in: L. Tesfatsion and K.L. Judd (Ed.), Handbook of Computational Economics. Elsevier, pp. 1109–1186.

Ioannides, Y.M., 1996. Evolution of Trading Structures (Working Paper No. 96-04-020). Santa Fe Institute.

Jackson, M.O., Wolinsky, A., 1996. A Strategic Model of Social and Economic Networks. Journal of Economic Theory 71, 44–74.

John, H., 1992. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge, MA.

John, 2001. Learning to speculate: Experiments with artificial and real agents. Journal of Economic Dynamics and Control 25, 295–319.

Kartik, N., Ottaviani, M., Squintani, F., 2007. Credulity, lies, and costly talk. Journal of Economic Theory 134, 93–116.

Kephart, J.O., Hanson, J.E., Greenwald, A.R., 2000. Dynamic pricing by software agents. Computer Networks 32, 731–752.

Kirman, A.P., Vriend, N.J., 2000. Learning to Be Loyal. A Study of the Marseille Fish Market, in: Gatti, P.D.D., Gallegati, P.M., Kirman, P.A. (Eds.), Interaction and Market Structure, Lecture Notes in Economics and Mathematical Systems. Springer Berlin Heidelberg, pp. 33–56.

Kirman, A.P., Vriend, N.J., 2001. Evolving market structure: An ACE model of price dispersion and loyalty. Journal of Economic Dynamics and Control 25, 459–502.

Kiyotaki, N., Wright, R., 1989. On Money as a Medium of Exchange. Journal of Political Economy 97, 927–54.

Kurahashi, S., Minami, U., Terano, T., 1999. Why not multiple solutions: agent-based social interaction analysis via inverse simulation, in: 1999 IEEE International Conference on Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. Presented at the 1999 IEEE International Conference on Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings, pp. 522–527 vol.2.

Kutschinski, E., Uthmann, T., Polani, D., 2003. Learning competitive pricing strategies by multi-agent reinforcement learning. Journal of Economic Dynamics and Control 27, 2207–2218.

Lebaron, B., 2002. Building the Santa Fe Artificial Stock Market. Physica A.

LeBaron, B., Arthur, W.B., Palmer, R., 1999. Time series properties of an artificial stock market. Journal of Economic Dynamics and Control 23, 1487–1516.

Lettau, M., 1997. Explaining the facts with adaptive agents: The case of mutual fund flows. Journal of Economic Dynamics and Control 21, 1117–1147.

Lucas, R.E., 1986. Adaptive Behavior and Economic Theory. The Journal of Business 59, S401–26.

Lux, T., Marchesi, M., 2000. Volatility clustering in financial markets: a microsimulation of interacting agents. International Journal of Theoretical and Applied Finance 03, 675–702.

Lux, T., Schornstein, S., 2005. Genetic learning as an explanation of stylized facts of foreign exchange markets. Journal of Mathematical Economics 41, 169–196.

Malleson, N., Birkin, M., 2012. Analysis of crime patterns through the integration of an agent-based model and a population microsimulation. Computers, Environment and Urban Systems 36, 551–561.

Marcet, A., Sargent, T. J. (1989, July). Least-squares learning and the dynamics of hyperinflation. In International Symposia in Economic Theory and Econometrics, edited by William Barnett, John Geweke, and Karl Shell (pp. 119-137).

Marimon, R., McGrattan, E., Sargent, T.J., 1990. Money as a medium of exchange in an economy with artificially intelligent agents. Journal of Economic Dynamics and Control 14, 329–373.

Marimon, R., Spear, S.E., Sunder, S., 1993. Expectationally Driven Market Volatility: An Experimental Study. Journal of Economic Theory 61, 74–103.

Maynard Smith, J., 1976. Evolution and the Theory of Games. American Scientist 64, 41–45.

Mazar, N., Amir, O., Ariely, D., 2008. The dishonesty of honest people: A theory of self-concept maintenance. Journal of marketing research 45, 633–644.

McFadzean, D., Tesfatsion, L., 1999. A C++ Platform for the Evolution of Trade Networks. Computational Economics 14, 109–134.

Minar, N., Burkhart, R., Langton, C., Askenazi, M., 1996. The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations (Working Paper). Santa Fe Institute.

Murray-Smith, D.J., 2000. The inverse simulation approach: a focused review of methods and applications. Mathematics and Computers in Simulation 53, 239–247.

Nilsson, D.-E., Pelger, S., 1994. A Pessimistic Estimate of the Time Required for an Eye to Evolve. Proc. R. Soc. Lond. B 256, 53–58.

Pagan, A., 1996. The econometrics of financial markets. Journal of Empirical Finance 3, 15–102.

Pingle, M., Tesfatsion, L., 2001. Non-Employment Benefits and the Evolution of Worker-Employer Cooperation: Experiments with Real and Computational Agents.

Riechmann, T., 2001. Genetic algorithm learning and evolutionary games. Journal of Economic Dynamics and Control 25, 1019–1037.

Resnick, M. 1996. StarLogo: An environment for decentralized modeling and decentralized thinking. In Conference companion on Human factors in computing systems (pp. 11-12). ACM.

Rouchier, J., Bousquet, F., Requier-Desjardins, M., Antona, M., 2001. A multi-agent model for describing transhumance in North Cameroon: Comparison of different rationality to develop a routine. Journal of Economic Dynamics and Control 25, 527–559.

Sargent, T.J., Wallace, N., 1981. Some unpleasant monetarist arithmetic. Quarterly Review.

Scarf, H.E., 1959. Some Examples of Global Instability of the Competitive Equilibrium (Cowles Foundation Discussion Paper No. 79). Cowles Foundation for Research in Economics, Yale University.

Schelling, T., 1969. Models of Segregation. The American Economic Review 59, 488–493.

Schelling, T.C., 2006. Micromotives and Macrobehavior. W. W. Norton & Company.

Smith, Vernon L 1962. An experimental study of competitive market behavior. The Journal of Political Economy 70.2 (1962): 111-137.

Tennyson, S., 1997. Economic institutions and individual ethics: A study of consumer attitudes toward insurance fraud. Journal of Economic Behavior & Organization 32, 247–265.

Turing, Alan M. 1950. Computing machinery and intelligence. Mind.

Tesfatsion, L., 2002. Agent-Based Computational Economics: Growing Economies From the Bottom Up. Artificial Life 8, 55–82.

Tesfatsion, L., 1998. Preferential Partner Selection in Evolutionary Labor Markets: A Study in Agent-Based Computational Economics (Staff General Research Paper No. 2048). Iowa State University, Department of Economics.

Wilhite, A., 2001. Bilateral Trade and "Small-World" Networks. Computational Economics 18, 49–64.

Wilhite, A., Allen, W.D., 2008. Crime, protection, and incarceration. Journal of Economic Behavior & Organization 67, 481–494.

Appendix 1: Code for Chapter 2

RPSsimulate.R

Below is the code for the file that contains the core of model. It takes in many parameters that all default to the values used to generate the results. The diagnostic and output elements were removed to keep the code as brief as possible. The parameters mostly deal with where to put the output, but since the output code is removed that is irrelevant. In R, comments are begun with two #'s. In order to run these, the auxiliary functions must be run first.

```
#### this version has the coop at the start, outputs files to make movies
#### and has different data output


RPSsimulate = function(outfile = "results.csv",out_coop="coop.jpeg",out_law =
 "law.jpg",out_lie = "lie.jpeg",out_believe = "believe.jpeg",out_final
 ="summary.jpeg",out_final2 = "summary2.jpeg",out_final3 = "summary3.jpeg",iterations =
 100,bias_paper = 0,bias_rock = 0, bias_scissors = 0, mutations = TRUE,mutation_chance = 5,
mutate_scramble = FALSE, record = TRUE,payoff_win = 1, payoff_tie = 0,
payoff_lose = -1,global_learning = TRUE, allow_lawsuits = FALSE, can_cooperate = TRUE,
 delay_lawsuits = 1, turns_to_learn = 20, movie = FALSE,strat = "RANDOM",all_honest =
 FALSE, all_believe = FALSE,all_coop = FALSE,all_nosue = FALSE)

{



total = 500   ### the total number of players, must be even and greater than zero
##iterations = 1000                    ### how many rounds there will be

bias_paper = 0                ### set the bias if all are equal, no bias.  If one is set higher, initial
bias_rock = 0                 ### strategies will be biased towards that choice
bias_scissors = 0

learning = "winner teaches loser"                    ### set the type of learning by commenting
 out one (using ##'s)
learning = "most wins teaches least wins"
```

```
##payoff_win = 1
##payoff_tie = 0
##payoff_lose = -1

payoff_win_lawsuit = 2
payoff_lose_lawsuit = -2

##global_learning = TRUE
##allow_lawsuits = FALSE

can_cooparate = can_cooperate


will_coop1 = 0
will_coop2 = 0
will_sue = 0


recorded_parms = 14
kProbability = 1
kPast = 2
kPast_history = 1
kCounter = 3
kConstant = 4

######################
####### create players, initial settings

all_players = as.vector(1:total)

prob_rock = as.vector(rep(0,total))

prob_paper = as.vector(rep(0,total))

prob_scissors = as.vector(rep(0,total))


prob_rock_public = as.vector(rep(0,total))

prob_paper_public = as.vector(rep(0,total))

prob_scissors_public = as.vector(rep(0,total))

distance = as.vector(rep(0,total))

liechance_dec = as.vector(rep(0,total))
```

```
coopchance_dec = as.vector(rep(0,total))

believechance_dec = as.vector(rep(0,total))
lawchance_dec = as.vector(rep(0,total))

winnings = as.vector(rep(0,total))

prob_scissors = prob_scissors
orock = prob_rock
opaper = prob_paper
oscissors = prob_scissors
history_rock = as.vector(NULL)
history_paper = as.vector(NULL)
history_scissors= as.vector(NULL)
history_player1rock = as.vector(NULL)
history_player1scissors = as.vector(NULL)
history_player1paper = as.vector(NULL)

lie_mat = matrix(c(round(runif(I(total*7),min=0,max=1))),nrow = total, ncol = 7     )
believe_mat = matrix(c(round(runif(I(total*7),min=0,max=1))),nrow = total, ncol = 7     )
coop_mat = matrix(c(round(runif(I(total*7),min=0,max=1))),nrow = total, ncol = 7     )
law_mat = matrix(c(round(runif(I(total*7),min=0,max=1))),nrow = total, ncol = 7     )

if(all_honest)
{
 lie_mat = matrix(rep(0,I(7*total)),nrow = total, ncol = 7     )

}

if(all_believe)
{
 believe_mat = matrix(rep(1,I(7*total)),nrow = total, ncol = 7     )
}

if(all_coop)
{
 coop_mat = matrix(rep(1,I(7*total)),nrow = total, ncol = 7     )
}

if(all_nosue)
{
 law_mat = matrix(rep(0,I(7*total)),nrow = total, ncol = 7     )
}

info_mat =   matrix(c(rep(0,I(total*11))), nrow = total, ncol=11, byrow=TRUE,
```

```
            dimnames = list(c(),
 c("player","winnings","distance","lies","beliefs","liechance","believechance","coopchance","su
 echance","coops","suits")))

##################  THIS SECTION INITIALIZES THE PLAYERS CHARACTERISTICS

counter1 = 1

while(counter1<=total)
{
### EACH PROBABILITY IS CREATED FROM A RANDOM SAMPLE, PLUS A BIAS.
  THE EXPECTED VALUE IS 50 FOR EACH
p_paper = sample(0:100,1) + bias_paper
p_rock = sample(0:100,1) + bias_rock
p_scissors = sample(0:100,1) + bias_scissors
total_p = p_paper + p_rock + p_scissors

if(strat == "SAME") {
 p_paper = 10
 p_rock = 10
 p_scissors = 10
 total_p = 30
 }

### THEY ARE SUMMED UP AND EACH PROBABILITY IS A ROUNDED OUTCOME
  OF ITS SHARE OF THE TOTAL (TOTAL IS EXPECTED TO BE 150)
prob_paper[counter1] = round((p_paper/total_p)*100)
prob_scissors[counter1] = round((p_scissors/total_p)*100)
prob_rock[counter1] = 100-prob_paper[counter1] - prob_scissors[counter1]

#### calculate distance from ideal 33/33/33
d1 = prob_paper[counter1] - 33
d2 = prob_scissors[counter1] - 33
d3 = prob_rock[counter1] - 34
distance[counter1] = abs(d1) + abs(d2) + abs(d3)

### THE PROBABILITIES OF LYING AND BELIEVING ARE ALREADY CREATED IN
  BINARY FORM
#### THE BELOW CONVERTS THEM TO DECIMAL
#### EACH BINARY NUMBER HAS 7 SPACES, ,AND IS WEIGHTED BASED ON THE
  MAXIMUM VALUE
#### SEVEN WAS CHOSEN BECAUSE IT IS CLOSEST TO 100 AT MAXIMUM
believechance_dec[counter1] = BinDec(believe_mat[counter1,], l = 7)/BinDec(c(1,1,1,1,1,1,1),l
 = 7)
liechance_dec[counter1] = BinDec(lie_mat[counter1,], l = 7)/BinDec(c(1,1,1,1,1,1,1),l = 7)
coopchance_dec[counter1] = BinDec(coop_mat[counter1,], l = 7)/BinDec(c(1,1,1,1,1,1,1),l = 7)
```

```
lawchance_dec[counter1] = BinDec(law_mat[counter1,], l = 7)/BinDec(c(1,1,1,1,1,1,1),l = 7)


believechance_dec[counter1] = round(I(believechance_dec[counter1])*100)
liechance_dec[counter1] = round(I(liechance_dec[counter1])*100)
coopchance_dec[counter1] = round(I(coopchance_dec[counter1])*100)
lawchance_dec[counter1] = round(I(lawchance_dec[counter1])*100)


counter1 = counter1+1


}
############################################## DONE WITH THAT

### set up telegraphing
prob_paper_public = prob_paper
prob_scissors_public = prob_scissors
prob_rock_public = prob_rock
#####




############## EXCEL OUTPUT SETUP DONE
################################################

#########################################################################################
 ###################################

############## MAIN MAIN MAIN MAIN MAIN MAIN

#########################################################################################
 ###################################

counter2 = 1

while(counter2<=iterations)
{

 total_suits = 0  ## keeps track of lawsuits in each round
 plaint_wins = 0
 total_coops = 0
 failed_coops = 0
```

```
total_lies = 0
total_beliefs = 0
def_wins = 0

number_ties = 0
game_match = sample(1:total,total,replace = F)

### now must go through the matches and make them play each other

match_counter = 1


while(match_counter <= (total-1))
{
p1 = game_match[match_counter]
p2 = game_match[match_counter+1]
choice1 = ""
choice2 = ""
will_coop1 = 0
will_coop2 = 0
will_sue = 0
are_coop = 0
will_lie = 0
will_believe = 1

### player 1's choice, will become choice if other player not believed
choice_private = player_choice(prob_rock[p1],prob_paper[p1],prob_scissors[p1])

### now make a pick based upon telegraphed
###### LYING WILL HAVE TO GO HERE

prob_rock_public[p2] = prob_rock[p2]
prob_paper_public[p2] = prob_paper[p2]
prob_scissors_public[p2] = prob_scissors[p2]

##### PLAYER 2'S CHOICE, based on own strategy by default, will change if coop or lie
choice2 = player_choice(prob_rock[p2],prob_paper[p2],prob_scissors[p2])



will_lie = chance_probability(liechance_dec[p2])
### if lie, will telegraph a 100/0/0 type strategy, which one depends on the least likely strategy
if(will_lie >0)  ### zero is honest, 1 is dishonest
{
        total_lies = total_lies + 1
        lie1 = lie_strategy(prob_rock[p2],prob_paper[p2],prob_scissors[p2])
```

```
        prob_rock_public[p2] = lie1[1]
        prob_paper_public[p2] = lie1[2]
        prob_scissors_public[p2] = lie1[3]
        choice2 = player_choice(prob_rock[p2],prob_paper[p2],prob_scissors[p2])


}
#########################
####### COOPERATION  ####
#########################
if(will_lie<1)
{
        will_coop1 = chance_probability(coopchance_dec[p2])
        if(can_cooparate == FALSE){will_coop1 = 0}
        if(will_coop1>0)
        {
                prob_rock_public[p2] = 100
                prob_paper_public[p2] = 0
                prob_scissors_public[p2] = 0
                choice2 =
player_choice(prob_rock_public[p2],prob_paper_public[p2],prob_scissors_public[p2])
        }
}
### player 1's choice based on 2's public info
choice_public =
counter_opponent(prob_rock_public[p2],prob_paper_public[p2],prob_scissors_public[p2])
choice_public = player_choice(choice_public[1],choice_public[2],choice_public[3])

#########################################################
####### for now, main choice will be public choice
####### WEIGHTING WILL HAVE TO GO HERE
#########
will_believe = chance_probability(believechance_dec[p1])
total_beliefs = total_beliefs + will_believe

##will_believe = 1
if(will_believe>0){

choice1 = choice_public
##      total_beliefs = total_beliefs + 1
will_coop2 = chance_probability(coopchance_dec[p1])
if(can_cooparate == FALSE){will_coop2 = 0}
if(will_coop2 >0)
{
        if(will_coop2 == will_coop1) {
        total_coops = total_coops + 1
```

```
        choice1 =
player_choice(prob_rock_public[p2],prob_paper_public[p2],prob_scissors_public[p2])

        }
        if((will_coop2 != will_coop1)&(will_lie < 1))
        {
        choice2 = player_choice(prob_rock[p2],prob_paper[p2],prob_scissors[p2])
        choice1 =
player_choice(prob_rock_public[p2],prob_paper_public[p2],prob_scissors_public[p2])
        choice1 = choice2
        }
}

}
if(will_believe<1){
choice1 = choice_private
#########################################################
############ LAWSUIT DECISION HERE ##################
#########################################################

will_sue = chance_probability(lawchance_dec[p1])
if((will_sue>0)&(allow_lawsuits)&(counter2>=delay_lawsuits))  ## now must evaluate lawsuit
{
        total_suits = total_suits + 1 ## one more lawsuit

        if((prob_rock_public[p2]!=prob_rock[p2])&(prob_scissors_public[p2]!=prob_scissors[p2
])) ## if true, then p1 wins lawsuit
        {
                plaint_wins = plaint_wins + 1
                winnings[p1] = winnings[p1] + payoff_win_lawsuit
                choice1 = "plaint_win"
                winnings[p2] = winnings[p2] + payoff_lose_lawsuit
                choice2 = "def_lose"
        }


        if((prob_rock_public[p2]==prob_rock[p2])&(prob_scissors_public[p2]==prob_scissors[p
2])) ## if true, then p2 wins lawsuit
        {
                winnings[p1] = winnings[p1] + payoff_lose_lawsuit
                choice1 = "plaint_lose"
                choice2 = "def_win"
                winnings[p2] = winnings[p2] + payoff_win_lawsuit
                def_wins = def_wins + 1

        }
```

```
}
############## END LAWSUIT DECISION #################
#####################################################
## now if believe, I want them to maybe go for the tie


##choice1 =
player_choice(prob_rock_public[p2],prob_paper_public[p2],prob_scissors_public[p2])


}
## check to see if there was failed cooperation
if(will_coop1 != will_coop2){failed_coops = failed_coops + 1}
#####################################################################




#### ONLY EVALUATE OUTCOME IF A LAWSUIT DID NOT HAPPEN

if(will_sue <1)
{
        ### now must evaluate outcome
 if(choice1 == 'rock')
 {
        if(choice2 == 'rock'){
                winnings[p1] = winnings[p1] + payoff_tie
                winnings[p2] = winnings[p2]+ payoff_tie
                number_ties = number_ties + 1
                w1 = 0
                l1 = 0
        } ### tie
        if(choice2 == 'paper'){
                winnings[p1] = winnings[p1] + payoff_lose
                winnings[p2] = winnings[p2]+ payoff_win
                w1 = p2
                l1 = p1
                } ### loss
        if(choice2 == 'scissors'){
                winnings[p1] = winnings[p1]+ payoff_win
                winnings[p2] = winnings[p2]+ payoff_lose
                w1 = p1
                l1 = p2
        } ### win
```

```
} ### end one

if(choice1 == 'paper')
{
	if(choice2 == 'rock'){### win
		winnings[p1] = winnings[p1]+ payoff_win
		winnings[p2] = winnings[p2] + payoff_lose
		w1 = p1
		l1 = p2
	} ### win

	if(choice2 == 'paper'){### tie
		winnings[p1] = winnings[p1]+ payoff_tie
		winnings[p2] = winnings[p2] + payoff_tie
		number_ties = number_ties + 1
		w1 = 0
		l1 = 0

		} ### tie
	if(choice2 == 'scissors'){ ### loss
		winnings[p1] = winnings[p1]+ payoff_lose
		winnings[p2] = winnings[p2]+ payoff_win
		w1 = p2
		l1 = p1
	} ### loss
} ### end two

if(choice1 == 'scissors')
{
	if(choice2 == 'rock'){
		winnings[p1] = winnings[p1] + payoff_lose
		winnings[p2] = winnings[p2]+ payoff_win
		w1 = p2
		l1 = p1
	} ### loss
	if(choice2 == 'paper'){### win
		winnings[p1] = winnings[p1]+ payoff_win
		winnings[p2] = winnings[p2] + payoff_lose
		w1 = p1
		l1 = p2
		} ### win
	if(choice2 == 'scissors'){ ### tie
		winnings[p1] = winnings[p1] + payoff_tie
		winnings[p2] = winnings[p2] + payoff_tie
		number_ties = number_ties + 1
		w1 = 0
```

```
                l1 = 0

        } ### tie
}
        } ### end three
        ### AS OF NOW, INFO_MAT IS A TOTAL SUMMARY (I.E. SUMMARIZES
 EVERYTHING UP UNTIL CURRENT ROUND)
        #### THIS WILL BE USED TO DETERMINE LEARNING, AS IT IS EVENTUALLY
 SORTED BY FITNESS
        info_mat[p1,"player"] = p1
        info_mat[p2,"player"] = p2

        info_mat[p1,"distance"] = distance[p1]
        info_mat[p2,"distance"] = distance[p2]

        info_mat[p1,"winnings"] = winnings[p1]
        info_mat[p2,"winnings"] = winnings[p2]

        info_mat[p2,"lies"] = info_mat[p2,"lies"] + will_lie
        info_mat[p1,"beliefs"] = info_mat[p1,"beliefs"] + will_believe

        info_mat[p1,"liechance"] = liechance_dec[p1]
        info_mat[p2,"liechance"] = liechance_dec[p2]

        info_mat[p1,"believechance"] = believechance_dec[p1]
        info_mat[p2,"believechance"] = believechance_dec[p2]

        info_mat[p1,"coopchance"] = coopchance_dec[p1]
        info_mat[p2,"coopchance"] = coopchance_dec[p2]

        info_mat[p1,"suechance"] = lawchance_dec[p1]
        info_mat[p2,"suechance"] = lawchance_dec[p2]

        info_mat[p1,"coops"] = info_mat[p1,"coops"] + will_coop1
        info_mat[p2,"coops"] = info_mat[p2,"coops"] + will_coop2



        info_mat[p1,"suits"] = info_mat[p1,"suits"] + will_sue

############## END EVALUATION OF OUTCOME



 match_counter = match_counter+2
 }
```

```
history_rock = append(history_rock,mean(prob_rock))
history_paper = append(history_paper,mean(prob_paper))
history_scissors = append(history_scissors,mean(prob_scissors))

winner1 = which.max(winnings)
winnings1 = winnings[winner1]
loser1 = which.min(winnings)
losings1 = winnings[loser1]

}
################## DONE RECORDING


#############################################################################
 ####################
############# GENETIC ALGORITHM
 ###########################################################
############           LEARNING  HERE
 ############################################################################
######################### GENETIC ALGORITHM
 ###########################################
#############################################################################
 ####################

###### steps for setting up learning
##### create matrix for each parameter that will contain each agent's strategy
##### create a vector of 100, and each player gets represented more times with more wins
##### reproduction goes on this
##### then do cross over, mutation
#### repeat for each group

if(counter2 %% turns_to_learn == 0)
{


 counter5 = 1
max = 20
if(global_learning == TRUE) {max = total}
totalmax = total
row_c = 1
while(counter5<=(totalmax-(max-1)))

{
 end1 = counter5 +(max-1)
```

```
##rep_lie_matrix = matrix((rep(0,I(7*max))),nrow = max, ncol = 7)
##rep_believe_matrix = matrix((rep(0,I(7*max))),nrow = max, ncol = 7)
info_sorted = info_mat[sort.list(info_mat[,"distance"]),]


### REPLICATION
## now that we have an order of fitness (with most fit being at the end, time to pick)
fitness = info_sorted[counter5:end1,]
fitness_sorted = fitness[sort.list(fitness[,"winnings"]),]

order1 = c(fitness_sorted[,"player"])
c1 = 1
new_string = replicator_pop(order1)
new_string = sample(new_string,max,replace = T)

while(c1<=max)
{
        ##new_string = replicator(order1)

        lie_mat[order1[c1],] = lie_mat[new_string[c1],]
        believe_mat[order1[c1],] = believe_mat[new_string[c1],]
        coop_mat[order1[c1],] = coop_mat[new_string[c1],]
        law_mat[order1[c1],] = law_mat[new_string[c1],]


        c1 = c1 +1
}
##### replication done, now need cross over
##### CROSSOVER
crossover_match = sample(1:max,max,replace = F)

cross_counter = 1

while(cross_counter <= (max-1))
{
        s1 = order1[crossover_match[cross_counter]]
        s2 = order1[crossover_match[cross_counter+1]]


        lie1 = lie_mat[s1,]
        lie2 = lie_mat[s2,]

        believe1 = believe_mat[s1,]
        believe2 = believe_mat[s2,]

        coop1 = coop_mat[s1,]
```

```
coop2 = coop_mat[s2,]

law1 = law_mat[s1,]
law2 = law_mat[s2,]

string1 = c(lie1,believe1,law1,coop1)
string2 = c(lie2,believe2,law2,coop2)


crossover_point = sample(1:length(string1),1,replace = F)




cutstring1 = string2[crossover_point:length(string1)]
cutstring2 = string1[crossover_point:length(string1)]

string2[crossover_point:length(string1)] = cutstring2
string1[crossover_point:length(string1)] = cutstring1

##string1 = c(lie1,believe1,law1,coop1)

mutate = chance_probability(mutation_chance)  ### 3% chance
if(mutate>0)  ### if mutate
{
        mutate_match = sample(1:length(string1),1,replace = F)
        value_string = string1[mutate_match]
        if(value_string>0){string1[mutate_match] = 0}  ## flips them
        if(value_string<1){string1[mutate_match] = 1}

}

lie_mat[s1,]  = string1[1:7]
lie_mat[s2,] = string2[1:7]

believe_mat[s1,] = string1[8:14]
believe_mat[s2,] = string2[8:14]

law_mat[s1,] =string1[15:21]
law_mat[s2,] = string2[15:21]

coop_mat[s1,] = string1[22:28]
coop_mat[s2,] = string2[22:28]
```

```
        cross_counter = cross_counter + 2

}
###### CROSSOVER DONE


#### MUTATION DONE


############## now go to the next 10


counter5 = counter5 + max
}


counter1 = 1
###################### RECALCULATE DECIMAL VALUES
while(counter1<=total)
        {

                believechance_dec[counter1] = BinDec(believe_mat[counter1,], l =
7)/BinDec(c(1,1,1,1,1,1,1),l = 7)
                liechance_dec[counter1] = BinDec(lie_mat[counter1,], l =
7)/BinDec(c(1,1,1,1,1,1,1),l = 7)

                lawchance_dec[counter1] = BinDec(law_mat[counter1,], l =
7)/BinDec(c(1,1,1,1,1,1,1),l = 7)
                coopchance_dec[counter1] = BinDec(coop_mat[counter1,], l =
7)/BinDec(c(1,1,1,1,1,1,1),l = 7)


                believechance_dec[counter1] = round(I(believechance_dec[counter1])*100)
                liechance_dec[counter1] = round(I(liechance_dec[counter1])*100)

                coopchance_dec[counter1] = round(I(coopchance_dec[counter1])*100)
                lawchance_dec[counter1] = round(I(lawchance_dec[counter1])*100)

                counter1 = counter1+1


        }
```

```
##################################
##############################


counter2 = counter2 + 1
}

#############################################################################################
 ##################################

############## END END END END END END END

#############################################################################################
 ##################################

}
```

RPS auxiliary functions

In this appendix the helper functions that are called frequently by the RPS program are shown. The key ones is "BinToDec," which is necessary because in R there is no way to handle binary numbers in the same manner as languages such as C/C++ or Fortran. To manipulate and generate binary strings that can be spliced in the genetic algorithm the program puts the strings in vectors with each element representing a bit at a position. Another function that is used to generate random strings is "random_binary," which returns a random string of 1's and 0's. The function "counter_opponent" returns the best choice given an opponents strategy that is used if a player believes what another player is broadcasting. Similarly "lie_strategy" picks the optimal lie if the player decides to lie. Other functions perform tasks such as picking a choice given a probability of rock, paper, or scissors. Finally "replicator" handles the gene pool upon genetic learning.

```
#### interprets binary values
BinDec = function(inValue, l = 4)
{

 counter1 = l
 out_value = 0
 place_holder = 0
 while(counter1>0)
 {
        mul1 = 2^(place_holder)
        out_value = out_value + inValue[counter1]*mul1

        counter1 = counter1 -1
        place_holder = place_holder + 1
 }
 return(out_value)
}
```

```
player_choice = function(r_odds,p_odds,s_odds)
{
choices = rep('rock',r_odds)
choices = append(choices,rep('paper',p_odds))
choices = append(choices,rep('scissors',s_odds))
pick = sample(1:100,1,replace = T)
return(choices[pick])

}


##### random binary number generator
random_binary = function(size1)
{
counter1 = 1
out_string = ""

while(counter1 <= size1)
{
 to_add = round(runif(1,min=0,max=1))
 out_string = paste(out_string,to_add,sep ="")
 counter1 = counter1+1
}

 return(out_string)
}

##############################################################################
 ##############################
counter_opponent = function(o_rock,o_paper,o_scissors)
{
out_vec = c(0,0,0)

in_vec = c(o_rock,o_paper,o_scissors)
in_vec2 = in_vec

max1 = which.max(in_vec)
max3 = which.min(in_vec)

max2 = 6-(max1+max3)

if(max1 == 1) {out_vec[2] = in_vec[1]}
```

```
if(max1 == 2) {out_vec[3] = in_vec[2]}
if(max1 == 3) {out_vec[1] = in_vec[3]}

if(max2 == 1) {out_vec[2] = in_vec[1]}
if(max2 == 2) {out_vec[3] = in_vec[2]}
if(max2 == 3) {out_vec[1] = in_vec[3]}

if(max3 == 1) {out_vec[2] = in_vec[1]}
if(max3 == 2) {out_vec[3] = in_vec[2]}
if(max3 == 3) {out_vec[1] = in_vec[3]}

return(out_vec)
}
##########################################################################################
 ##################################
##### match on distance


chance_probability = function(in_chance)
{
choices = rep(1,in_chance)
choices = append(choices,rep(0,I(100-in_chance)))

pick = sample(1:100,1,replace = T)
return(choices[pick])

}


lie_strategy = function(o_rock,o_paper,o_scissors)
{

in_vec = c(o_rock,o_paper,o_scissors)
in_vec2 = in_vec

max1 = which.max(in_vec)
max3 = which.min(in_vec)

## if the minimum is rock (least likely to choose rock) I want opponent to pick paper
if(max3 == 1) {out_vec = c(100,0,0)}

## if the minimum is paper (least likely to choose paper) I want opponent to pick scissors
if(max3 == 2) {out_vec = c(0,100,0)}

## if the minimum is scissors (least likely to choose scissors) I want opponent to pick rock
if(max3 == 3) {out_vec = c(0,0,100)}
```

```
return(out_vec)
}


#######  REPLICATOR FUNCTION
### returns a position to replicate
###inVec = c(2,5,7,3,10,11,40,21,23,70)
replicator = function(inVec, remove = .3)
{
 size1 = length(inVec)

 ## create pool based upon following formula:
 ## sum up n*10 add together, chance of each is %

 c1 = remove*length(inVec)  ## cuts off bottom two
 pool = as.vector(NULL)

 while(c1<=size1)
 {
 pool = append(pool,rep(c1,I(c1*10)))
          c1 = c1 + 1
 }
 pick = sample(1:length(pool),1,replace = T)
 ##inVec[pool[pick]]
 return(inVec[pool[pick]])


}

#######  REPLICATOR FUNCTION

replicator_pop = function(inVec, remove = .3)
{
 size1 = length(inVec)

 ## create pool based upon following formula:
 ## sum up n*10 add together, chance of each is %

 c1 = remove*length(inVec)  ## cuts off bottom two
 pool = as.vector(NULL)

 while(c1<=size1)
 {
```

```
 pool = append(pool,rep(inVec[c1],I(c1)))
          c1 = c1 + 1
 }
 pick = sample(1:length(pool),1,replace = T)
 ##inVec[pool[pick]]
 return(pool)


}
```

Appendix 2 – Chapter 3 source code

Objective C Framework for genetic algorithms

```
//
//  NFGlobals.h
//  NFAgent
//
//  Created by Nathan Forczyk on 7/1/13.
//  Copyright (c) 2013 Nathan Forczyk. All rights reserved.
//

#import <Foundation/Foundation.h>
#import <stdbool.h>        // true/false
#import <stdint.h>         // UINT8_MAX
#import <stdio.h>          // fprintf
#import <stdlib.h>         // EXIT_SUCCESS
#import <string.h>         // strerror()




 #define _message_agent_init_ 0xFF00000000000000

 #define _update_genes_  [self createString]; \
                 [self interpretString];

#define parameters paramaters

#define _report_open_ -(void) report{
#define _report_close_ }

#define _defaultReport_ [super report];
//#define attach(x) (retain)

#define _close }

#define _get_pop_        printf( "Enter Number of Populations: "); \
scanf("%d",&numPopulation);
#define _get_agents_     printf("Enter Total Agents in population %d : ",i); \
scanf("%d",&counter1_reserved);

#define _get_duration_         printf("Enter Duration: "); \
scanf("%ld",&duration);

#define _endTick_open_ -(void)endTick{
#define _endTick_close_ }

#define _init_sim_open_ -(id) init{ \
pop_index = [[NSMutableArray alloc] init];\
int counter1_reserved = 0;
```

```
#define _init_sim_close_    int counter100; \
for(counter100 = 1; counter100<= duration;counter100++)\
{ \
[self endTick]; \
} \
return self; \
}

#define _handleMessage_open_ -(unsigned long)handleMessage{
#define _handleMessage_open -(unsigned long)handleMessage{

#define _handleMessage_close_ }
#define _handleMessage_close }


#define _send

#define _default_endTick_ [super endTick];

#define _match_open_ -(void)match{
#define _match_close_ }

#define _match_ [self match];

#define _send_message_(x,y) return_message = [(NFAgent *) x sendMessage:y];
#define _send_return_(x) [(NFAgent *) x sendMessage:return_message];

/// opens all agents
#define _allAgents_open_ int i; \
for(i = 0; i <totalAgents; i++) \
{ \
NFAgent *allAgents = (NFAgent *)[pop_index objectAtIndex:i];

#define _allAgents_close }
#define _allAgents_close_ }


//subset open
#define _subsetAgents_open_(x,y)  \
for(y = 0; y <[x count]; y++) \
{ \
NFAgent *subsetAgents = (NFAgent *)[x objectAtIndex:y];

#define _subsetAgents_close  }
#define _subsetAgents_close_ }

//following defines object creation and declaration
#define _open_describe_agent_(x) @interface x : NFAgent
#define _open_describe_traits_ {
#define _evolved_traits_
#define _close_describe_traits_ }
#define _nonevolved_traits_
#define _unique_methods_
#define _end_unique_methods_
#define _close_describe_agent_(x) @end
```

```
#define _open_describe_simulation_(x) @interface x : NFAgentSimulation
#define _close_describe_simulation_(x) @end

#define _open_describe_population_(x) @interface x : NFPopulation
#define _close_describe_population_(x) @end


#define _open_implementation_(x) @implementation x
#define _close_implementation_(x) @end


#define _attach_agent_type_(x,z,w,genetic_,protect_,muterate_,population_id_) int ii; \
NSMutableArray *population = [[NSMutableArray alloc] init]; \
for ( ii = 0; ii < counter1_reserved; ii++) { \
x *toAdd = [[x alloc] init]; \
[toAdd setAgentID:ii];\
[population addObject:toAdd]; \
} \
z * pop_add = [ [ z alloc] initWithCustom:population andTurns:w andGenetic:genetic_ andProtect:protect_
andMutationRate: muterate_]; \
[pop_add setPopID:population_id_]; \
[pop_index addObject:pop_add];\
numAgents = numAgents + counter1_reserved;


//z = [[NFPopulation alloc] initWithCustom:population andTurns:w];

#define _attach_agent_type_no_learn_(x,y)


#define _simulation_run_(x) int main(int argc, const char * argv[]) \
{ \
@autoreleasepool { \
\
x *toRun = [ [x alloc] init]; \
\
[toRun report_sim]; \
\
}\
return 0;\
}
```

```
//
//  NFAgent.h
//  NFAgent
//
//  Created by Nathan Forczyk on 6/22/13.
//  Copyright (c) 2013 Nathan Forczyk. All rights reserved.
//

#define BYTE_EIGHT 56
```

```
#define BYTE_SEVEN 48
#define BYTE_SIX 40
#define BYTE_FIVE 32
#define BYTE_FOUR 24
#define BYTE_THREE 16
#define BYTE_TWO 8
#define BYTE_ONE 0
#define PARMS 8

#define GET_BYTE(X,Y)  (X>> ((Y -1) * 8)) &0xFF    //returns byte Yth position from variable X  bits 0-7 are
byte 1, and so on



@interface NFAgent : NSObject
{
   NSMutableArray *to_encode;
   unsigned long info_string;
   int num_codes;
   Byte used_parms;
   unsigned long genetic_string;
   unsigned long _population_id; // keeps track of population id
   unsigned long _agent_id; //keeps track of agents location in population
   unsigned char paramaters[PARMS];  //outputs values extracted from string
   double fitness;       // fitness
   int string_size;   //size of genetic string
   int message_size;  //size of message in bits
   unsigned int x,y,z;  //for spatial abilities
   unsigned char message[8]; // contains the spliced message
   unsigned long raw_message; // contains the raw prescpliced message
   char markers[8]; //area to leave markers, reset at end of every tick
   unsigned long return_message;
   void * genetic_code;
   FILE *to_write;
}



-(void)report;
-(void)printBits:(unsigned long) inBits;
-(void)createString;
-(void)interpretString;
-(void)randomize;
-(void)endTick;
-(void)resetMarkers; //resets temporary markers to zero, low level
-(char)getMarker:(int) which;
-(void)setMarker:(int) which;
-(unsigned long)sendMessage:(unsigned long) inMessage;
-(unsigned long)handleMessage;
-(unsigned long)getString;
-(void) setString:(unsigned long) inString;
-(void) clearFitness;
-(double)getFitness;
-(Byte) getTotalParms;
-(unsigned long)getGenes;
-(void) setGenes:(unsigned long) inGenes;
```

```
-(void) setPopID:(unsigned long) inID;
-(unsigned long) getPopID;
-(void) setAgentID:(unsigned long) inID;
-(unsigned long) getAgentID;
@end




#define _endTick_open_ -(void)endTick{
#define _endTick_close_ }

@implementation NFAgent



-(id) init
{
   self = [super init];
   string_size = 64;
   message_size = 8;
   info_string = 0;
   unsigned int  v1 = ((unsigned int)random() % 0xFFFFFFFF) + 1;
   unsigned int  v2 = ((unsigned int)random() % 0xFFFFFFFF) + 1;

   info_string = (unsigned long) v1;
   info_string = info_string << 32;

   info_string = info_string | (unsigned long) v2;
   // [self randomize];

   return self;

}

//replace with method that will be called whenever the agent is asked to report its info
-(void) report{
   printf("Agent %lu in Population %lu \n", _agent_id,_population_id);
   return;
}

-(void) printBits:(unsigned long) inBits{

   int i;
   unsigned char out;


   for(i = (string_size - 1); i>= 0;i--)
   {
      out = inBits >> i;
      printf("%d",out&0x01);

   }
```

```
}

//creates a string from a paramter array
-(void)createString{

    unsigned long outString;
    unsigned long workString;
    Byte workInfo;
    outString = (unsigned long)0;
    workString = (unsigned long)0;


    int i;

    for(i = used_parms; i >0; i--)
    {
        outString = outString << 8;
        workInfo = paramaters[(i-1)];

        workString = (unsigned long)workInfo;
        outString = outString | workString;

    }

    info_string = outString;

}

//randomizes a 64 bit string
-(void)randomize{

    int counter1 = 0;
    unsigned int  v1;
    unsigned long bit_changer;


    while(counter1 <string_size)
    {

        v1 = ((int)random() % 100) + 1;

        if(v1>50)
        {

            bit_changer = 2^counter1;
            info_string = info_string | bit_changer;
        }

        counter1++;

    }



}
```

```
//interprets the string that is already stored and splices it into an 8 byte array
-(void)interpretString{




   int i;
 //  int which_parm = PARMS;

   for(i = 1; i<PARMS; i++)
   {
     /* printf("\nBeing called \n");
      printf(" i is %d \n",i);
      printf(" parms is %d \n", PARMS);
      */
      paramaters[i-1] = (unsigned char)GET_BYTE(info_string,i);
      //printf("\n %d \n",paramaters[i]);
   }


}
//
-(void)endTick{
   [self resetMarkers];
}

//reset markers
-(void)resetMarkers{
   int i;
   raw_message = 0;

   for(i=0; i<8;i++)
   {
      markers[i] = 0; //resets markers
      message[i] = 0;

   }

}

//
-(char)getMarker:(int) which
{

   return markers[which];
}

//
-(void) setMarker:(int)which
{
   markers[which] = 1;
}

//should be left alone, handles the splicing of messages
-(unsigned long)sendMessage:(unsigned long) inMessage{
```

```
    unsigned long return_message1 = 0;
    raw_message = inMessage;
    int i;

    // printf("\n Bits for in Message:");
    // [self printBits:raw_message];
    // printf("\n");
    for(i = 0; i<8;i++)
    {

        message[i] = GET_BYTE(inMessage,(i+1));

        //   printf("Bits for message[i] %d   ", i);
        //   [self printBits:message[i]];
        //   printf("\n");
    }
    return_message1 = [self handleMessage];
    return(return_message1);
}


-(unsigned long)handleMessage{

    unsigned long return_message1 = 0;
    //should be overridden


    return return_message1;


}

-(unsigned long)getString{

    return info_string;
}

-(void)setString:(unsigned long) inString
{
    info_string = inString;

}

-(void)clearFitness
{
    fitness = 0;
}

-(double)getFitness
{
    return fitness;
}

-(Byte) getTotalParms{
    return(used_parms);
```

```
}

-(unsigned long) getGenes
{
   return(info_string);
}

-(void) setGenes:(unsigned long) inGenes
{
   info_string = inGenes;
}



-(void) setPopID:(unsigned long) inID
{
      _population_id = inID;
}

-(unsigned long) getPopID
{

   return(_population_id);
}

-(void) setAgentID:(unsigned long) inID
{

   _agent_id = inID;
}

-(unsigned long) getAgentID
{
   return(_agent_id);
}



@end



//
//  NFPopulation.h
//  NFAgent
//
//  Created by Nathan Forczyk on 6/22/13.
//  Copyright (c) 2013 Nathan Forczyk. All rights reserved.
//



@interface NFPopulation : NSObject
{

   /// NFLookUp *lookup;
```

```
    int totalAgents;
    NSMutableArray *pop_index;
    int turns_to_learn;
    Byte use_genetic;
    Byte guarantee; //the top # of these will pass on
    int mutation_rate; //char, not a decimal
    int turn_tracker;
    unsigned long return_message;
    unsigned long PROTECT;
    unsigned long *pool;
    unsigned long _population_id;
    unsigned long pool_size_;
    int chance_mutation;
    NSMutableArray *replace_index; //holds index of agents yet to be picked, for replacement pairing
    FILE *to_write;

}

@property (assign) unsigned long population_id;

-(void)initAgents:(int)numAgents;
-(id)initWithValues:(int)numAgents andTurns: (int) turns;
-(id)initWithCustom: (NSMutableArray *) list andTurns:(int)turns andGenetic:(int)is_genetic
andProtect:(int)to_protect andMutationRate: (int)mute_rate;
-(void)report;
-(void)endTick;
-(void)learnGenetic;
-(NFAgent *)getRandomAgent:(char)replace;
-(int )getMaxFit:(NSMutableArray *) inArray; //returns the max fit object
-(void)match;
-(void)setParameters:(unsigned long)inProtect andMutationChance: (int)inChance;
-(void) setPopID:(unsigned long) inID;
-(unsigned long) getPopID;

@end

//
//  NFPopulation.m


#define _endTick_open_ -(void)endTick{
#define _endTick_close_ }

@implementation NFPopulation

@synthesize population_id = _population_id;

-(id) init
{
    self = [super init];


    if(self)
    {
        turn_tracker = 1;
```

```
    }

    return self;

}

-(id) initWithValues:(int)numAgents andTurns:(int) turns
{

    self = [super init];
    turns_to_learn = turns;
    turn_tracker = 1;
    if(self)
    {
        totalAgents = numAgents;
        [self initAgents:totalAgents];

    }
    return self;
}

-(void)initAgents:(int)numAgents
{
    int i;
    pop_index = [[NSMutableArray alloc] init];

    for (i = 1; i <=totalAgents; i++)
    {
        NFAgent *toAdd = [[NFAgent alloc] init];
        [pop_index addObject:toAdd];
    }

}

//report values
-(void)report
{
    int i;

    for(i = 0; i <totalAgents; i++)
    {
        NFAgent *toReport = (NFAgent *)[pop_index objectAtIndex:i];
        [(NFAgent *)toReport report];
    }

}

//called at end of tick, merely calls all its members endTicks, which  does nothing if left to itself
-(void)endTick
{
    int i;

    for(i = 0; i <totalAgents; i++)
    {
        NFAgent *toReport = (NFAgent *)[pop_index objectAtIndex:i];
```

```
        [(NFAgent *)toReport endTick];
    }

    //learning goes here
    if((turn_tracker % turns_to_learn) == 0)
    {
        if(use_genetic >0) // if genetic learning is turned on
        {

            [self learnGenetic];

        }
        //here means time to learn.
    }

    turn_tracker++;
    ////

    replace_index = nil;

}

//initializes with custom agents, should be used
-(id)initWithCustom: (NSMutableArray *) list andTurns:(int)turns andGenetic:(int)is_genetic andProtect:
(int)to_protect andMutationRate: (int)mute_rate
{
    self = [super init];
    use_genetic =is_genetic;
    turns_to_learn = turns;
    turn_tracker = 1;
    mutation_rate = mute_rate;
    chance_mutation = mute_rate;
    guarantee = to_protect;

    pop_index = (NSMutableArray *) list;
    totalAgents = (int)[pop_index count];

    return(self);
}

//
-(NFAgent *) getRandomAgent:(char) replace{
    NFAgent *outAgent = nil;

    int v1;

    if(replace == 0) //no replacement
    {
        v1 = ((int)random() % totalAgents) + 1;
        outAgent = (NFAgent *)[pop_index objectAtIndex:(v1-1)];
    }
    if(replace >0) //replacement
    {
        if(replace_index == nil)
        {
```

```
            //go here because this is the first pick, no index left
            replace_index = pop_index;
        }
        v1 = ((int)random() % [pop_index count]) + 1;
        outAgent = (NFAgent *)[replace_index objectAtIndex:(v1-1)];
        [replace_index removeObjectAtIndex:(v1-1)];



    }

    return outAgent;
}

///// learnGenetic
-(void)learnGenetic
{
    NSMutableArray *tracker;
    NSMutableArray *sorted;
    NFAgent *toAdd;
    int i=0;
    int location;
    int total_parms;
    //tracker = pop_index;

    sorted = [[NSMutableArray alloc] init];
 //   tracker = [[NSMutableArray alloc] init];
    tracker = [pop_index mutableCopy];


    for(i = 0; i < [pop_index count]; i++)
    {
        location = [self getMaxFit:tracker];
        toAdd = (NFAgent *) [tracker objectAtIndex:location];

        [sorted addObject:toAdd];
        [tracker removeObjectAtIndex:location];
        if(i == 1){
            total_parms = [toAdd getTotalParms];
        }
    }
    // by this point, sorted contains a list of pointers to agents in the
    // population that is ordered on fitness with 0 being the most fit
    // as of now, first agent is protected from death and reproduction/mutation


    int max_used_bits = total_parms * 8;
    unsigned long *genetic_info = malloc(sizeof(unsigned long) * [sorted count]);
    unsigned long *new_genetic_info = malloc(sizeof(unsigned long) * [sorted count]);

    int agent_pool_size = (int)[sorted count]; //size of the genetic pool

    /*
        GENETIC LEARNING GOES HERE, AGENTS ORGANIZED BY FITNESS

        description:
```

```
      total_parms has the used byte length of the genetic code (max 8 or 64 bits).
   uses bitwise manipulators to only deal with a subset of the 64 bits if necessary
 */
#define POOL_SIZE agent_pool_size
int MAX_BIT  = max_used_bits - 1;
#define MIN_BIT 0
unsigned long parent1=0,parent2=0;
int random1,random2;
unsigned long bit_mask = ~0; //will control which areas we won't zero out on exit
unsigned long new_string,parent_string1,parent_string2;
unsigned int cross_point;

bit_mask = bit_mask >> sizeof(bit_mask)*8 - max_used_bits;
//bit_mask2 = ~bit_mask;
_subsetAgents_open_(sorted,i)
genetic_info[i] = [subsetAgents getGenes];
_subsetAgents_close_

//went through and got all the genes

unsigned long local_counter;

if(!pool)
{
pool_size_ = POOL_SIZE/2 * ((2*1) + (POOL_SIZE -1)*1);

pool = malloc(sizeof(pool_size_) * pool_size_);

unsigned long counter55 = 0;
unsigned long outVal = 1;
unsigned long local_counter55 = 1;
unsigned long num_times = 1;

while(counter55<pool_size_)
{

   int local_counter = 0;
   num_times = POOL_SIZE - (outVal-1);

   while(local_counter <num_times)
   {
   pool[counter55++] = outVal;
   local_counter++;
   }


   outVal++;
}
}


for(local_counter = 0;local_counter<guarantee;local_counter++)
{
   new_genetic_info[local_counter] = genetic_info[local_counter];
}
```

```
 for(local_counter = guarantee; local_counter < POOL_SIZE;local_counter++)
 {

       random1 = ((int)random() % pool_size_) + 1;
       random2 = ((int)random() % pool_size_) + 1;

       parent1 = pool[--random1];
       parent2 =  pool[--random2];



    parent_string1 = genetic_info[parent1];
    parent_string2 = genetic_info[parent2];
    cross_point = ((int)random() % MAX_BIT) + 1;
    unsigned long left_side = ((bit_mask >> cross_point) << cross_point) & parent_string1;;

    unsigned long right_side = ~((bit_mask >> cross_point) << cross_point) & parent_string2;

    new_string = left_side | right_side;

    random1 = ((int)random() % 100) + 1;

    if(random1 >= (100 - chance_mutation) )
    {
       //mutation

       random2 = ((int)random() % max_used_bits) + 1;
       random2--;
       unsigned long mutation_mask = pow(2,random2);

       new_string = new_string ^ mutation_mask;




    }


    new_genetic_info[local_counter] = new_string;



   parent1 = parent2 = 0;



 }

 //now sorted will contain a list of most wins

// printf("%ld",genetic_info[2]);

  _subsetAgents_open_(sorted,i)
//  [subsetAgents report];
  [subsetAgents setGenes:new_genetic_info[i]];
  //printf("\n\n Called new genes %lu set genes %lu \n",new_genetic_info[i],[subsetAgents getGenes]);
```

```
    [subsetAgents interpretString];
    _subsetAgents_close_


}

-(int )getMaxFit:(NSMutableArray *) inArray
{
    NFAgent *checkAgent;
    NFAgent *maxAgent;
    double maxFit;
    double checkFit;
    int location;

    int i;

    for(i = 0; i< [inArray count];i++)
    {
        checkAgent = (NFAgent *)[inArray objectAtIndex:i];
        checkFit = [(NFAgent *)checkAgent getFitness];
        if(i ==0)
        {
            maxAgent = checkAgent;
            maxFit = [(NFAgent *)maxAgent getFitness];
            location = i;
        }
        else
        {
            if(checkFit > maxFit)
            {
                maxFit = checkFit;
                maxAgent = checkAgent;
                location = i;
            }
        }

    } //end for loop

    return location;
}


//matching function
-(void)match{


}


-(void)setParameters:(unsigned long)inProtect andMutationChance:(int)inChance
{
    PROTECT = inProtect;
    chance_mutation = inChance;

}
```

```
-(void) setPopID:(unsigned long) inID
{
      _population_id = inID;
      _allAgents_open_
         [allAgents setPopID:inID];
      _allAgents_close_
}

-(unsigned long) getPopID
{

   return(_population_id);
}


@end


//
//  NFAgentSimulation.h
//  NFAgent
//
//  Created by Nathan Forczyk on 6/22/13.
//  Copyright (c) 2013 Nathan Forczyk. All rights reserved.
//




@interface NFAgentSimulation : NSObject

{

   long numAgents; // number of agents
   int numPopulation; //number of populations
   long duration; // how many ticks the simulation will run
   NSMutableArray *pop_index;
   FILE *to_write;

}

-(void)setAgents: (long)num1;
-(void)setPopulation:(int)num2;
-(void)setDuration:(long)num3;
-(void)initPop:(int)popSize;
-(id)initWithValues:(long)total1 andPop:(int)pop1 andDuration:(long)duration1;
-(void)report_sim;
-(void)endTick;
-(void)match;
@end



//
//  NFAgentSimulation.m
//  NFAgent
```

```
//
//  Created by Nathan Forczyk on 6/22/13.
//  Copyright (c) 2013 Nathan Forczyk. All rights reserved.
//


#define _endTick_open_ -(void)endTick{
#define _endTick_close_ }

@implementation NFAgentSimulation


-(id)init
{
   int i;
   pop_index = [[NSMutableArray alloc] init];

   self = [super init];



   for(i = 1; i<= duration;i++)
   {
      [self endTick];
   }

   return self;
}


-(id)initWithValues:(long)total1 andPop:(int)pop1 andDuration:(long)duration1
{

   self = [super init];

   if(self){
      [self setAgents:total1];
      [self setPopulation:pop1];
      [self setDuration:duration1];
   }

   return self;
}

-(void)initPop:(int)popSize
{
   int i;
   pop_index = [[NSMutableArray alloc] init];

   for (i = 1; i <=popSize; i++)
   {
      NFPopulation *toAdd = [[NFPopulation alloc] init];
      [pop_index addObject:toAdd];
   }

}
```

```
//called at end of tick, merely calls all its members endTicks, which  does nothing if left to itself
-(void)endTick
{
   int i;

   for(i = 0; i <numPopulation; i++)
   {
      NFPopulation *toEnd = (NFPopulation *)[pop_index objectAtIndex:i];
      [(NFPopulation *)toEnd endTick];
   }

}



//replace with method to be called to report overall simulation info
-(void)report_sim
{

   int i;

   for (i = 0; i <numPopulation; i++)
   {
      NFPopulation *toReport = (NFPopulation *)[pop_index objectAtIndex:i];

      [toReport report];
   }

}

-(void)setAgents:(long)num1
{

   numAgents = num1;


}

-(void)setPopulation:(int)num2
{
   numPopulation = num2;

}

-(void)setDuration:(long)num3
{
   duration = num3;
}

-(void)match
{
}
@end
```

Objective C code for Chapter 4

The technical approach for Chapter 4's model is different despite the two making use of genetic algorithms. First, an application framework was created to streamline their usage. Taking advantage of similarities that all GA's in economics have, much of the redundant code is eliminated.

In addition, the framework makes use of modern programming techniques. While object oriented programming (OOP) is a feature of C++, it is entirely possible to make a C++ program that avoids some or all of the OOP abilities. Objective-C is the language used here and it also makes use of some of Mac OS X's "Cocoa" framework as well. The Cocoa framework has functions for sorting arrays that are useful in GA's. Furthermore, the core of the framework can be expanded to allow display abilities similar to R by making use of Cocoa.

The framework makes two assumptions. Every GA in ABM consists of one or more populations consisting of agents. Also it assumes a timing mechanism and at each interval the model control object sends a message to its populations. Most GAs in ABM are single population. The population object in turn controls its constituent agents. It is possible to allow populations to communicate with each other. The core is a messaging system that is common to all populations and agents. Messages are sent from control object, to population, and finally to its constituent agents.

In addition, the framework makes heavy use of what is called a "preprocessor." This allows Objective-C to be somewhat customized so as to make it more readable to reviewers. For instance the evolved traits of each agent are apart from the non-evolved traits. Also, anytime a population needs to communicate with all its constituent agents, the code is _allAgents_open_ and the preprocessor is able to replace that with the correct Objective-C code.

The code is organized as follows. First is the agent object declaration. Its unique traits and its methods are declared. Next is the population object declaration that handles interactions between its agents. The control object tells the population which agents it has and how many. The population handles the trading between the agents and the agents up date their own utility after the trades.

Upon running, the program asks the user how many groups there are, and how many agents in each group there are, how many groups there are, if the ends of each group overlaps, and what percentage of agents can seek cross overs. The parameters of the final model run where each agent was its own group and could decide how many trading partners it had were 500 agents in one group, with no overlap and 100% crossover.

```
//
//  main.m



#import "TSSimulation.h"

#define death TRUE
#define random_utility TRUE
#define utility_grouping FALSE

#define outputfile "     deathTRUE20.txt"
#define agentfile  "agentDEATHTRUE20.txt"
/* local constants
 */

#define AGENT_SIZE 500


#define _get_groups_  printf("Input Number of Groups: \n"); \
scanf("%d",&num_groups);

#define _get_agents_in_group_ printf("Agents in Each Group: \n"); \
scanf("%d", &agents_in_each_group);

#define _get_overlap_ printf("Do end agents overlap? 0 = No, > 0 = Yes \n");\
scanf("%d",&end_agents_overlap);

#define _get_cross_overs_ printf("Input Number of Crossovers: \n");\
scanf("%d",&cross_overs);
```

```
//define messages here


///// AGENT

_open_describe_agent_(TestAgent)

   _open_describe_traits_

   _nonevolved_traits_
      unsigned int g1;
      double g2;
      double utility;
      double previous_utility;
      unsigned int group;
      float alpha,beta;
      double MRS; // equals g2^i / g1 ^ i
      bool is_crossover;
      unsigned long cross_over_partner;
      Byte cross_overs_total;
      unsigned long number_of_searches_in_round;
   _evolved_traits_
      Byte meaningless_evolved_trait;
      Byte number_of_crosses;

   _close_describe_traits_

   _unique_methods_
      -(unsigned int)getG1;
      -(double)getG2;
      -(void)addG1:(int)to_increment;
      -(void)addG2:(double)to_increment;
      -(bool)isCross;
      -(void)setCross:(bool)cross;
      -(int)getNumberCross;
      -(void)setNumberCross:(int)inCross;
      -(void)incrementSearches;
      -(double)get_utility;
      -(float)get_alpha;
      -(float)get_beta;
   _end_unique_methods_

_close_describe_agent_(TestAgent)

_open_implementation_(TestAgent)


   -(float)get_alpha
   {
      return(alpha);
   }

   -(float)get_beta
   {
      return(beta);
```

```
}

-(double)get_utility
{
   utility = pow(g1,alpha) * pow(g2,beta);
   MRS = (alpha * pow(g1,alpha-1) * pow(g2,beta)) / ( beta * pow(g1, alpha) * pow(g2,beta-1) );

   return(utility);
}

-(void)incrementSearches
{
   number_of_searches_in_round++;
}
-(bool)isCross
{
   return(is_crossover);

}

-(void)setCross:(bool)cross
{
   is_crossover = cross;
   [self setNumberCross:0];

}

-(unsigned int)getG1
{
   return g1;
}

-(double)getG2
{
   return g2;
}

-(void)addG1:(int)to_increment
{
   g1 += to_increment;
}

-(void)addG2:(double)to_increment
{
   g2 +=to_increment;
}

-(int)getNumberCross
{
   return(cross_overs_total);
}

-(void)setNumberCross:(int)inCross
{
   cross_overs_total = inCross;
}
```

```
_endTick_open_
   utility = pow(g1,alpha) * pow(g2,beta);
   //MRS = (alpha * pow(g1,alpha-1) * pow(g2,beta)) / ( beta * pow(g1, alpha) * pow(g2,beta-1) );
   MRS = (alpha/beta) * (g2/g1);
   FILE *agent_file = fopen(agentfile,"a");
   fitness = (utility - previous_utility)-  pow(number_of_searches_in_round,2) ;
   fprintf(agent_file,"agent: %lu fitness is: %f searches are: %li utility change is: %f \n",_agent_id,fitness,
   number_of_searches_in_round, (utility - previous_utility));
   fclose(agent_file);
   _update_genes_
   if(is_crossover)
   {
   number_of_crosses = paramaters[0];
   // number_of_crosses = ((float)number_of_crosses/(float)255 * );
   cross_overs_total = number_of_crosses;
   }

   if(death)
   {
      int prob_death = ((unsigned int) random()% 100) + 1;
      if(prob_death < 3) //2% chance of death
      {
         alpha = ((unsigned int)random() % 100) + 1;
         alpha = --alpha/100;
         if(alpha == 0) {alpha = .01;}
         if(alpha == 1) {alpha = .99;}
         beta = 1-alpha;
      }

   }

   _default_endTick_ //calls the agent default end tick
   number_of_searches_in_round = 0;
   previous_utility = utility;
_endTick_close_

_report_open_
   utility = pow(g1,alpha)*pow(g2,beta);
   MRS = (alpha/beta) * g2/g1;
   FILE *agent_file = fopen(agentfile,"a");
   fprintf(agent_file,"Agent %lu in Population %lu number of g1: %i number of g2: %f utility: %f MRS: %f
Number of Crosses: %i, alpha: %f, beta %f \n",
   _agent_id,_population_id,g1,g2,utility,MRS,number_of_crosses,alpha,beta);
   fclose(agent_file);

   //_defaultReport_
_report_close_

_handleMessage_open_

   return_message = 0x0;
   switch(raw_message)
      {
         case _message_agent_init_:
            used_parms = 1;
```

```
            parameters[0] = 0;
            _update_genes_
            number_of_crosses = paramaters[0];
            number_of_searches_in_round = 0;
            g1=0x0; g2 = 0x0;
            to_write = stdout;

            while( !( g1>=10) )
            {g1 = ((unsigned int)random() % 1500) + 1;}

            while( !( g2>=10) )
            {g2 = (random() % 1500) + 1;}


            if(!random_utility)
            {
               alpha = 1;
               beta = 1;
               utility = pow(g1,alpha) * pow(g2,beta);
               MRS = g2 / g1;


            }

            if(random_utility)
            {
               if(utility_grouping)
               {


               }
               else
               {
                  alpha = ((unsigned int)random() % 100) + 1;
                  alpha = --alpha/100;
                  if(alpha == 0) {alpha = .01;}
                  if(alpha == 1) {alpha = .99;}
                  beta = 1-alpha;
                  utility = pow(g1,alpha) * pow(g2,beta);

               }

            }

            previous_utility = utility;
         break;

         default:
         break;
      }

   return return_message;
_handleMessage_close_
```

```
_close_implementation_(TestAgent)


////////////////////////////////////// POPULATION
_open_describe_population_(TestPopulation)
      _open_describe_traits_
         int num_groups;
         int agents_in_each_group;
         int cross_overs;
         int end_agents_overlap;
         unsigned long number_of_trades_per_round;
         unsigned long number_of_trades_total;
         unsigned long number_of_searches_per_round;
         unsigned long number_of_searches_total;
      _close_describe_traits_

      _unique_methods_

         -(void)trade;
         -(void)setParms:(int)ngroups andNumInEach: (int)agentsineach andCross: (int)c_overs andOverlap:
(int)endOverlap;
         -(int)getMaxPrice:(NSMutableArray *) inArray andG1: (int) ing1 andG2: (double) ing2 andAlpha:
(float)inAlpha andBeta: (float)inBeta;
       _end_unique_methods_


_close_describe_population_(TestPopulation)


_open_implementation_(TestPopulation)

   -(void)setParms:(int)ngroups andNumInEach: (int)agentsineach andCross: (int)c_overs andOverlap:
(int)endOverlap
   {
      num_groups = ngroups; agents_in_each_group = agentsineach; cross_overs = c_overs; end_agents_overlap =
endOverlap;


   }

   -(int)getMaxPrice:(NSMutableArray *) inArray andG1: (int) ing1 andG2: (double) ing2 andAlpha: (float)inAlpha
andBeta: (float)inBeta
   {
      NFAgent *checkAgent;
      NFAgent *maxAgent;
      double maxFit;
      double maxPrice;
      int location;
      double checkPrice;
      double in_MRS = (inAlpha / inBeta) * (ing2/ing1);

      int i;

      for(i = 0; i< [inArray count];i++)
      {
```

```
        checkAgent = (NFAgent *)[inArray objectAtIndex:i];

        int bG1 = [checkAgent getG1];
        double bG2 = [checkAgent getG2];
        float alpha1, alpha2, beta1,beta2;

        alpha1 = [checkAgent get_alpha];
        beta1 = [checkAgent get_beta];

        double opp_MRS = (alpha1/beta1) * (bG2/bG1);
        checkPrice = (in_MRS + opp_MRS) /2;
        //checkPrice = (bG2 + ing2)/(bG1 + ing1);


        if(i ==0)
        {
           maxAgent = checkAgent;
           maxPrice = checkPrice;
           location = i;
        }
        else
        {
           if(checkPrice > maxPrice)
           {
              maxPrice = checkPrice;
              maxAgent = checkAgent;
              location = i;
           }
        }

    } //end for loop

    return location;
}


/* handles matchin and trading */
-(void)trade
{

    /* steps

        1: calculate MRS
        2: rank others by lowest price to highest
        3: exchange with lowest until utility reduced
        4: price is g2 + g2 / (g1 + g1)
    */
    number_of_trades_per_round = 0;
    number_of_searches_per_round = 0;
    int group_counter = 1;
    int agents_per_group = totalAgents/num_groups;
    int previous, next;
    /*
        must also determine trade groupings.
    */
    while(group_counter <= num_groups)
```

```
    {

        NSMutableArray *sub_group = [[NSMutableArray alloc] init];
        int local_counter = 0;
        while(local_counter<agents_per_group)
        {
            NFAgent *local_ad = (NFAgent *) [pop_index objectAtIndex:((group_counter-1)*agents_per_group) +
local_counter];
            [sub_group addObject:local_ad];
            local_counter++;
        }
         if(end_agents_overlap)
         {
           //keeps track over overlap
          (group_counter>1)?(previous = ((group_counter-1)*agents_per_group -1)):(previous = (totalAgents -1));
          (group_counter<num_groups)?(next = ((group_counter-1)*agents_per_group +agents_per_group)):(next =
0);
          NFAgent *previous_add = (NFAgent *) [pop_index objectAtIndex:previous];
          NFAgent *next_add = (NFAgent *) [pop_index objectAtIndex:next];
          [sub_group addObject: previous_add];
          [sub_group addObject:next_add];

         }

     long agent_index = 0;

    while(agent_index < [sub_group count]) //cycles through agents in the group
    { // BEGIN CYCLING THROUGH GROUP
        NFAgent *agent_a = (NFAgent *)[sub_group objectAtIndex:agent_index];
        double mrs_a = [agent_a getG1] *[agent_a getG2];
        int aG1 = [agent_a getG1];
        double aG2 = [agent_a getG2];
        float _alpha, _beta;
        _alpha = [agent_a get_alpha];
        _beta = [agent_a get_beta];


          /*
             Sort on prices
          */
          NSMutableArray *tracker;
          NSMutableArray *sorted;
          NFAgent *toAdd;
          int i=0;
          int location;
          int total_parms;



          /*
             price sorting
          */
          sorted = [[NSMutableArray alloc] init];
          tracker = [sub_group mutableCopy];
          /*
             must determine here if have cross overs, how many, and add those to the list
```

```
*/
if([agent_a getNumberCross] > 0)
{
   /* have cross overs, lets add them*/
   int crosses = [agent_a getNumberCross]; // assumed to be between 0-100% of total max
   int lc4 = 0;
   NSMutableArray *local_sorted = [[NSMutableArray alloc] init];
   NSMutableArray *local_tracker = [pop_index mutableCopy];
   int max_crosses = totalAgents - agents_per_group;    //formula for maximum crosses
   (end_agents_overlap > 0)?(max_crosses -=2):(max_crosses +=0); //account if there are overlap
   //find range of acceptable numbers

   int lower_bound = ((group_counter-1)*agents_per_group);
   int upper_bound = ((group_counter-1)*agents_per_group) + (agents_per_group-1);
   //(end_agents_overlap > 0)?(lower_bound =previous):(upper_bound =next); //account if there are
overlap
  // int total_c = ( (crosses/255 * max_crosses);
   int total_c =  crosses;

   (total_c > max_crosses)?(total_c = max_crosses):(total_c = total_c);
   while((lc4<total_c)&&(lc4<max_crosses))
   {
      int to_add = ((unsigned int)random() % (totalAgents-lc4)) + 1;
      to_add--;
      if( !((to_add >= lower_bound) && (to_add<=upper_bound)) )
      {

         bool _add = true;
         if(end_agents_overlap >0)
         {
            if(! (to_add != previous) && (to_add !=next) )
            { _add = false; }
         }

         if( _add)
         {
            NFAgent *localAdd = (NFAgent *) [local_tracker objectAtIndex:to_add];
            [local_sorted addObject:localAdd];
            [tracker addObject:localAdd];
            [local_tracker removeObjectAtIndex:to_add];
         }


      }
      lc4++;
   }

}

int total_in_tracker = [tracker count];

for(i = 0; i <total_in_tracker; i++) //sorts prices
{
   location = [self getMaxPrice:tracker andG1: aG1 andG2: aG2 andAlpha: _alpha andBeta: _beta];
   toAdd = (NFAgent *) [tracker objectAtIndex:location];
```

```
    [sorted addObject:toAdd];
    [tracker removeObjectAtIndex:location];

}
//print out max prices
for(int ii = 0; ii<[sorted count]; ii++)
{ // START SORTING THROUGH PRICES
    TestAgent *toList = [sorted objectAtIndex:ii];
    int bG1 = [toList getG1];
    double bG2 = [toList getG2];
    double price = (bG2 + aG2)/(bG1 + aG1);
    unsigned long agent1ID = [agent_a getAgentID];
    unsigned long agent2ID = [toList getAgentID];
    if(agent1ID != agent2ID)
    { //BEGIN IF ONLY GO HERE IF AGENT IS NOT THE SAME ONE

        //going to trade price unites of good 2 for 1 unit of good1
        double to_decrease = price;
        int to_increase = 1;

        bool continue_trading = true;

        number_of_searches_per_round ++;
        [agent_a incrementSearches];
        [toList incrementSearches];
        while(continue_trading)  // continue trading as long as it is beneficial
        {
            double utility1,utility2,new_utility1,new_utility2;
            bG1 = [toList getG1];
            bG2 = [toList getG2];
            aG1 = [agent_a getG1];
            aG2 = [agent_a getG2];

            utility1 = [agent_a get_utility];
            utility2 = [toList get_utility];
            float alpha1,alpha2,beta1,beta2;

            alpha1 = [agent_a get_alpha];
            alpha2 = [toList get_alpha];
            beta1 =[agent_a get_beta];
            beta2 = [toList get_beta];

            new_utility1 = pow((aG1+to_increase),alpha1)*pow((aG2 - to_decrease),beta1);
            new_utility2 = pow((bG1-to_increase),alpha2)*pow((bG2 + to_decrease),beta2);

            if((new_utility1 > utility1) && (new_utility2 > utility2))
            {
                [agent_a addG1:to_increase];
                [agent_a addG2:(-1*to_decrease)];

                [toList addG1:(-1*to_increase)];
                [toList addG2:(to_decrease)];



                bG1 = [toList getG1];
```

```
                    bG2 = [toList getG2];
                    aG1 = [agent_a getG1];
                    aG2 = [agent_a getG2];


                    number_of_trades_per_round++;
                  }
                  else {continue_trading = false;}
                }
                //////////
              }/// END IF
            } // DONE SORTING THROUGH PRICES
          agent_index++;
        } // DONE CYCLING THROUGH GROUP
          group_counter++;
        }
        number_of_trades_total +=number_of_trades_per_round;
        number_of_searches_total +=number_of_searches_per_round;

        to_write = fopen(outputfile,"a");

        if(to_write)
        {
          //fprintf(to_write,"Trades in this round: %lu   Number of Searches: %lu
\n",number_of_trades_per_round,number_of_searches_per_round);
          fclose(to_write);
        }
    } // end method


    _endTick_open_

        double *price_vector = (double *)malloc(sizeof(double) * totalAgents);

        if(turn_tracker ==1)
        {
          to_write = fopen(outputfile,"a");
            fprintf(to_write,"Turn \t Avg Price \t Std Dev \t trades \t searches \n");
            fclose(to_write);

        }

        _allAgents_open_
          if(turn_tracker == 1)
          {
            number_of_trades_per_round = 0;

            number_of_trades_total = 0;
            number_of_searches_per_round = 0;
            number_of_searches_total = 0;



            _send_message_(allAgents,_message_agent_init_);

            if(cross_overs>0)
```

```
        {

                int _crossPerc = ((unsigned int)random() % 100) + 1;
                if( (cross_overs >= (100 - _crossPerc )) )
                {
                   [allAgents setCross:true];
                }
        }
                   }

        // printf("\n\nReport for Agent Number: %d",i);

        double _g2 = [allAgents getG2];
        double _g1 = [allAgents getG1];
        double _alpha = [allAgents get_alpha];
        double _beta = [allAgents get_beta];
        double _price = (_alpha/_beta) * ( _g2/_g1);

        if(isnan(_price))
        {
           ;
        }

        price_vector[i] = _price;


    _allAgents_close_

    // get vector of prices
    // calculate average prices, std dev
    int N = totalAgents;
    double sum = 0;
    int _i = 0;

    while(_i < N)
    {
       sum += price_vector[_i];
       _i++;
    }
    double average = sum/N;
    _i = 0;
    sum = 0;

    while(_i < N)
    {
       sum += (price_vector[_i] - average)*(price_vector[_i]-average);
       _i++;
    }
    double variance = sum/N;
    double standard_deviation = sqrt(variance);

    [self trade];
    to_write = fopen(outputfile,"a");

    fprintf(to_write," %i \t %f \t %f\t \t %lu \t %lu
\n",turn_tracker,average,standard_deviation,number_of_trades_per_round,number_of_searches_per_round);
```

```
        fclose(to_write);

      /*

      DO MATCHING, TRADING, ETC

      */


      _default_endTick_

   _endTick_close_

_close_implementation_(TestPopulation)

////////////////////////////////////////////////// SIMULATION

_open_describe_simulation_(TestSimulation)
   _open_describe_traits_

   int num_groups;
   int agents_in_each_group;
   int cross_overs;
   int end_agents_overlap;

   _close_describe_traits_
_close_describe_simulation_(TestSimulation)


_open_implementation_(TestSimulation)


// init

   _init_sim_open_
      int i = 0;
      int turns_to_learn = 3; //how many ticks until learning is initiated
      int use_genetic = 1;
      int protect = 5;
      int mutation_rate = 2;

      // _get_pop_

    /*  _get_agents_in_group_
      _get_groups_
      _get_cross_overs_
      _get_overlap_*/

       num_groups = 500;
       agents_in_each_group = 1;
       end_agents_overlap = 0;
       cross_overs = 100;
      counter1_reserved = num_groups * agents_in_each_group;
      _attach_agent_type_(TestAgent,TestPopulation,turns_to_learn,
      use_genetic,protect,mutation_rate,i)
```

```
        numPopulation = 1;

        NFPopulation *toStart = (NFPopulation *)[pop_index objectAtIndex:0];


        to_write = fopen(outputfile,"a");
        if(to_write)
        {
            fprintf(to_write,"Genetic is: %d \t Protecting: %d \t mutation rate: %d
\n",use_genetic,protect,mutation_rate);
            fprintf(to_write,"groups = %d  in each = %d cross overs = %d overlap = %d
\n",num_groups,agents_in_each_group,cross_overs,end_agents_overlap);
            fclose(to_write);
        }

        [toStart setParms:num_groups andNumInEach: agents_in_each_group andCross: cross_overs andOverlap:
end_agents_overlap];

        duration = 1000;
    _init_sim_close_



    //where the simulation is run, called at end of each tick
    _endTick_open_
        _default_endTick_ //default is to call each populations end tick, which calls its agents end tick
    _endTick_close_



_close_implementation_(TestSimulation)

//////////////////



_simulation_run_(TestSimulation)
```