COMMUNICATING SEQUENTIAL AGENTS:
AN ANALYSIS OF CONCURRENT AGENT SCHEDULING

by

Stefan D. McCabe
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Master of Arts
Interdisciplinary Studies

Committee:

_____ Director

_____

_____

_____ Program Director

_____ Dean, College of
Humanities and Social Sciences

Date: _____ Spring Semester 2016
George Mason University
Fairfax, VA

Communicating Sequential Agents: An Analysis of Concurrent Agent Scheduling

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Arts at George Mason University

By

Stefan D. McCabe
Bachelor of Arts
George Mason University, 2013

Director: Robert L. Axtell, Professor
Department of Computational and Data Sciences

Spring Semester 2016
George Mason University
Fairfax, VA

Dedication

For Mom—I couldn't have done it without your support every step of the way.

Acknowledgments

First, I must thank my committee for their massive contributions to this project:

Dr. Robert Axtell, who mentored me throughout this process and supported my research in this area.

Dr. Kenneth Comer, whose dissertation provided a foundation for my own work, and who encouraged me along the way.

Dr. Andrew Crooks, without whom I would never have finished writing.

I owe a great deal of thanks to Matthew Oldham, who encouraged me throughout the thesis-writing process and who served as a sounding board for many of my ideas.

I also would like to thank my friends and family for putting up with my endless worrying and complaining.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABM	Agent-based model
CA	Cellular automata
CoMSES	Computational Modeling for SocioEcological Science
CPU	Central processing unit
CSP	Communicating Sequential Processes
CUDA	Compute Unified Device Architecture
DM	Deviation from the mode
D-MASON	Distributed MASON
FLAME	FLexible Agent Modeling Environment
GPU	Graphics processing unit
HPC	High-performance computing
IBM	Individual-based model
MASON	Multi-Agent Simulator of Neighborhoods or Networks
MPI	Message Passing Interface
MRS	Marginal rate of substitution
ODD	Overview, Design Concepts, Details
PPHPC	Predator-Prey for High-Performance Computing
PRNG	Pseudorandom number generator
RNG	Random number generator
TRNG	True random number generator
UI	User interface

Abstract

COMMUNICATING SEQUENTIAL AGENTS: AN ANALYSIS OF CONCURRENT AGENT SCHEDULING

Stefan D. McCabe

George Mason University, 2016

Thesis Director: Robert L. Axtell

Concurrent scheduling of agents presents a challenge for researchers who wish to develop scalable agent-based models (ABMs) without sacrificing intelligibility or fine control over model elements. In this thesis, I advance our understanding of the requirements and challenges of concurrent scheduling by investigating the problem outside of existing ABM modeling frameworks. I examine the possibility space of agent activation regimes, considering as axes: parallelization, selection order, updating regime, endogenous or exogenous access to model state, uniformity of activation, and reproducibility.

This analysis informs a systematic review of ABMs on a popular repository of ABM source code to determine how researchers are currently addressing agent activation issues. The review suggests that there is currently widespread homogeneity of modeling practices regarding agent activation.

I also expand an existing ABM of economic exchange to demonstrate the effects of varying activation regime on model results and model runtime, extending the analysis to a parallel computing context. This work also extends previous work on agent activation by applying the examination on a more complex model. Varying the activation regime

produces significant differences in behavior and model outcomes in this more complex model.

This research contributes to the existing literature on the implementation of agent-based models and may be of use for further advances in ABM library development. The results of the case study may also be of interest to researchers of the foundations of economic theory.

Chapter 1: Introduction

1.1 Overview

Agent-based models (ABMs) allow researchers to simulate interactions between a population of agents. These agents will have novel behaviors and properties but the precise pattern of their interactions, the activation regime, can be of interest in its own right (Comer, 2014). Unfortunately, these interactions—the source of the ABM's novelty—are also a hindrance to parallelization, a common way to improve the runtime of a computer program. Until recently, the there has been a trade-off between fine control of the activation regime and model parallelization. This trade-off has remained underexamined. For example, are some activation regimes intrinsically better-suited to parallelization than others? The literature is silent.

The central question animating this work is therefore: do advances in our understanding of the activation regime in serial (non-parallel) ABMs also hold for parallel ABMs? This opens up a number of related sub-questions, each of which will be handled in a different chapter.

In Chapter 2, I examine what is known about ABM activation and updating and present a typology of activation regimes. The typology raises the question of relative suitability of different activation regimes in a parallel computing context. Finally, I examine the question: does there exist a class of activation regimes that are specifically *parallel regimes*? The single-threaded focus in the agent activation literature means that these have likely not been identified or explored yet.

This typology also touches on issues of reproducibility. How desirable is it that an ABM be reproducible from a random seed, given that some methods of parallelizing ABMs are explicitly nondeterministic? Is this a grave problem for those ABMs or is it sufficient to demonstrate that the range of outcomes follow some probability distribution? Compared to traditional social science methodologies, the requirements to fall under the aegis of "reproducible research" may be significantly different.

In Chapter 3, I conduct a systematic review of a set of publicly-available ABMs and their source code in order to better understand how modelers have attended to agent activation. How many modelers make explicit their assumptions in this area, and how often is activation regime implemented in novel fashion? Does the default behavior of modeling software constrain the number of activation regimes used? The data suggest that, although modeling software does not constrain modeler decision-making, there is nevertheless widespread homogeneity with respect to agent activation and parallelization.

Chapter 4 connects activation issues and parallelization at the implementation level by extending the bilateral exchange model of Axtell (2005) to include a range of activation regimes in serial and in parallel. Model runtimes and quantitative outcomes are both analyzed. The results support the notion that activation regime continues to play an important role in the behavior of increasingly complex agent-based models while also providing reason to believe that performance gains are likely to be modest and are affected by the choice of activation regime.

This research should be of interest to much of the agent-based modeling community. Though many models are prototyped in NetLogo or Python, once a model becomes sufficiently large it is more efficient to re-implement that model in a language with faster performance. This research contributes to that effort. It also draws on extant literature in the computer science and software engineering disciplines and may have implications there. From a practical standpoint, it may serve as the basis for the development of an

agent-based modeling library or framework or a set of revisions to an existing library or framework.

This research contributes to a severely neglected strand of the agent-based modeling literature dealing with agent scheduling. It has been well-established (Axtell, 2000a; Comer, 2014) that agent activation regimes have a significant effect on model behavior and outcomes, but it is relatively rare for modelers to make their assumptions here explicit. The earlier work of Axtell and Comer operate in a purely single-threaded context, where I address agent activation regimes potentially unique to concurrently-structured models.

1.2 Literature Review

This review will begin with a discussion of agent-based models as a methodology: their origin, motivation, and areas of use. I will then introduce concerns about the computational performance of ABMs, the ways in which previous researchers have attempt to speed up their models, and how related disciplines (like scientific computing) have attempted to address the same issue. Research on agent scheduling, a relatively small and neglected branch of the ABM literature, will then be introduced and it will be shown that these two branches of research—scheduling and performance—have rarely overlapped. I will also address how similar computational modeling paradigms, like cellular automata (CA), have (or have not) addressed scheduling issues.

1.2.1 Agent-Based Modeling: An Overview

Complex Systems

Computational social science is the study of social systems using an approach informed by *complexity theory*. Complexity can present an obstacle to studying a system because of the challenge it presents to traditional statistical and mathematical tools, but also because

"complexity" is a difficult and elusive term. Herbert Simon (1996) defines a *complex system* as a system "made up of a large number of parts that have many interactions" (pp. 183-184). A complex system has two defining features: *hierarchy* and *near-decomposability*. Hierarchy refers to the fact that complex systems consist of interacting subsystems; near-decomposability implies that these subsystems can (at least in the short-term) be analyzed as systems in their own right. Complexity theory builds on prior work in chaos theory and catastrophe theory, implying that many complex systems may have nonlinear dynamics and multiple equilibria.

There are a number of texts addressing the relationship between complex systems and social science (Cioffi-Revilla, 2014; Gilbert & Troitzsch, 2005; Miller & Page, 2007; Mitchell, 2011). Of particular interest to computational social scientists is the concept of *emergence*: when a "phenomenon... requires new categories to describe it which are not required to describe the behavior of the underlying components" (Gilbert & Troitzsch, 2005, p. 11). In other words, relatively simplistic behavior at a micro level can produce complex patterns at the macro level. Emergence can be difficult to identify using statistics or mathematical equations because the level of analysis for those methods is typically at the population level, not the individual level.

Agent-Based Models

For many classes of research questions, complexity implies that a given system cannot be productively analyzed using mathematical equations. Axtell (2000b) presents the justification for using agent-based models in these scenarios. An *agent based-model* is simply a collection of computational agents, with "states and rules of behavior" (p. 2), that interact for a period of time. When instantiating a population of agents, it is trivial to vary agent attributes along some distribution, allowing for agent heterogeneity and thereby allowing researchers to move away from "ideal types" (p. 1) like the representative agent

		Agent	
		Designed	Analysed
Environment	Designed	Model description – Abstract Purpose/intent – Discovery of new relationships – Existence proof Verification and validation strategy – Theoretical comparison – Replication Appropriate development tools – Easy to implement simulation/ modelling system Example model – Filatova et al. (2009)	Model description – Experimental Purpose/intent – Role-playing games among stakeholders – Laboratory experiments Verification and validation strategy – Repetitions – Adequacy of design Appropriate development tools – Flexible simulation/modelling systems with well developed user interfaces Example model – Mooij et al. (2002)
	Analysed	Model description – Historical Purpose/intent – Explanation Verification and validation strategy – Qualitative: goodness of fit Appropriate development tools – Advanced simulation/modelling systems linked with GIS Example model – Mathevet et al. (2003)	Model description – Empirical Purpose/intent – Explanation – Projection – Scenario analysis Verification and validation strategy – Quantitative: goodness of fit Appropriate development tools – Low-level programming languages Example model – Jackson et al. (2008)

Figure 1.1: A typology of agent-based models, from Crooks and Castle (2012, p. 228).

of economics. Furthermore, agent-based models scale easily: increasing the agent population may require more computational power or more memory, but it does not require the researcher to write more lines of code.

Agent-based models have proven useful in a wide variety of disciplines, including ecology (Bennett & Tang, 2006; Mooij, Bennetts, Kitchens, & DeAngelis, 2002), epidemiology (Crooks & Hailegiorgis, 2014), economics (Geanakoplos et al., 2012; Gode & Sunder, 1993) and political science (Laver & Sergenti, 2012). They also provide value across the applied-theoretical divide: the most simple agent-based models are essentially Monte Carlo simulations (Axtell, 2000b, pp. 6–7) whereas more complex models are extensively calibrated and validated against empirical data (Crooks & Castle, 2012). Figure 1.1 provides a representative typology of (spatial) ABMs and their various use cases.

Software

As shown in Figure 1.1, agent-based models can be implemented in a variety of ways. Historically they have been written in low-level programming languages but in recent years frameworks like Repast (North, Collier, & Vos, 2006) and MASON (Luke, Cioffi-Revilla, Panait, Sullivan, & Balan, 2005) have attempted to make the development process easier and more consistent.

NetLogo (Wilensky, 2015) is an increasingly popular integrated modeling environment, written in Java. It provides an intuitive interface and simplified domain-specific language for agent-based models. Different agent types can be declared using `breeds`; all agents of a given type can be queried using the `ask` primitive: `ask agents [ DoSomething ]`. NetLogo is particularly well-suited to developing spatial ABMs; where representing and visualizing space can be time-consuming in lower-level languages, it is built in to NetLogo.

1.2.2 Agent-Based Modeling and High-Performance Computing

Motivation

Axtell (2000b, p. 6) noted that, due to the ability to vary the size and behavior of the agent population, "a relatively short 'program' at compile-time is actually a very large 'program' at runtime." The ability to scale models easily is one of the great advantages of ABMs, but it comes at a steep cost in performance. Agent-based models consist of dense interactions between agents (and, potentially, their environment).

Jaffry and Treur (2011) examined the performance of a relatively simple agent-based model of trust dynamics against a population-based model that produced similar results. They found that the time complexities of the agent-based and the population-based models were $O(N^2\tau)$ and $O(\tau)$ respectively, where N is the size of the agent population and τ is the number of time steps analyzed. In other words, run time scaled linearly with τ for

the population-based model but ran in polynomial time for the agent-based model.[1] This result suggests that ABMs often scale poorly as the agent population increases, forcing researchers to investigate methods to improve performance.

Axtell (2005) discusses the theoretical relevance of time complexity for economics. The prevailing Walrasian model of exchange, as formalized by Arrow and Debreu, postulates a hypothetical "auctioneer" that computes the prices for a market of agents exchanging heterogeneous goods. The auctioneer cannot efficiently calculate these prices; the time complexity is non-polynomial (p. F193). An alternative theoretical model, in which distributed agents independently and asynchronously trade goods at local prices, has a much better time complexity and a computational complexity of this process is P. The model presented by Axtell will be extended in Chapter 4.

The simplest way of speeding up model performance is often simply to write the agent-based model in a different language. Many agent-based models are prototyped in NetLogo (a Java-based integrated modeling environment) or in Python. These high-level languages allow for rapid development but often perform more slowly than lower-level languages like C or C++. NetLogo in particular struggles with large numbers of agents (Comer, 2014, p. 6). Figure 1.1 acknowledges this tendency; as model complexity increases (that is, moves from the top left of the typology to the bottom right), the recommended development environment changes from high-level languages or toolkits to low-level languages.[2]

Parallel Computing

Significant performance gains can often be made simply by writing in a lower-level programming language, but that can be insufficient. Simple C or C++ code still runs *serially*,

[1] Note that usually one cannot simply reduce the size of the agent population to improve model runtime—in some models (e.g., Crooks & Hailegiorgis, 2014) it can result in different outcomes.

[2] ReLogo, a variant of RePast that uses many of the design concepts of NetLogo, may ease the transition from high-level to low-level, but Lytinen and Railsback (2012) report extremely poor performance results.

that is, along a single thread of execution. A more fruitful approach can be found in *parallelism*—"harnessing multiple processors to work on a single task" (Herlihy & Shavit, 2012, p. 1)—which improves scalability, subject to Amdahl's law.[3] Achieving parallelism is sometimes easy (so-called *embarrassingly parallel* tasks) but typically requires more attention to the structure of the program.

The traditional paradigm of parallel programming has been multithreaded programming, implemented in languages like C, C++, and Java. In this paradigm the key abstraction is the idea of *threads*, "sequential processes that share memory" (Lee, 2006, p. 33). Each processor (or logical processor) can execute a thread. Therefore, on an octal-core system there can theoretically be 8 threads executing in parallel (16 if the processor is hyperthreaded). If memory must be shared between threads, extreme care must be taken to ensure *mutual exclusion*—that is, to make sure that two threads do not attempt to access or modify the same piece of information at the same time. For example, in a parallel ABM two execution threads should not try to move the same agent simultaneously. Although sometimes difficult to work with, threads are immensely powerful tools.

Lee (2006) is highly critical of the thread-based paradigm, however. The thread is too low-level an abstraction; programmers are too easily bogged down in the minutiae of memory management. More problematically, threads introduce *nondeterminism*, the possibility that the same program can return different results. Lee thinks this is a serious problem:

> Threads, on the other hand, are wildly nondeterministic. The job of the programmer is to prune away that nondeterminism. We have, of course, developed tools to assist in the pruning. Semaphores, monitors, and more modern overlays on threads... offer the programmer ever more effective prunings. But

[3] Amdahl's law simply notes that the proportion of a model that cannot be parallelized significantly affects the speed-up that can be obtained through parallelization. (Herlihy & Shavit, 2012)

pruning a wild mass of brambles rarely yields a satisfactory hedge. (p. 35)

While controversial and possibly overstated, Lee's critique of threads exposes some real weaknesses of the thread-based paradigm. Lee (2009) presents some of his own proposals, but they have not been widely accepted. In "The Problem With Threads," however, he suggests that message-passing processes (see below) may be a helpful approach.

Hoare (1978) anticipated many of the problems that would appear in the thread-based paradigm. He proposed his own language, CSP (for Communicating Sequential Processes), that treated process inputs and outputs as basic primitives of the language. Hoare's proposed language incorporated seven central ideas, most importantly the guarded command[4] of Dijkstra (1975), for communication between processes. A command within a process can specify that it is receiving input from another (named) process and will simply block until it receives such input; the same process is followed for outputs. For interested readers, Hoare (p. 674) includes a parallel implementation of the Sieve of Eratosthenes as an example of how these concepts can be used in a scientific computing context.

Hoare (1985) later formalized his CSP language further, but the insights of the initial proposal are more important than the semantics of the language itself. Languages that implement CSP or draw heavily on Hoare's insights include Occam (the first language to use this model), Limbo, Plan 9, Clojure, and Go. Libraries emulating CSP have been written for other languages, including C++, Haskell, and Scala. CSP has not yet been used in an ABM context, and the languages that prominently implement CSP have not traditionally been used for ABMs.

Two frameworks for parallel or distributed ABMs have recently appeared. Cordasco et al. (2013) present D-MASON (Distributed MASON) is a framework-level approach to writing distributed ABMs. Using a "master/worker paradigm" (p. 1236), they distribute

[4] A guarded command is a command of the form $G \rightarrow X$, where G is the guard. If G is true, X is executed, otherwise something else happens.

the execution of spatial ABMs using *field partitioning*. In this technique, the global model state is synchronously updated from multiple spatial subcomponents. As of publication, the authors reported some limitations on agent movement as a result of this technique, summarized as a rule against "teleportation" (p. 1241), and it is not clear how field partitioning applies to aspatial models. However, field partitioning does allow for fully reproducible parallel models by placing a pseudo-random number generator (see below) on each field.

Another extension of a currently-existing ABM library is Repast-HPC, a C++ library for distributed ABMs (Collier & North, 2013; North, Collier, & Murphy, 2016). Designed to support parallel simulation of "either relatively few heavy, computationally complex agents per process or a multitude of lighter-weight agents per process" (Collier & North, 2013, p. 1221), Repast-HPC uses the discrete-event scheduler of Repast Simphony, with agents scattered across different threads. Synchronization across threads is managed using the Message Passing Interface (MPI) framework for distributed computing. The generation of random numbers is handled by a single, central random number generator in order to guarantee reproducibility.

GPU Programming

A second approach to writing high-performance ABMs involves programming on graphics processing units (GPUs). GPUs were originally developed to facilitate graphics-intensive tasks like computer gaming and video editing but have also proven quite useful for scientific computing. A GPU consists of a number of *shaders*, lightweight CPUs designed for efficient calculation of 3D graphics. Because there are a number of shaders and these shaders are extremely fast, GPUs are well-suited for many scientific computing tasks, e.g. the solving of very complicated differential equations (Kindratenko, 2014). Perhaps the best-known example of how GPUs can be leveraged for scientific computing is the Folding@Home project (Beberg, Ensign, Jayachandran, Khaliq, & Pande, 2009), which uses

distributed GPU programming to simulate protein folding for biomedical research.

GPU programming was first used for ABMs by D'Souza, Lysenko, and Rahmani (2007), who implemented the well-known Sugarscape model of Epstein and Axtell (1996). They successfully ran the model with over two million interacting agents on a large grid while still calculating over 50 model turns per second. Their model demonstrates well the performance gains that can be achieved on relatively cheap hardware by using the GPU, but also some of the pitfalls: because shaders are designed to perform calculating on relatively light objects, the memory structure used on the GPU is very different from a traditional computing context. In particular, agents are represented as graphical textures and their state is changed by modifying a texture's color.

Their work predated the development of more general-purpose GPU programming libraries like Nvidia's Compute Unified Device Architecture (CUDA), so these problems have been mitigated somewhat. For most mathematical analysis tasks, CUDA is quite effective at speeding up these calculations (Kindratenko, 2014), but the memory requirements of agent-based models still impose some difficulties. Richmond, Coakley, and Romano (2009) use CUDA to develop a framework for writing ABMs for the GPU, based on the FLexible Agent Modeling Environment (FLAME) (Coakley, Smallwood, & Holcombe, 2006). Their framework, FLAME-GPU, mitigates this problem somewhat. Note, however, that their approach provides explicitly nondeterministic models (Rousset, Herrmann, Lang, & Philippe, 2014).

1.2.3 Activation and Updating in Agent-Based Models

Overview

Agent-based models are essentially the aggregation of a series of interactions between computational agents. The manner and order in which agents interact is therefore of paramount

importance. Axtell (2000a) discusses the effect agent *activation regime* can have on the behavior of an ABM. He identifies two activation regimes—uniform and parallel. Under uniform activation, all agents activate once per turn. In the most naïve implementation, each agent activates in order (agent 1, then agent 2, and so forth). This can bias the model, so more commonly the agents are randomly shuffled before the turn. A random activation regime simply activates agents at random; agents are not guaranteed to activate once per turn. He finds, using a simple firm dynamics model that the choice of activation regime can bias the model significantly. It is well-known that the variance in firm growth rates decreases as firm size increases. Under a random activation regime, the outputs of his model closely match the distribution of firm growth rates in the United States—the variances follows a power law with negative exponent. With uniform activation, his model produces demonstrably incorrect results—the variances of firm growth rates follow a power law with *positive* exponent.

Earlier Work: Contributions from Cellular Automata

Axtell (2000a) did not have to develop this idea in a vacuum, for a similar literature on updating and activation exists in the field of cellular automata (CA). Nowak and May (1992) developed a CA representation of the prisoner's dilemma in a spatial context. They found, contrary to existing theory, outcomes in which some agents persisted in cooperating over long periods of time. This finding was criticized by Huberman and Glance (1993), who demonstrated that this finding was a product of *synchronous updating*.[5] That is, all agents at time t made their decisions using information from time $t-1$, and then the state changed

[5]Note that updating and activation are closely related but slightly different: activation refers to the order in which agents interact and updating refers to the information they have when the interact. Using just the examples of Axtell (2000a) and Huberman and Glance (1993), one can construct a matrix of potential designs: asynchronous-random, asynchronous-uniform, synchronous-random, synchronous-uniform. This point will be elaborated upon later.

globally. With *asynchronous updating*,[6] in which agents used the most up-to-date information about their state when making a decision, agents overwhelmingly defected, as predicted by game theory (but see Nowak, Bonhoeffer, & May, 1994, for a rejoinder).

Schönfisch and de Roos (1999) present a rigorous explication of CA updating schemes, including various asynchronous regimes. They distinguish asynchronous event-driven and asynchronous time-driven regimes, the latter incorporating activation via a "Poisson clock" calculating arrival times for different cells (for more on Poisson activation, see below). They note that the Poisson clock is likely the most "satisfying" (p. 193) regime for many CAs. Time-driven updating avoids the arbitrariness that comes from assuming that events happen with some regularity or some even spacing, as event-driven updating does. They also caution that many asynchronous activation regimes "usually introduce a lot of additional, unintended structure into the dynamics and patterns of the cellular automaton" (p. 139).

Baetens, Van der Weeën, and De Baets (2012) summarize a number of reports of qualitative changes in model behavior observed by switching between synchronous and asynchronous updating. They also quantitatively measure the effect of varying both activation order and updating scheme on the stability of their CA using Lyapunov exponents. They find that moving from synchronous to asynchronous updating has the largest effect on model stability, with varying between different asynchronous activation orders producing significant but more modest effects.

Radax and Rengs (2010) connect the CA updating literature to ABMs. While stressing that "there is no definite best choice" (p. 1), they note that asynchronous updating often has higher external validity. They also warn that within asynchronous updating, there may

[6]The definition used by Caron-Lormier, Humphry, Bohan, Hawes, and Thorbek (2008, p. 523) may be verbose but helpful: "asynchronous updating refers to a method by which the objects' characteristics are updated and made available for the other objects as the simulations run through the interactions within a time step."

be significant effects from the choice of agent selection order. Based on this, they caution that integrated modeling environments (e.g., NetLogo) that obscure the activation regime from users should face particular scrutiny.

Agent Selection

While ABM development is significantly affected by choice of activation regime, there is still little mention of it as a *choice* made by modelers. Although Railsback and Grimm (2012)—a standard textbook—includes a few pages on execution order, a cursory examination of the ABM literature shows that the decision is rarely considered (or at least, rarely made explicit when specifying the model) and a review of errors and artifacts in ABMs (Galán et al., 2009) mentions scheduling only briefly. Part of the reason for this neglect is that modelers are often nudged into a default activation regime by their choice of software. In particular, the popular ABM framework NetLogo (Wilensky, 2015) implements uniform activation through the `ask` command, obscuring the choice of activation regime from the modeler (Comer, 2014; Radax & Rengs, 2010). Frameworks with flexible discrete-event schedulers (e.g., MASON and RePast) offer more flexibility, limited by the modelers' creativity.

The difficulties posed by poor documentation of activation regime can impede the replication process. For example, Axtell, Axelrod, Epstein, and Cohen (1996) attempted to dock two relatively similar ABMs: Axelrod's cultural transmission model and Epstein and Axtell's Sugarscape model. Because the two models used different activation regimes, they produced subtly different outcomes. In a relatively large-scale version of the docking experiment, Sugarscape's uniform activation regime produced fewer stable regions than the Axelrod model, which used random, non-uniform activation. This difference was statistically significant. In this particular case the differences were minor but notable; in other cases it can be more severe.

Page (1997) is among the first papers to examine the importance of activation and updating for ABMs. Examining two CAs[7]—the game of life and the conformity game—Page finds only modest effects of synchronous versus random asynchronous updating on these CAs. However, he introduces a third activation regime ("incentive-based updating", where an agent's chance to activate is governed by a utility function) which increases the sensitivity of the model to initial conditions, i.e., it strengthens first-mover advantage.

The dissertation of Comer (2014) examines the effect of activation regime in a more systematic fashion. Comer argues that "policy-centric" (p. 12) models (those tending toward the bottom right of the typology represented in Figure 1.1) have generally neglected addressing issues with activation regime, leaving the choice unspecified. He implements a number of models and demonstrates how different activation regimes can alter their behavior. Importantly, he imports two notable concepts from the CA and econophysics literature: the *Poisson activation regime*, in which time is modeled in a more continuous fashion than random or uniform activation; and *endogenous activation regimes*, in which the state changes of the model can effect the agents' rates of activation. For example, in an endogenous Poisson activation regime, agent activation is governed by a heterogeneous parameter λ. As the model runs, agents can change their λ and therefore the rate at which they activate relative to other agents.

Comer (2014) applies these insights to a set of ABMs: the spatial prisoner's dilemma (Nowak & May, 1992), the civil violence model of Epstein (2002), the zero-intelligence traders model (Gode & Sunder, 1993) and a wealth distribution model based on a prior model of interacting particles in physics (Aldous, 2013). In all cases, he finds that changing activation regimes can affect model behavior. For example, in the last case—what Comer calls the "Leveller" model—the activation regime affects gradient of change in the standard

[7] Although the paper is directed to the agent-based modeling literature, the models examined are actually cellular automata. Presumably at this time the boundaries between these two approaches were less clear.

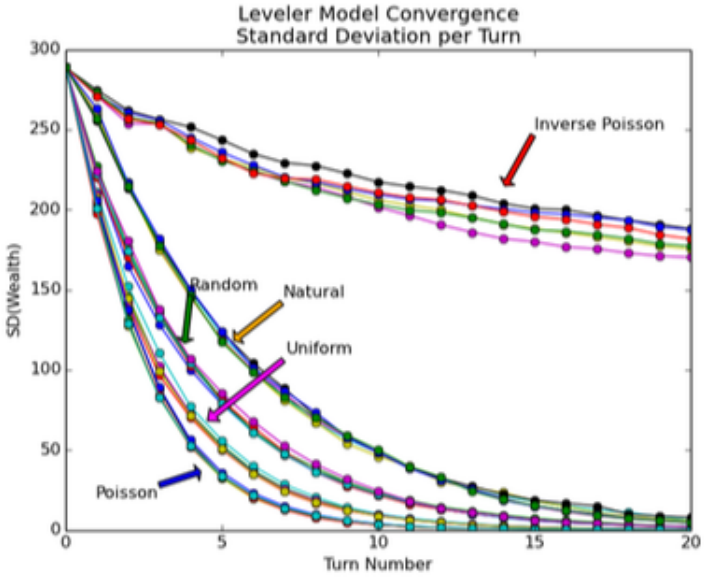

Figure 1.2: Dynamics of standard deviation of wealth in a Leveller model with different activation regimes. From Comer (2014, p. 136).

deviation of wealth in an agent population, as shown in Figure 1.2. Comer's models were developed in NetLogo and Python and all were single-threaded.

Alizadeh and Cioffi-Revilla (2015) examine the effect of similar activation regimes on an agent-based model of opinion dynamics. Examining two forms of Poisson activation regimes in addition to uniform and random activation, they find that the choice of activation regime has significant effect upon the emergence of opinion extremists. In the first Poisson activation regime, more extreme agents activate more frequently, while in the second moderates activate more frequently. Perhaps counterintuitively, the former produces fewer opinion extremists than the latter, a result that supports increased engagement with individuals harboring extreme opinions.

Since Page (1997) introduced updating into the ABM literature, it has received comparatively little attention. Caron-Lormier et al. (2008) also examine updating in a "deliberately

simple" individual-based model[8] (IBM) of proteins with multiple trophic levels. They find that synchronous updating introduces a burn-in process resembling a periodic function and slightly lower overall energy levels for the proteins. They tie this result to an extant literature on the role of delay in ecosystems; more important here is their discussion of the difficulties of implementing synchronously-updating IBMs, where numerous dummy variables have to be attended to in order to create this artificial delay.

Fatès and Chevrier (2010) examine the effects of updating schemes on a simple IBM, the multiple Langton's ants model. Using this very simple model (scarcely more complex than a CA), they introduce the difficulties of resolving collisions in synchronously updated IBMs. With certain collision rules, they produce deadlock behavior previously unseen in earlier analysis of the multiple Langton's ants model. They note that many "artefacts" (p. 540) of updating regimes may disappear in more complex models, meaning investigation into more complex models is needed.

Related Issues

A related concern is the precise method in which agents activate within a given regime. A random activation regime means that agents are activated randomly, but it says nothing about how the random order is determined. The typical approach is to use a pseudorandom number generator (PRNG) (Katzgraber, 2010). PRNGs are a class of algorithms that return a series of numbers by repeatedly applying a transformation to a *seed* number. PRNGs are not truly random; they eventually repeat themselves once the seed value is reached again. This typically takes many iterations; one commonly used PRNG, the Mersenne Twister, generates $2^{19937} - 1$ unique numbers before repeating itself. However, PRNGs can produce numbers that correlate with each other in subtle ways. PRNGs are *deterministic*;

[8] "Individual-based model" is the preferred nomenclature of ecologists; the term is effectively interchangeable with "agent-based model" (Railsback & Grimm, 2012, p. xi).

provided the seed is unchanged running a single-threaded program that uses pseudorandom numbers will always generate the same output. This determinism enables reproducibility; the simulation can be run again with the same seed. If a program needs so many random numbers that the period of a PRNG is exhausted or if there is some concern about autocorrelation, a true random number generator (TRNG) can be used (Katzgraber, 2010, pp. 3–4). TRNGs use some sort of physical process—for example, atmospheric noise—to produce a non-deterministic stream of numbers. Because a TRNG is bound to some physical process, they are typically much slower than PRNGs and they are not reproducible. Rouly (2015) reports on a scenario in which enough random numbers were required of an ABM that it became advantageous to use a commercially-available TRNG, TrueRNG.[9]

Agent activation issues are typically considered to be a subset of a larger literature on verification and validation of agent-based models. This is an important and difficult area; Crooks, Castle, and Batty (2008, p. 419) summarizes some of the difficulties presented in verifying and validating ABMs. The choice of activation regime may be empirically tied to real-world phenomena, so it is therefore a matter of ABM validation.

Other Reasons for Parallelization

Performance is not the only motivation for parallelization. Many multithreaded programs use separate threads to handle visualization and input/output. These are important tasks for ABMs as well; additionally Auble (2015) presents another possible motivation. He develops the idea of narrative agents that report on interesting features of the model dynamics. One approach he presents is the "journalist" agent that summarizes different sections of the model. This sort of agent would be an ideal candidate for parallelization, since it does not change the model state. All of these applications, however, are outside the scope of this thesis.

[9] `http://ubld.it/products/truerng-hardware-random-number-generator/`

The pursuit of high-performance ABMs, in the form of environments like FLAME-GPU and D-MASON, has succeeded in providing helpful tools for writing very fast ABMs. Unfortunately, the designs of these frameworks has obscured an important decision—the choice of activation order and updating system—from the modeler. A better understanding of activation and updating—in a parallel, high-performance context—is needed in order to equip modelers with the knowledge to choose between these tools and construct more valid models.

Chapter 2: A Typology of Agent Activation Regimes

Activation and updating issues are typically presented in a selective or non-systematic fashion. Activation is often treated differently from updating and both are typically kept siloed away from discussions of reproducibility and parallelization, as if they were unrelated concepts. They are not; all these issues are intimately connected. The goal of this chapter is to explicate something close to a full typology of activation regimes.

The possibility space of agent activation regimes is not well-explored. Existing research (Comer, 2014) on agent activation, by focusing narrowly on the order in which agents are selected, neglects the effect of closely related design decisions, including the model updating process and the source of the random numbers used to generate the selection order. Even where analyses of agent activation acknowledge the choice of updating regime, as in the study of asynchronous activation regimes by Alizadeh and Cioffi-Revilla (2015), the treatment of updating is incomplete. There appear to be legitimate use cases for synchronous agent-based models; as such, researchers of ABM methodology should also examine synchronous updating.

I define an *agent activation regime* broadly, encompassing agent selection order, uniformity of agent selection, model updating, reproducibility, and parallelization. Table 2.1 provides a high-level representation of this typology. Many of these issues have traditionally been treated as concepts separate from agent activation; I propose that they are, if not inseparable, at the very least intimately connected. A *model turn* is defined, following Comer (2014, p. 88), as a number of agent activations equal to the size of the agent population.

Table 2.1: Overview of activation regime typology. In **bold** are the design choices represented by NetLogo's `ask` command.

Category	Description	Potential values
Selection criteria	In what order are agents selected for activation?	**Random**, Fixed, Poisson Clock
Uniformity	Is each agent guaranteed one activation per model turn?	**Yes**/No
Updating regime	Does the model state change immediately or do all agents act on information from the start of a turn?	**Asynchronous**, Synchronous
Reproducibility	Can a given model run be repeated consistently (i.e., through use of a random seed)?	**Yes**/No
Endogeneity	If the selection criteria is based on some agent characteristic, can that characteristic change over the course of a model run?	Yes/**No**
Parallelization	Can the agents be activated in parallel?	Yes/**No**

2.1 Agent Selection

Much of the extant literature on agent activation, including Axtell (2000a) and Comer (2014) focus primarily on the choice of which agent goes when. For clarity, I will refer to this as *agent selection* rather than agent activation. Deciding the agent selection process is, for many modelers, the beginning and ending of deciding their model's activation regime. The overwhelming majority of models feature a *random* selection order. To illustrate, let $A = \{a_1, a_2, \dots\ a_n\}$. In random selection, agents are drawn from a discrete uniform distribution $a_x = A_{U(1,n)}$.

Another possibility is some sort of *fixed* selection order. Agents may simply activate in the order in which they were initialized, following the form $a_1, a_2, \dots\ a_n$. (This is typically poor practice that produces unwanted interaction biases. Another possibility, raised by Railsback and Grimm (2012, pp. 186–187) is to encourage positive feedback by sorting agent selection on some attribute of the agent population. For example, agents could have

some attribute governing speed or initiative; it might be intuitive to set the selection order based on that attribute.

Comer (2014) presents the idea of using a *Poisson clock* to govern agent selection.[1] Agents are given some (presumably heterogeneous) parameter λ; "agents have activation times taken as arrival times from an exponential distribution with the arrival rate parameter, λ" (p. 21). That is, the probability of an agent being selected follows $Poisson(\lambda)$. Poisson activation regimes are more complex to implement, particularly while tracking model turns, but have a number of attractive qualities. Algorithm 1 provides an example of how to implement a Poisson activation scheme. Some phenomena are better represented using a Poisson-distributed variable. For example, transit systems can be studied using queuing theory; arrivals of buses or trains can be modeled using a Poisson process. Poisson clocks also provide a way to introduce positive feedback without the possible bias of a fixed, and therefore deterministic, selection order.

Many models require more than one agent to be activated at a time, e.g., an economic model in which agents trade with each other. Taking the simple example of pairwise activation (two agents activated at a time), two agents are selected using the usual criteria (some check must be performed to prevent the same agent from being selected twice) and then activation occurs. A model turn therefore consists of $\frac{n}{2}$ activations of two agents.

2.2 Uniformity

The idea of *uniformity* is closely related to agent selection. The central question here is: are agents allowed to activate more than once per turn? The overwhelming majority of

[1] Note that Comer is not the originator of the Poisson clock; it has been discussed in earlier work by Axtell (2000a), Newth and Cornforth (2009), and Schönfisch and de Roos (1999). However, Comer provides a better overview and his algorithm is better suited for event-driven scheduling of ABMs.

Algorithm 1 Example of a Poisson activation method. λ is based on parameter w.

```
1:  procedure SETPOISSONDISTRIBUTION
2:      totalLambda ← 0
3:  assignLambdas :
4:      for all Agents do
5:          lambda ← w
6:          totalLambda ← totalLambda + lambda
7:  normalizeLambdas :
8:      for all Agents do
9:          lambda ← lambda * (NumberOfAgents * 1.1/totalLambda)
10:         if lambda = 0 then
11:             lambda ← 0.00001
12: scheduleAgents :
13:     for all Agents do
14:         nextT ← −1 * log(U(0,1)/lambda)
15:         while nextT < 1 do
16:             Schedule.add(Agent)
17:             nextT ← nextT + −1 * log(U(0,1)/lambda)
```

ABMs choose a random selection order; within that group there is less unanimity regarding uniformity.[2] The combination of random and uniform selection is often called *uniform activation*; non-uniform random selection then becomes *random activation*. The distributions described above no longer hold; a uniform activation regime does not follow $U(1,n)$ but rather each agent's probability of activation approaches one as a turn progresses. Uniform activation imposes a measure of "fairness" on the model.[3]

Axtell (2000a) discusses the effect uniformity can have on the behavior of an ABM. He identifies two activation regimes—uniform and random. He finds, using a simple firm dynamics model, that the choice of activation regime can bias the model significantly. It is well-known that the variance in firm growth rates decreases as firm size increases. Under a random activation regime, the outputs of his model closely match the distribution of firm growth rates in the United States—the variances follows a power law with negative

[2] Quantitative evidence for this claim is presented in Chapter 3.

[3] The idea of each agent getting their "fair" share of activations is from Axtell et al. (1996, p. 131).

exponent. With uniform activation, his model produces demonstrably incorrect results—the variances of firm growth rates follow a power law with *positive* exponent.

2.3 Reproducibility

It is not immediately obvious why reproducibility would be a concern when developing a typology of agent activation regimes. However, because the activation and updating regimes play such a large effect on the behavior of the model—they are an important part of what Axtell (2000a) calls the "interaction topology"—they are an important determinant of a model's reproducibility. Reproducibility is especially important here because many activation regimes require the generation of large amounts of random numbers.

ABMs are rarely trivial to replicate. If a model features identical behavior from run to run, then it has missed many of the advantages of being an ABM. Agent heterogeneity entails randomness; the ability of agents to interact suggests that there should be something novel about the manner and order in which they interact.

An ABM's activation regime can be fully reproducible without the model itself being fully reproducible. For example, if the criteria by which agents are selected is fixed based on some attribute of the agents, and that attribute is randomly distributed, then the model's reproducibility is contingent on how the random numbers governing the agent attributes were randomly generated.

There are two main approaches to the generation of random numbers: the use of a truly random number generator (TRNG) or the use of a pseudorandom number generator (PRNG). A TRNG uses some process, such as the monitoring of atmospheric noise, to generate random numbers (Katzgraber, 2010). These numbers are not predictable, though they may be biased in some fashion. Katzgraber (p. 3) notes that many TRNGs apply some mathematical function to the raw numbers to "remove any bias in the process." By contrast,

a PRNG is an algorithm that produces a deterministic stream of random-looking numbers from a seed value. Though not truly random, PRNGs are preferred in many applications, including simulations. Katzgraber (p. 4) notes three advantages of using PRNGs: (1) they are much faster than TRNGs, (2) they are portable and not system-specific, and (3) the values can be reproduced by using the same random seed.

I focus here on the distinction between PRNGs and TRNGs, and therefore the distinction between reproducible and nonreproducible random numbers, but closely related in importance is RNG *quality*. Not all RNGs are equal in quality; many are biased in ways that can vary in significance. A poorly designed TRNG is in many ways the worst thing to use in an ABM, as it is both non-reproducible and biased. The Mersenne Twister (Matsumoto & Nishimura, 1998) is a popular PRNG that is both fast and high-quality. The period of a PRNG—the amount of numbers it can generate before repeating itself—may be particularly important for ABMs, which, like many Monte Carlo simulations, generates very large amounts of random numbers.

2.4 Updating

The *updating regime* is the system under which the model resets its global state in reaction to changes in the model. It is closely tied to the cellular automata literature; because CA cells have fixed rules and do not move, the choice of updating regime and the choice of selection regime are interwoven. There are two broad categories of updating regime: synchronous and asynchronous.[4] Under synchronous updating, state is changed for the model as a whole at some specific time, likely the beginning or ending of a time step. Most CAs work this way: each cell evaluates its neighbors, decides to change state according to

[4]One should be careful of treating (a)synchronicity as a rigid binary; in complex systems differences in subsystem behavior and attributes may create "semi-synchronous" states where information about the model state is rapidly received by some parts of the agent but not others.

some decision rule, and then all cells change state (update) simultaneously. By contrast, in asynchronous updating regimes, one cell would evaluate its neighbors and change state, and then another cell would evaluate its neighbors based on that new model state.

The application of updating regimes to ABMs is not as direct as it is for CAs. For CAs, selection and updating issues are usually considered together under the aegis of "updating," where for ABMs the two are usually folded into "activation." Asynchronous updating seems to be the norm for ABMs.

I have already asserted that activation (in terms of agent selection) and updating are intimiately related. Why? The extant literature on activation and updating have typically treated them separately for reasons of parsimony. This has allowed for deep analyses of activation (Axtell, 2000a; Comer, 2014) and updating (Baetens et al., 2012) but has also reinforced the perception that they are distinct issues. Many (though not all) activation orders depend on updating regime to be coherent, however. For example, under synchronous updating all uniform selection orders are identical, but this is not true of all non-uniform selection orders, as Alizadeh and Cioffi-Revilla (2015) assume. In endogenous activation regimes, the time at which the activation parameter is updated may bias the model. When focusing on simple regimes, this is typically not a problem. The choice of asynchronous versus synchronous updating also has serious implications for parallelization, as seen below.

2.5 Endogeneity

In the event that a non-random selection order is chosen, there is likely some agent attribute governing the selection order. This may be a constant value, like a number assigned at initialization, or it may be a variable. If the agent has a variable attribute that influences the selection order, the agent activation regime is *endogenous* rather than exogenous. (Because

most ABMs use random selection, most ABM activation regimes are exogenous.)

To illustrate the concept, consider the Leveller model of Comer (2014, pp. 117–136). A population of agents is initialized with an unequal distribution of wealth. Two agents are activated and their wealth is distributed evenly. If the activation regime is set to an endogenous, non-uniform Poisson clock regime, with λ equal to agent wealth, the wealth distribution of the population is quickly redistributed (see the results for "Poisson" in Figure 1.2). However, if λ is instead based on the closeness of agent wealth to the mean, wealth is redistributed much more slowly (the "Inverse Poisson" results in Figure 1.2). Alizadeh and Cioffi-Revilla (2015) show how similar tweaking of the parameter λ can produce interesting results in models of opinion dynamics.

2.6 Parallelization

The vast majority of agent-based models run along a single thread of execution. Such models execute *serially*. In contrast, agent-based models can run in *parallel*, with multiple simultaneous threads of execution. Parallel computing is highly hardware-dependent, but this typically entails tasks being evaluated on different processors.

The primary motivation for running an agent-based model in parallel is performance. Especially in models with very large agent populations, the dense interactions of the agents can cause a superlinear or even polynomial increase in run time—i.e., increasing the size of the agent population by 10% may increase the run time by much more than 10%. Because the size of the agent population is a relevant parameter for many models (e.g., Crooks & Hailegiorgis, 2014), some models must be examined with extremely large agent populations to produce valid results.[5]

Performance may not be the only motivation to move to a parallel ABM. Although

[5]Note, however, that there be ways to run the model at reduced scale, see Osgood (2009) and Richardson, Richiardi, and Wolfson (2015).

more difficult to implement, parallel ABMs can be easier to reason about and explain to lay audiences because of their increased verisimilitude. Most real-world behaviors are parallel and asynchronous—all the relevant agents perform behaviors simultaneously and the model state (i.e., the world) updates accordingly. Some concurrency models, like Communicating Sequential Processes (Hoare, 1978), are said to "provid[e] a natural abstraction that can make some programs much simpler" (Cox, n.d.). Parallelization can also be used outside of the activation regime to render user interface (UI) elements or to perform logging and reporting functions (Auble, 2015); these applications are outside the scope of this thesis.

The difficulty of parallelizing an agent-based model depends substantially upon whether or not it is synchronously or asynchronously updated. Much of the challenging in implementation parallelizaion involves the management of shared memory (Lee, 2006); by using synchronous updating the global model state and therefore the shared memory is modified less often.

Spatial models present a unique challenge for parallelization. The global model state, now including the environment, is much larger. Depending on the specific model, collision detection may be necessary to prevent two agents from staying at the same location. For example, the Schelling (1971) segregation model assumes that two agents cannot reside at the same location, and therefore an agent has no more than eight neighbors. For this reason, some distributed ABM frameworks, including D-MASON, use synchronous updating.[6]

Another critical challenge in writing parallel ABMs is maintaining reproducibility. Many parallel-executing programs become nondeterministic (Lee, 2006), even if any random numbers are generated from a PRNG with fixed seed. Maintaining determinism often requires careful attention to the manner in which random numbers are generated and how memory is shared within the program. Reproducibility is often a valuable quality

[6]Specifically, D-MASON uses *field partitioning*, where the landscape is divided into segments; these segments are evaluated in parallel and synchronously updated (Cordasco, Milone, Spagnuolo, & Vicidomini, 2014).

in a model, but it is sometimes reasonable to trade off reproducibility for performance. For example, the GPU programming library FLAME-GPU, like most GPU architectures (but not all, see Jooybar, Fung, O'Connor, Devietti, & Aamodt, 2013), is explicitly non-deterministic (Coakley et al., 2006).

One common approach to parallelizing agent-based models is to divide the agent population into smaller subpopulations, then run these as if they were the full model on different cores. This "fork-and-join" method is extremely simple to implement and typically produces good speedups—it is "embarrassingly parallel" (Herlihy & Shavit, 2012)—but can bias the results in certain cases, by preventing the interaction of certain important agents with each other. For example, imagine an agent population A with attribute w that is distributed according to a Pareto distribution. If the population is partitioned into N equally-sized segments, it is possible that one of these partitions contains no agents whose w resides in the tail of the distribution.

2.7 Discussion

I believe it is worthwhile to think of agent activation as incorporating a number of interrelated subconcepts. Doing so allows for added clarity: for example, all synchronous and uniform activation regimes are the same but not all synchronous and non-uniform activation regimes. The presentation of these results are incomplete, probably by necessity. For example, in presenting selection methods I have limited myself to methods with a clear use case. However, many probability distributions could be used, for example a Weibull distribution of agent activations.

The lack of an obvious use case for such a selection order points to a caveat: more exotic activation regimes are not necessarily better. The choice of activation regime should tell a story about intended behavior of your model; random selection should imply a different

social process than Poisson selection. However, the diversity of possible stories and social processes also implies that there should not be a monoculture of activation regimes used.

The definition of a model turn provided at the beginning of this chapter has elided one notable distinction: the different between event-driven and time-driven updating regimes. In an effort to provide a comprehensive framework, I have ignored the case of time-driven updating. Many authors (Radax & Rengs, 2010; Schönfisch & de Roos, 1999, e.g.,) distinguish time-driven and event-driven updating as subsets of asynchronous updating. Considering time-driven updating requires a different definition of a model turn (and therefore uniformity) and the reward of doing so is fairly minimal. Not many agent-based models use time-driven updating and most can be easily represented as asynchronous event-driven models. Repast's scheduler uses a "ticks" mechanism that resembles time-driven updating but is implicitly converted to a discrete event scheduler (Collier & North, 2013, p. 1220). Comer (2014) shows how a nominally time-driven Poisson selection order can be converted to an event-driven one.

This analysis has assumed for simplicity and clarity that there is only one type of agent. This is true for many models, but not nearly all of them. An important question regarding activation with multiple agent type is: is it acceptable, feasible, or desirable to mix activations of agents of different types?

This analysis has omitted discussion of agent entry and exit. Exit is straightforward to handle under this framework, although it does change the length of the model turn over time. Handling agent entry should not be difficult either; the primary question being whether the agent is well-mixed into the original population or appended to it.

Chapter 3: A Systematic Review of the Usage of Different Agent Activation Regimes

3.1 Overview

I present a systematic review of how modelers have heretofore handled agent scheduling issues. The documentation and source code of 268 agent-based models were examined and classified according to the typology of Chapter 2. The analysis suggests major homogeneity regarding many of these decisions.

This appears to be the first time modeling decisions regarding agent activation have been examined in any large scale or systematic fashion. Angus and Hassani-Mahmooei (2015), in their survey of articles published in the *Journal of Artificial Societies and Social Simulation*, touch briefly on how agent scheduling issues are represented in published figures, but this is not a focus of their analysis.

3.2 Data

The data is sourced from OpenABM[1], maintained by the Network for Computational Modeling for SocioEcological Science (CoMSES). OpenABM provides a repository for researchers to host the source code for their agent-based models. As of March 5, 2016, OpenABM had 330 publicly-viewable entries; private source code was excluded from the analysis.

[1] `https://www.openabm.org/`

I downloaded all the files hosted for examination. Some models (approximately 20%) were thrown out for not being agent-based models; they were libraries, pedagogical examples, or microsimulations. Other models were thrown out because the source code was not provided or because they were designed for proprietary software, making the code too difficult to examine. The specific OpenABM entries that were excluded can be seen in Appendix B.

3.3 Methodology

Each model's source code and documentation was examined and categorized in terms of the components of the activation regime described in Chapter 2. The models were also categorized by whether or not the documentation was provided using the Overview, Design concepts, Details (ODD) protocol recommended by Grimm et al. (2010) and by whether or not the activation regime could be considered "default behavior" for the given modeling environment. In low-level programming languages, this means that agents were activated serially using a range-based for loop; in NetLogo, this means agents were activated using the `ask` command.

In some cases, one OpenABM entry contained multiple models. If the models were closely related, i.e., they were in the same language and had similar behavior, they were treated as one model. In cases where the models were written in different languages, this has been noted.

One difficulty when it comes to classifying NetLogo models is that the model behavior can be affected by the NetLogo version. In versions of NetLogo prior to version 4, the `ask` primitive (the primary way to query and activate agents) used what the documentation[2] describes as "simulated concurrent behavior." Within the ask block, each action that affects

[2] `https://ccl.northwestern.edu/netlogo/docs/programming.html#ask-concurrent`

the model state is broken up as finely as possible and each agent takes turns (randomly, of course), executing these chunks. This is still not full synchrony, so this behavior will be labeled *semi-synchronous*. If `ask-concurrent` is paired with the `without-interruption` primitive, the updating regime will be labeled fully asynchronous.

Another difficulty in examining NetLogo models is that most call `ask` more than once. Each time agent behavior is divided into its own `ask`, the model's activation regime becomes slightly less asynchronous (though not necessarily synchronous). For example, consider otherwise identical NetLogo models that do the following:

Model A

```
ask agents[
         doX
         doY
         doZ
]
```

Model B

```
ask agents [ doX ]
ask agents [ doY ]
ask agents [ doZ ]
```

The agents in Model A each perform `doX`, then `doY`, and then `doZ`. In Model B, each agent performs `doX`, then each agent performs `doY`, and then each agent performs `doZ`. However, within each task, the model state is still updated asynchronously. This "repeated ask" style therefore represents a sort of *partial asynchrony*.

This classification is a subjective process, and there is undoubtedly some error involved. In particular, deciding when a NetLogo model exhibits fully or partially asynchronous updating should be viewed as having relatively large error bars.

Much of the data is binary or categorical. In many cases predicting the modal result of categorical is not interesting; here we are interested in the *variability* of the data, which represents a heterogeneity of practices in the modeling community. The variance cannot be applied as it would with numerical data so the deviation from the mode (DM) is preferred

instead (Wilcox, 1973):

$$1 - \frac{\sum_{i=1}^{k}(f_m - f_i)}{N(K-1)}$$

where f_i is the frequency, K is the number of categories, and N is the sample size. DM produces a normalized uncertainty measure comparable to the variance. A value of 0 indicates exclusion of all but one category from the sample; a value of 1 indicates that the frequencies are uniformly distributed.

3.4 Results

In total 268 models (and their documentation) were analyzed. The full list of models can be found in Appendix C. The models exhibited significant homogeneity in many respects. Most (207) were developed in NetLogo. A notable minority were developed in Java (6 MASON, 25 Repast, 1 JADE, 12 other Java). One model was implemented in two languages (NetLogo and Cormas). Surprisingly, Python is quite poorly represented in the sample; only six models used Python. The overall distribution of activation methods is summarized in Figure 3.1

Most models (85%) use some form of random selection to activate their agents, per Figure 3.4, and nearly all (94%) activate their agents uniformly. There is near-unanimity with regards to reproduciblity (most models are) and parallelization (most models aren't). Only three models run in parallel; of those, two are implemented in such a way that agent activation is non-reproducible. These models are presented in Fachada, Lopes, Martins, and Rosa (2016); Rubio-Campillo, Cela, and Cardona (2013); and Sutcliffe, Wang, and Dunbar (2015).

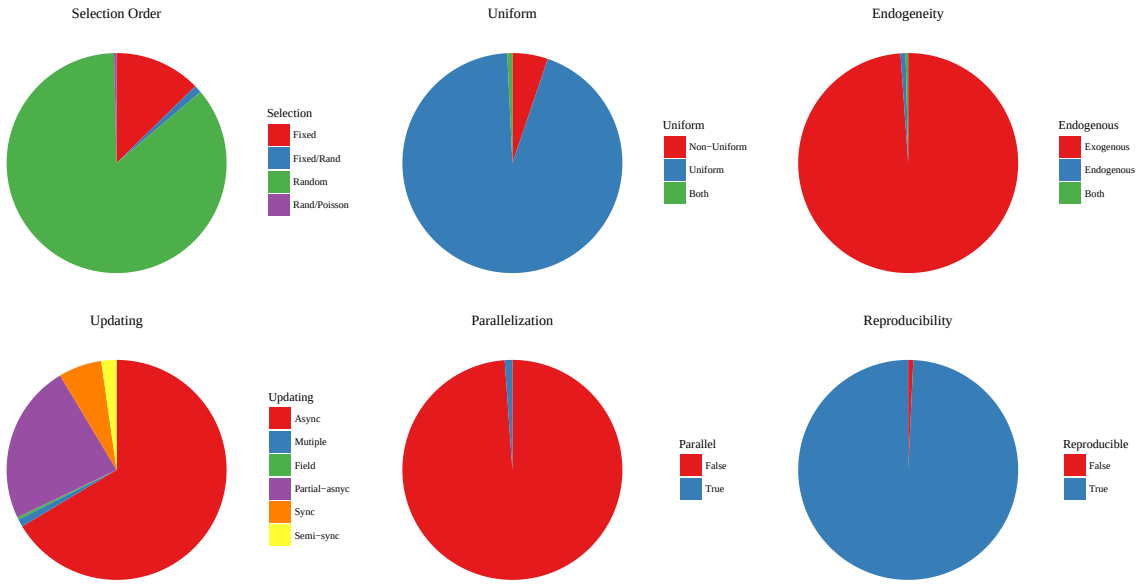

Figure 3.1: Pie charts showing the distribution of different areas of the activation regime across all public OpenABM models.

There is a pleasant heterogeneity in terms of updating regime: mostly thanks to NetLogo, only 178 models are fully asynchronous, while 17 are fully synchronous and the rest are in intermediate states. Most surprising is the unpopularity of endogenous updating methods; only two models (Alizadeh & Cioffi-Revilla, 2015; Crooks et al., 2015) select agents for updating based on some endogenous feature of the model.

A sizable minority (41%) of models included their documentation in a manner conforming to the ODD protocol, as shown in Figure 3.2. This represents a substantial buy-in to a fairly recent protocol (it was first promulgated in 2006 and updated in 2010).

One might hypothesize that the ODD protocol, by encouraging modelers to explicitly describe the scheduling process (the protocol contains a section labeled “Process overview and scheduling”), thereby encourages reflection about the activation process itself. This could lead to an increased heterogeneity of activation regimes, as modelers consider the best fit for the system under analysis. This hypothesis was examined quantitatively using

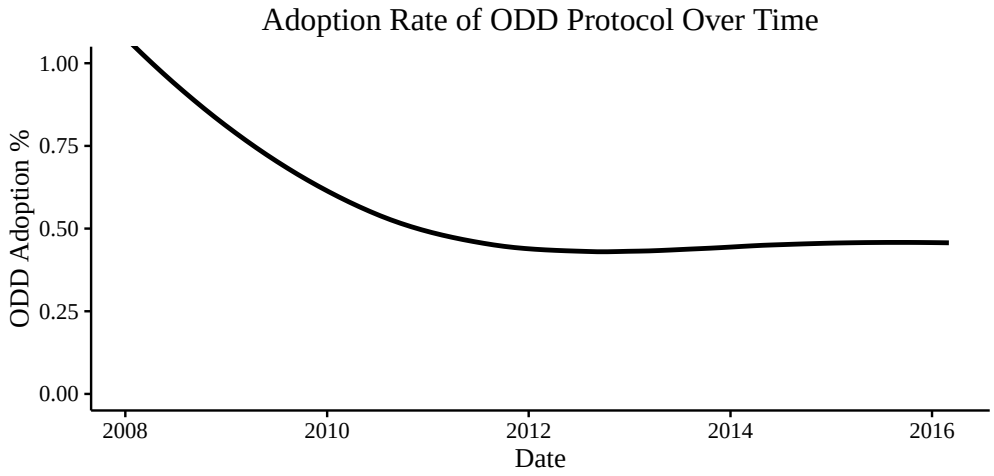

Figure 3.2: Adoption rate of ODD protocol over time. Note that the high early adoption rate is likely an artifact of researchers updating very old models.

the DM of each portion of the activation regime. Table 3.1 presents the relevant data; see Figure 3.3 for a visual representation of the variability with regard to the selection order.

The data do not support the hypothesis. The deviations from the mode go in both directions and are typically minor. If anything, the larger changes in DMs for selection and uniformity suggest the opposite effect: ODDs may induce conformity.

The choice of modeling platform, presented in Table 3.2, also has little effect on activation regime. NetLogo's `ask` primitive encourages the use of random selection, substantially decreasing heterogeneity with respect to selection order, but this is offset by much greater heterogeneity with regard to updating, as seen in Figures 3.4 to 3.5. These findings ease the concern by Radax and Rengs (2010) that NetLogo leads to increased conformity of modeling practices.

Table 3.1: Deviations from the mode for different aspects of the activation regime, contingent on model's use of the ODD protocol.

Type	ODD	NoODD
Selection	0.09	0.27
Uniformity	0.06	0.11
Updating	0.42	0.39
Endogeneity	0.01	0.02
Reproducibility	0.02	0.01
Parallel	0.03	0.01
n	119	149

Table 3.2: Deviations from the mode for different aspects of the activation regime, contingent on the modeling platform used.

Type	C	Java	MASON	Repast	NetLogo
Selection	0.59	0.56	0.44	0.64	0.03
Uniformity	0.17	0.00	0.00	0.06	0.08
Updating	0.13	0.50	0.20	0.14	0.45
Endogeneity	0.00	0.00	0.25	0.00	0.01
Reproducibility	0.22	0.17	0.00	0.00	0.00
Parallel	0.22	0.33	0.00	0.00	0.00
n	9	12	6	25	207

3.5 Discussion

An encouraging finding of this research is that modeling practices with regards to activation regime are not significantly affected by the choice of modeling platform. That this is true because there is such widespread conformity with respect to the activation is less encouraging. While heterogeneity with respect to reproducibility may not be desirable (*ceteris parbius*, reproducibility is better), relatively few modelers develop agent-based models in parallel or with endogenously changing selection orders. This is a disappointing finding because endogenous activation encourages positive feedback, something that is typically of interest to computational modelers.

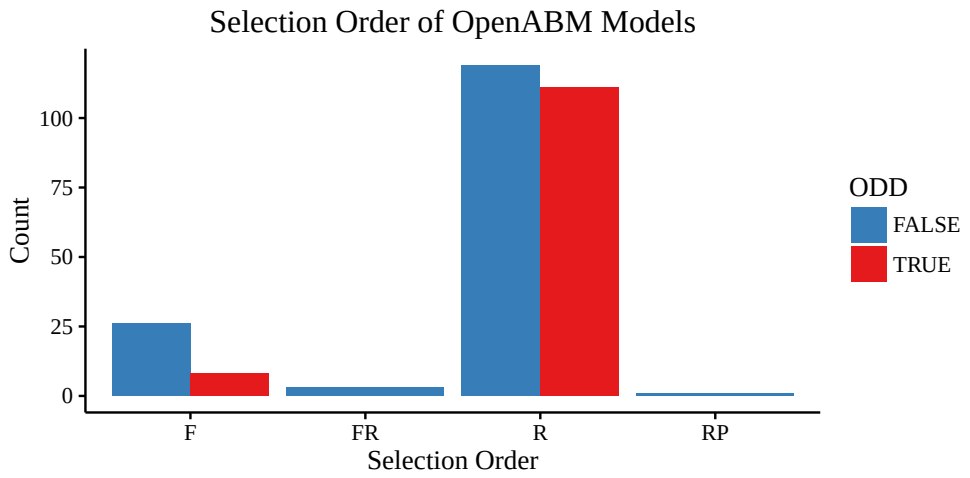

Figure 3.3: Selection orders used, contingent on ODD adoption. "F" is fixed, "R" is random, "P" is Poisson. Multiple letters means the model includes multiple activation regimes.

While the concerns of Radax and Rengs (2010) about NetLogo appear to be overstated, that the overwhelming majority of models in the repository used NetLogo warrants some attention. Radax and Rengs is correct that NetLogo encourages certain behaviors vis-à-vis agent activation, but in the case of selection this is probably a good thing. Around 10% of all models use non-endogenous fixed selection; while heterogeneity of practices is to be encouraged, Chapter 4 and other research (Axtell, 2000a) indicates that this particular activation regimes introduces serious bias. Additionally, NetLogo users seem to be far more inventive with regards to updating: the simplicity of using the `ask` primitive seems to encourage flexible usage.

This research suggests that parallel models are still extremely rare within the ABM community.

3.5.1 Future Work

Systematic, large-scale analysis of ABMs is a relatively rare thing. This analysis contributes to our understanding of how modelers are approaching agent activation issues but

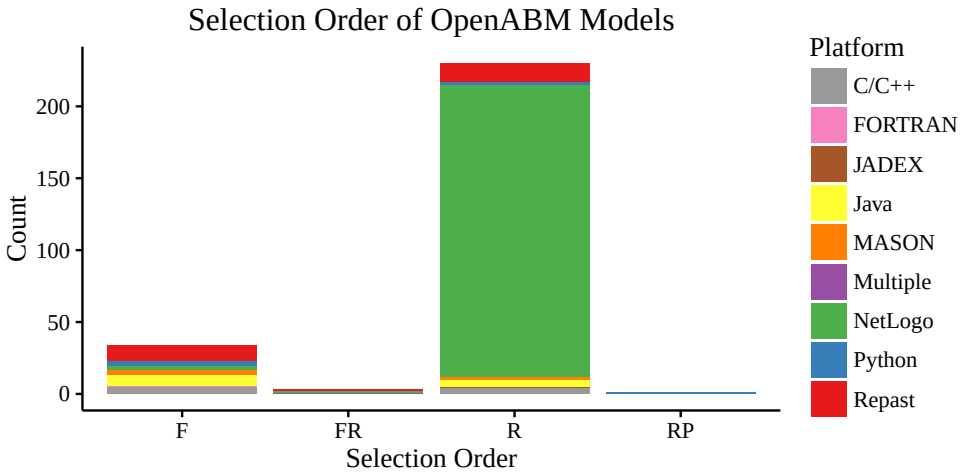

Figure 3.4: Selection orders used, contingent on modeling platform used. "F" is fixed, "R" is random, "P" is Poisson. Multiple letters means the model includes multiple activation regimes.

there is clearly more work that can be done.

It may be the case that the OpenABM repository is biased in some fashion. For example, perhaps that certain research institutions encourage the use of OpenABM, and these institutions also encourage the use of NetLogo, explaining its prominence within the repository. An extension of this review should therefore include other potential sources: the electronic appendices of prominent ABM-related journals, GitHub, etc.

One weakness of the approach used here is the subjectivity of the asynchronous/synchronous distinction for updating. As mentioned above and in Chapter 2, updating is more of a continuum than the other parts of the activation regime, which are typically fairly easy to classify. Developing a more objective or quantitative measure of how (a)synchronous a model is would likely be very insightful. The work of Baetens et al. (2012) using Lyapunov exponents may be helpful here.

Another approach that may be insightful is to distinguish ABMs from IBMs from the

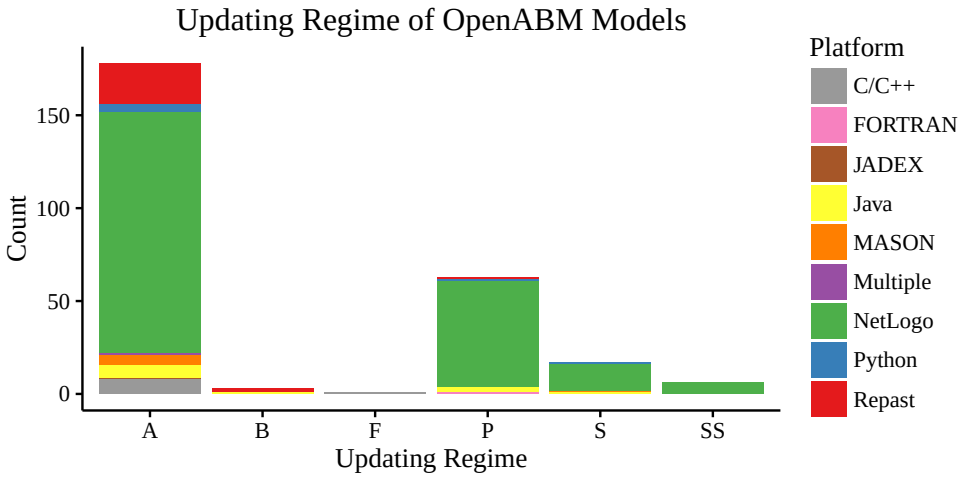

Figure 3.5: Updating systems used, contingent on modeling platform used.

review. It may be the case that many biological and ecological systems are better represented through synchronous updating than messier and more chaotic social systems. Is this represented in the modeling decisions of ABM and IBM researchers?

The same is true with regards to environment. There is intuitive reason to believe that spatial models are less sensitive to changes in the activation regime because agents' actions are mediated by the the environment.

One pleasant surprise of this research has been the widespread adoption of ODDs. Because ODDs include a great deal of information about the model workings (although not enough about activation) and are generally consistently structured and formatted, they may be well-suited for text mining.

Chapter 4: Studying Agent Activation in a Model of Bilateral Exchange

4.1 Overview

The most straightforward way to demonstrate the effect of varying agent activation regimes is to present a detailed analysis of how the regime effects a specific agent-based model. I present an agent-based model of economic exchange, originally developed by Axtell (2005), and demonstrate how different activation regimes affect model dynamics. This approach follows the recent work of Alizadeh and Cioffi-Revilla (2015) and Comer (2014), extending it to include a more diverse range of activation regimes, including parallel activation. The model presented is significantly more complex that those analyzed in previous work, demonstrating, *contra* the speculation of Fatès and Chevrier (2010), that these issues persist outside of so-called "toy models."

4.2 Model Specification

4.2.1 Overview

The model presented here was originally developed by Axtell (2005). In contrast to Walrasian exchange models, a group of economic agents engages in decentralized exchange of a number of commodities. The number of interactions between these decentralized agents required to equilibrate the population is a function of population size and the number of

commodities traded. For example, if equilibration is quantified based on the largest variance of the marginal rate of substitutions for each commodity, the number of bilateral interactions required is increases linearly with the size of the agent population (Axtell, 2005, p. F205).

4.2.2 Motivation

Chen (2012) identifies a number of motivations for agent computing in economics. The most prominent of these is the *markets approach*, which involves the "pursuit of a real construction... and hence a real understanding of markets" (p. 2). In particular, this research agenda often focuses on alternatives to the Walrasian model of price formation. Axtell (2005) focuses on the implausible computational complexity of the Walrasian auctioneer as a model of exchange. Extending existing work on fixed-point theorems (Geanakoplos, 2003; Hirsch, Papadimitriou, & Vavasis, 1989; Papadimitriou, 1994), Axtell concludes that "there are no polynomial time algorithms for the general case [of calculation of Brouwer fixed points] with nonlinear utility functions" (p. F196). The complexity of this algorithm places the task faced by Walrasian auctioneer outside of the class of "those that can be realistically solved by computers" (p. F197).

Axtell (2005) instead proposes a decentralized exchange process that can be simulated using agent-based modeling. This bilateral exchange model is as effective as the Walrasian model at converging to equilibrium but does so much more quickly. A list of model parameters can be found in Table 4.1. An agent population of size A trades N commodities. Agents interact randomly, trading commodities to satisfy their individual (heterogeneous) utility functions. Axtell demonstrates (p. F203) that the computational complexity of this mechanism is in polynomial time, making it a significantly more efficient, and therefore more plausible, model of price formation.

Table 4.1: Parameterization of the exchange model.

Symbol	Description
A	Size of the agent population
N	Number of commodities traded
τ	Time step
α	Cobb-Douglas preferences
w	Wealth
ε	Threshold required to initiate trade

4.2.3 Behavior

The model is written in C++, a low-level object-oriented programming language.[1] At runtime, a population of A agents and N commodities is instantiated. Each agent possesses an endowment of each commodity, uniformly distributed in the range $[w_{min}, w_{max}]$. By default, $50 \leq w \leq 150$. The average size of each agent's endowment is therefore approximately $N\frac{w_{min}+w_{max}}{2}$. Agents possess Cobb-Douglas preferences for each commodity. These preferences are also uniformly distributed in the range $(\alpha_{min}, \alpha_{max})$. These preferences are then normalized such that $\alpha_1 + \alpha_2 + \ldots + \alpha_N = 1$.

Agents are activated in pairs to evaluate potential trades. Each agent selects different commodities. If the positive increase in utility (evaluated using the marginal rate of substitution) from a potential trade is greater than e raised to a parameter ε, $0 \leq \varepsilon \leq 1$, exchange occurs. The number of trades per model turn can be specified by the user, but for consistency with the definitions earlier, experiments in this paper will define the number of trades per turn as $\frac{A}{2}$.

The marginal rate of substitution is calculated for each agent and each commodity. For C_1, the first commodity, the marginal rate of substitution is always one; that is, all other

[1] A specification of the model in terms of the ODD protocol (Grimm et al., 2010) can be found in Appendix A.

MRSes are defined in terms of the first commodity. For all other commodities the marginal rate of substitution is given by

$$MRS(C_{n>1}) = \frac{\alpha_n S_1}{\alpha_1 S_n}$$

where α_n is the Cobb-Douglas preference for C_n and S_n is the size of the allocation of C_n.

Upon activation, the two agents a^1 and a^2 each randomly select different commodities, C_x and C_y. If the ratio of their marginal rates of substitution for each commodity is greater than the model parameter ε, a trade is initiated. The amount of each commodity exchanged is given by:

$$\Delta_x = \frac{\alpha_x^1 \alpha_y^2 S_x^1 S_y^2 - \alpha_y^1 \alpha_x^2 S_y^1 S_x^2}{\alpha_x^2 S_y^2 + \alpha_x^1 S_y^1}$$

$$\Delta_y = \frac{\alpha_x^1 \alpha_y^2 S_x^1 S_y^2 - \alpha_y^1 \alpha_x^2 S_y^1 S_x^2}{\alpha_y^2 S_x^2 + \alpha_y^1 S_x^1}$$

The model can terminate under various conditions, but for simplicity in this analysis the model continues for 100 turns.

The utility function of the agents is given by:

$$U = {S_1}^{\alpha_1} \times {S_2}^{\alpha_2} \times \ \ldots \ \times {S_N}^{\alpha_N}$$

where S is the size of the allocation and α is the preference for commodities one through N.

There are only local prices in the model. The global price is only an aggregate of individual preferences for commodities. An individual's "wealth function," its evaluation

of the size of its allocation of commodities, is the dot product of the vector of MRSes and the vector of commodity allocations:

$$w = \begin{bmatrix} MRS_1 \\ MRS_2 \\ \cdots \\ MRS_N \end{bmatrix} \cdot \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix}$$

The MRSes are the subjective marginal rates of substitution for the individual agent.

Synchronous updating was *a priori* rejected for this model. In a model of exchange, it makes little sense to have synchronous updating. There is no global state considered by the agents and it is not clear what collision resolution would look like in this case.[2]

4.2.4 Performance

The model is capable of handling very large agent populations trading a number of commodities, subject to processing power and time. Its moderate runtime and large number of interactions make it well-suited as a test model for parallelization. Figures 4.1 and 4.2 present some benchmarking results for the serial version of the model. As predicted by Proposition 11 of Axtell (2005), the basic relationship between A and the runtime of the model is linear.

For compatibility with parallel execution, the model uses an array of PRNGs, one for each thread of execution. To ensure that each PRNG has a different random seed, their randomly-assigned hash value is added to the base random seed used. This means that as

[2]Note that this does not mean that all economic ABMs should be asynchronous; for example it may make sense to represent stock markets, with their bid and ask orders, synchronously.

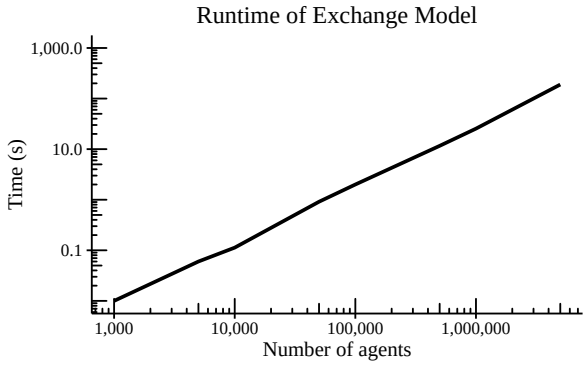

Figure 4.1: Runtime of the bilateral exchange model, $N = 2$.

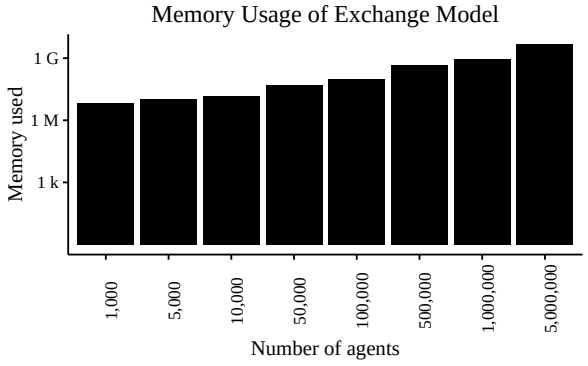

Figure 4.2: Memory usage of the bilateral exchange model, $N = 2$.

implemented the model is not reproducible from run to run. Future work will tweak this implementation in an effort to improve reproducibility of model results.

4.3 Methodology and Results

4.3.1 Environment

All experiments were run within a Docker session on a 16-core computer with 256 GB of RAM running Fedora Linux.[3] Docker is a service for maintaining and running lightweight virtual machines that may be useful for ensuring reproducible research (Boettiger, 2015).

4.3.2 Serial Activation Regimes

The class of serial activation regimes were examined first. For each regime, 25 model runs were performed, each lasting 100 turns. For these runs, $A = 100000$ agents traded $N = 10$ commodities. Initial endowments were uniformly distributed in the range $[50, 150]$ for each commodity and preferences were uniformly distributed in the range $[0.01, 0.99]$ for each commodity. Trades were executed if there was a mutual gain in utility of at least one percent.

Six activation regimes were examined for this experiment. For consistency, some facets of the activation regime were held constant—serialized activation, asnychnrouous updating, non-reproducible random number generation. Table 4.2 provides an overview of the six regimes. All three Poisson regimes are similar, differing only in how the endogenous activation parameters λ is defined:

[3]The model code, including the Dockerfile used to generate the runtime environment, is available in Appendix D or at `https://bitbucket.org/mccabe_s/bilateral-exchange`. The model was written to compile with GCC 5.3; compatibility with other compilers is not guaranteed.

Table 4.2: Overview of serial activation regimes examined.

Regime	Label	Selection order	Uniform?	Endogenous?
1	Fixed	Fixed	Yes	No
2	Random	Random	No	No
3	Uniform	Random	Yes	No
4	Poisson-Poor	Poisson	No	Yes
5	Poisson-Rich	Poisson	No	Yes
6	Poisson-Middle	Poisson	No	Yes

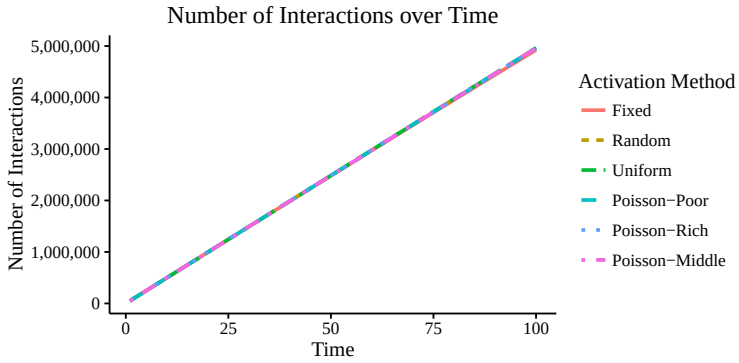

Figure 4.3: Number of interactions (successful trades) over time, serial activation.

- In Poisson-Poor, the activation parameter λ is defined as the inverse of the agent's wealth.[4]
- In Poisson-Rich, the activation parameter λ is defined as the agent's wealth.
- In Poisson-Middle, the activation parameter λ is defined as the distance of the agent's wealth from the population's mean wealth.

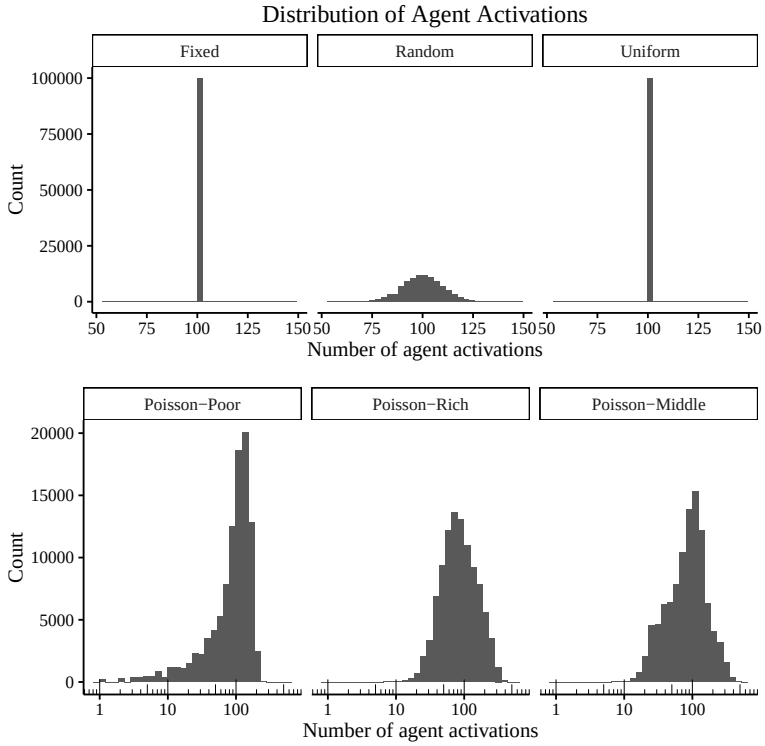

Figure 4.4: Histograms of agent activations, serial activation. Note that the second row has log-10 axes.

Results

Varying the activation regime has significant effects on model behavior and outcomes. Varying the activation regime does not, however, significantly affect the total number of interactions (successful exchanges) within the model, as seen in Figure 4.3. Therefore, one is led to the conclusion that changes in activation regime are successfully affecting the "interaction topology" (Axtell, 2000a) rather than simply the raw number of trades.

The effect of activation regime on the "interaction topology" can be further demonstrated by examining the distribution of trade opportunities afforded each agent. Figure 4.4

[4]Because there is no currency in this model, there is no "objective" wealth as would typically be understood in the model. Wealth here is a subjective measure computed by the dot product of two vectors, the agent's current endowments and its current marginal rates of substitution.

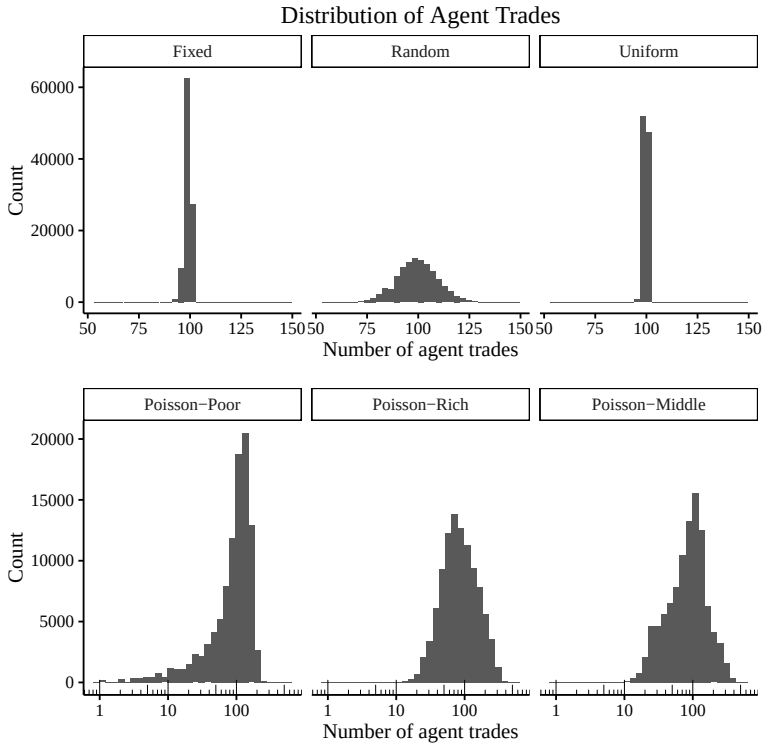

Figure 4.5: Histograms of successful agent trades, serial activation. Note that the second row has log-10 axes.

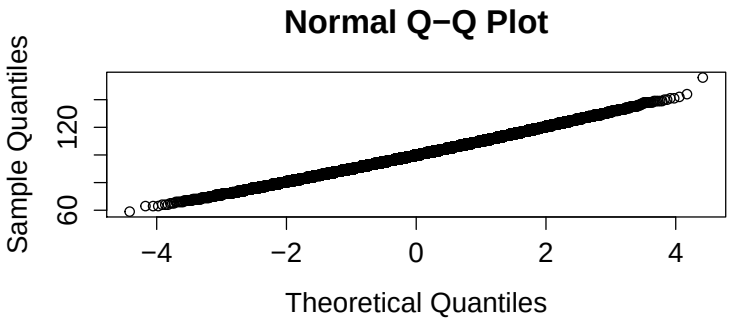

Figure 4.6: QQ plot of agent activations, serial random selection.

shows the histograms of agent activations for a single, representative run for each activation regime, while Figure 4.5 shows the histogram of successful trades. The two uniform activation regimes (including fixed activation here) have no variability, of course: they have been defined such that each agent gets exactly the same number of activations. The (non-uniform) random activation regime has some variability but it follows a normal distribution,[5] as shown by the Q-Q plot in Figure 4.6.

The Poisson activation regimes exhibit more interesting behavior. To make visualization easier, log10 axes were used here. In the Poisson-Poor regime, the distribution of activations is slightly left-skewed with a much higher standard deviation ($\sigma \approxeq 49$) than the random activations. In the Poisson-Poor activation, a very small number of agents actually fail to get any activations. This is not true of the Poisson-Rich regime, though it has a higher standard deviation ($\sigma \approxeq 60$). It is relatively right-skewed. The Poisson-Poor is also right-skewed and has similar variability ($\sigma \approxeq 63$).

Perhaps the most notable effect of varying activation regime can be seen in the effect on the poorest agents in the simulations. Figure 4.7 shows the increase in the poorest agents' wealths over time. Under the Poisson-Rich regime (activation favors the richest agents),

[5] Specifically it is distributed $N(100, 100)$.

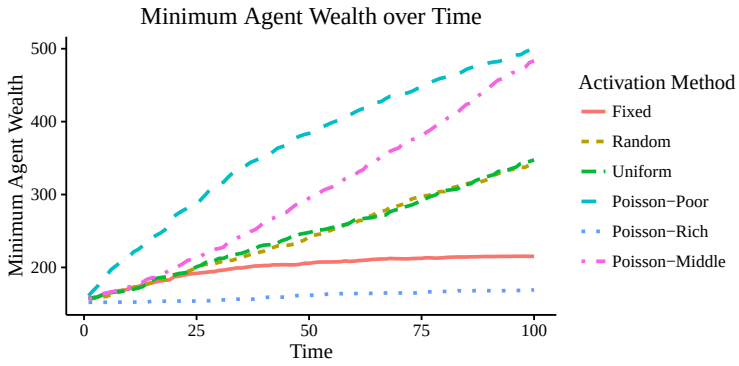

Figure 4.7: Minimum agent wealth over time, serial activation.

the systematic exclusion of the poorest agents from the model prevents any meaningful increase in the wealth of the poorest agent. Perhaps counterintuitively, while Poisson-Poor increases the minimum wealth more than the others in the short run, this does not hold over the long run.

In Axtell (2005), the exchange model terminated when the variance of the agents' MRSes fell below some threshold, typically equal to the trade parameter ε. Figure 4.8 shows how the convergence rate of the model is affected by the activation regime. Fixed activation models never converge: the population is not sufficiently well-mixed for widespread trade. The Poisson-Middle regime converges nearly as quickly as the uniform and random regimes, but the other two Poisson regimes have much more gentle slopes, suggesting that these also do not produce well-mixed populations within the market.

4.3.3 Parallel Activation Regimes

The same set of activation regimes was examined in a parallel computing environment. Once the scheduler activated a pair of agents, their trade function was sent to a thread pool for processing. The size of the thread pool was determined by the system hardware; the number of threads used should not be a relevant model parameter in this scenario. This

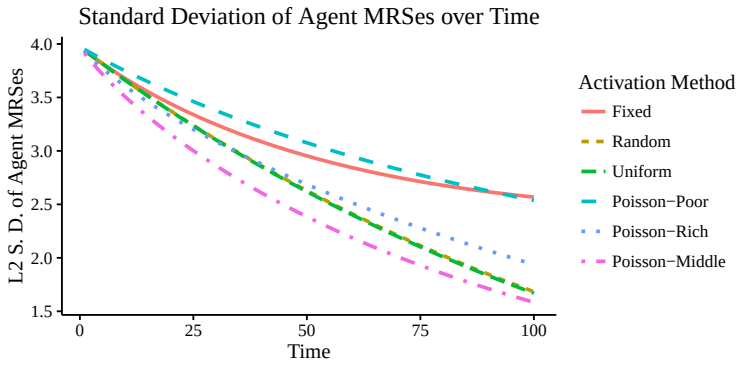

Figure 4.8: Standard deviation of agent marginal rates of substitution, serial activation.

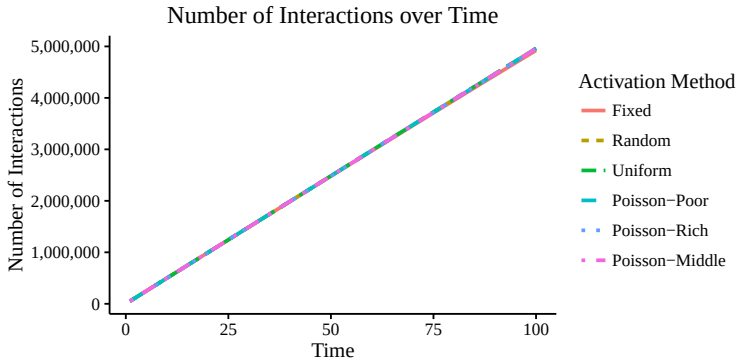

Figure 4.9: Number of interactions (successful trades) over time, parallel activation.

design pattern should produce functionally similar results to the serial activation case.

Parallel execution of the model produces generally similar interaction topologies when compared with the serial case. Figures 4.9 to 4.11 provide visual evidence that the distribution of agent activations is not affected at either the macro-level or the micro-level. These results suggest that the model can be effectively parallelized with respect to program correctness; the performance of the parallel model will be discussed below.

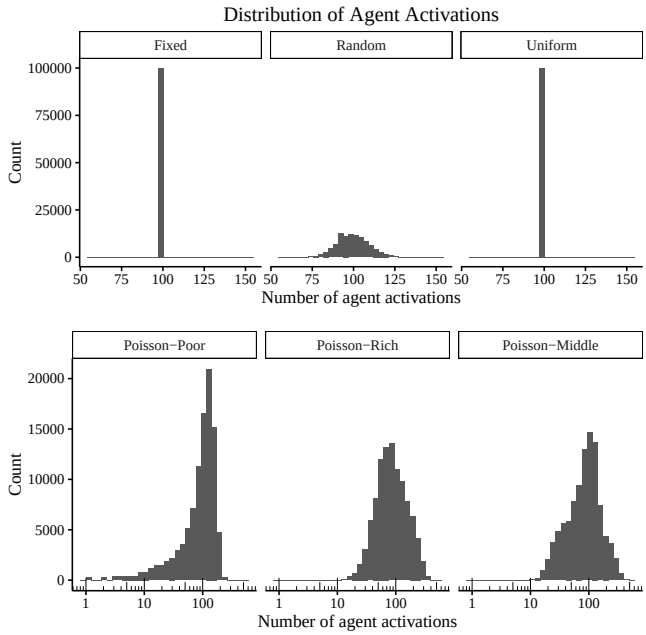

Figure 4.10: Histograms of agent activations, parallel activation. Note that the second row has log-10 axes.

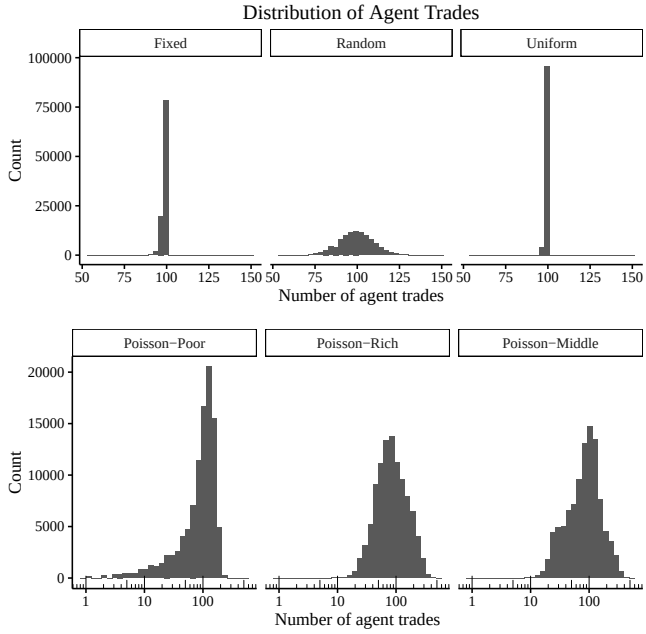

Figure 4.11: Histograms of successful agent trades, parallel activation. Note that the second row has log-10 axes.

4.3.4 Fork-and-Join Activation Regimes

Another approach to parallelizing agent-based models is to divide the agent population into a number of subsections and then run these subpopulations in parallel. Particularly for economic models, where interactions are mostly decentralized, this can be an attractive way of increasing model performance; the code used is a rather simple extension of the serial case. However, there is some risk that it could bias the model. The circumstances under which fork-and-join is a safe approach are not well-known.

For this model, only fixed, random, and uniform activation were considered. The Poisson regimes are not well-defined for fork-and-join; are the λs of the agents defined relative to the model population as a whole, or just to the subpopulations? Should each subpopulation get an equal number of activations?

Visual insepction of the results in Figures 4.12 to 4.15 suggest that this model is well-suited to parallelization via fork-and-join; the key model variables do not appear obviously biased in any fashion by division into subpopulations. In some ways, this is unsurprising; in a model of decentralized exchange it does not necessarily matter who you interact with as long as there is sufficient heterogeneity. However, it is surprising in other respects. If some important variable of the model followed a skew distribution, the behavior of the model could have changed significantly between subpopulations. This will be explored further in the next section.

4.4 Performance

For the previous experiments, model runtimes were also recorded. The runtimes of different models can be found in Figure 4.16. Performance gains from parallelization were modest. In the best case (random activation), moving from serial to parallel activation produced a 29% speedup. Using fork-and-join activation had a poor effect on model runtimes,

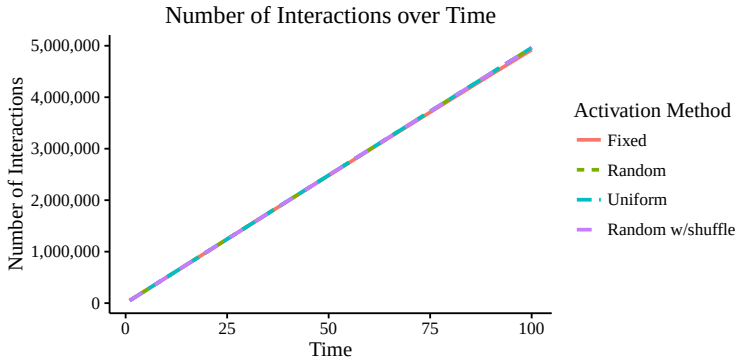

Figure 4.12: Number of interactions (successful trades) over time, fork-and-join activation.

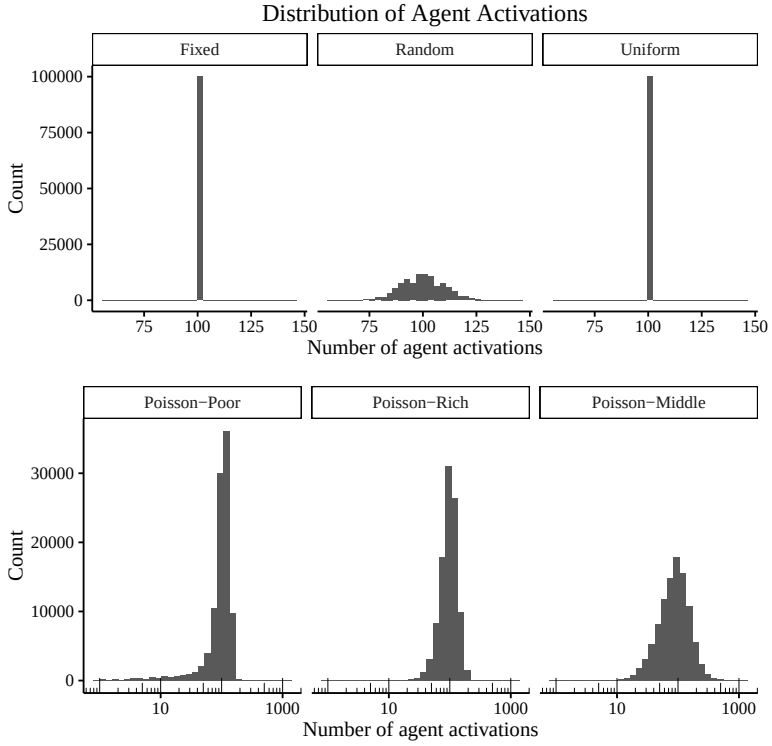

Figure 4.13: Histograms of agent activations, fork-and-join activation. Note that the second row has log-10 axes.

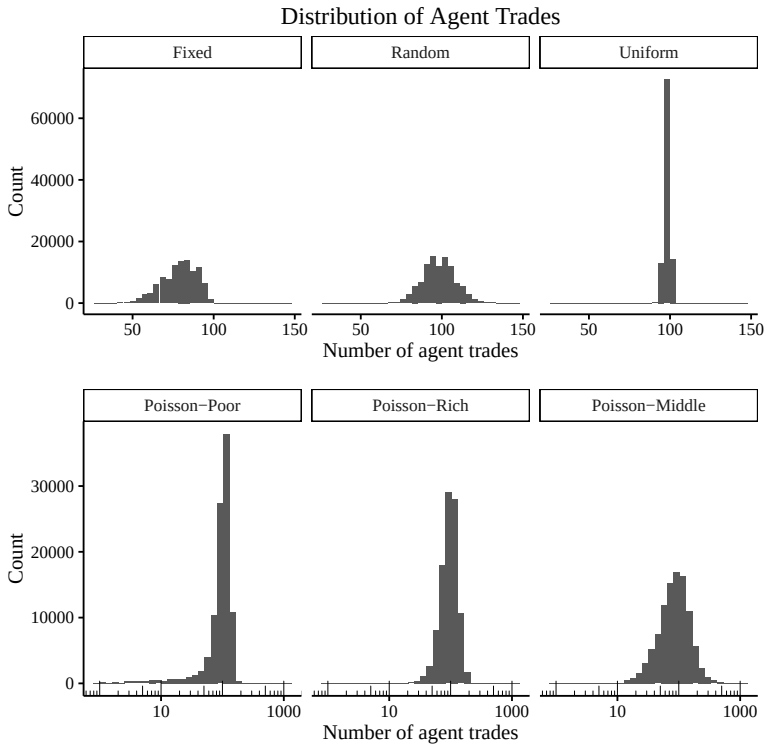

Figure 4.14: Histograms of successful agent trades, fork-and-join activation. Note that the second row has log-10 axes.

Figure 4.15: Standard deviation of agent marginal rates of substitution, fork-and-join activation.

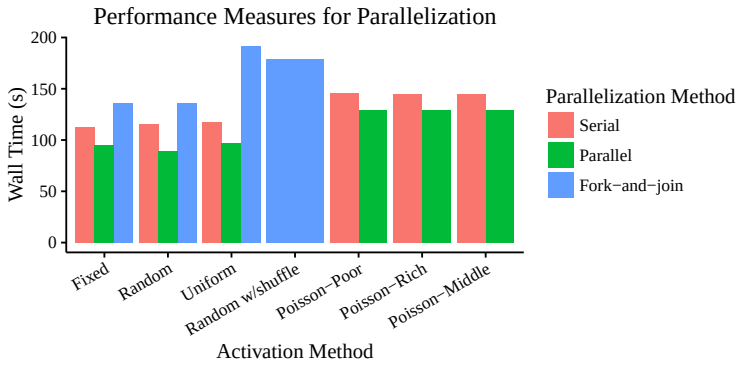

Figure 4.16: Performance measures for different implementations.

producing negative speedups across the board.

Although parallel activation improved model runtimes without sacrificing model correctness, the increase in developer workload likely would not justify the effort of transitioning an existing model from serial to parallel activation.

4.4.1 Scaling in Fork-and-Join

In the previous section, the fork-and-join implementation used the system default number of threads. In that case, 32 threads were used. Here I examine the effect of varying the number of threads—and therefore the number of model subdivisions—on model behavior and performance.

As before, $A = 100000$ traded $N = 10$ commodities. The effect on model runtime of varying the model parameter t, the number of threads, can be seen in Figure 4.17. Twenty-five runs were performed.

Changing the number of threads used by the model can improve the performance of the model. Although initially slower than the serial model, once $t > 50$, the runtime falls below the serial case.. However, even in this case the speedup is relatively minor. If the model population is shuffled after each turn to avoid bias, this speedup essentially vanishes.

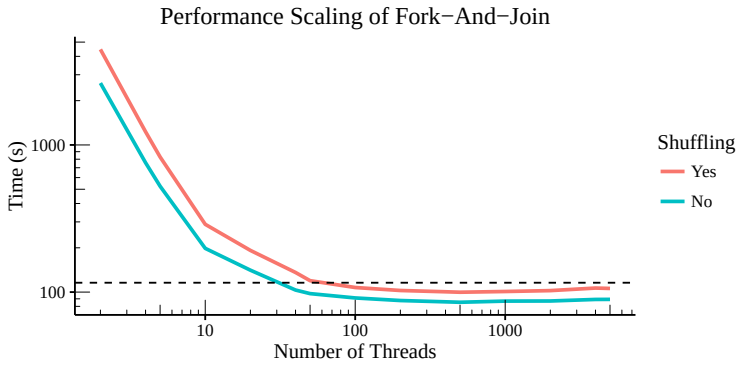

Figure 4.17: Scaling of fork-and-join implementation. Random and random-with-shuffle only. The dashed line represents the average runtime of the serial model.

More important than performance is, of course, model correctness. The first concern is that dividing the population into subpopulations may reduce the number of opportunities for trade. Figure 4.18 shows the number of trades made by agents over time. Varying the number of threads[6] does not affect the the raw number of successful trades in any meaningful sense, a finding consistent with earlier results regarding the activation regime. The question is therefore: is the interaction topology biased by the division into subpopulations?

The answer to that question appears to be yes. Figure 4.19 and Figure 4.20 present two key model outputs—the average agent utility and the standard deviation of agent MRSes—over time. As the number of threads increases, the convergence rate of the model slows down. (Note, however, that the highest values for the number of threads are quite extreme: 1000 threads for 10000 agents.) This result suggests that there is a trade-off: fork-and-join can improve performance but at the cost of slightly biasing model results. This can be ameliorated by shuffling the subpopulations between turns, but then the performance benefit of fork-and-join is lost. Combined, these results suggest that fork-and-join activation is poorly suited to (some subset of) economic ABMs.

[6]To clarify: a specific set of threads was used in the parameter sweep. To be considered, the number of threads must be a divisor of both A and $\frac{A}{2}$. This is to prevent any bias from truncating the model or running too few trades during a turn.

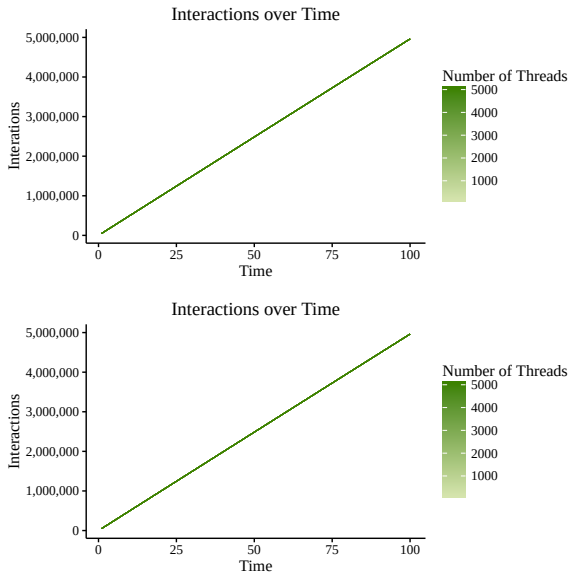

Figure 4.18: Number of interactions as the number of threads are varied. Above is random activation, below is random-shuffle activation.

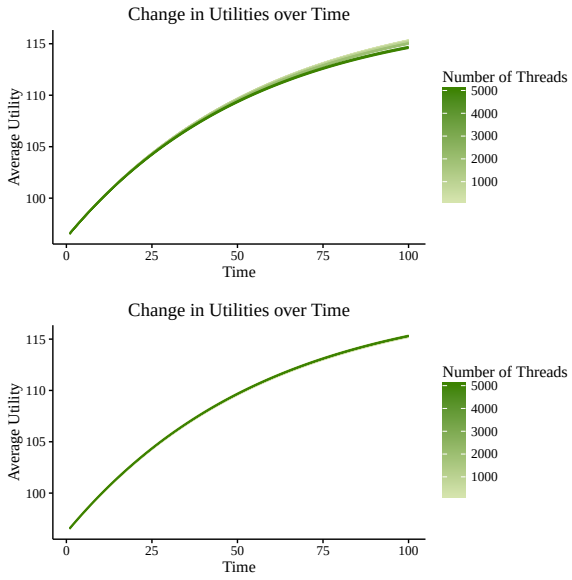

Figure 4.19: Change in mean agent utility as the number of threads are varied. Above is random activation, below is random-shuffle activation.

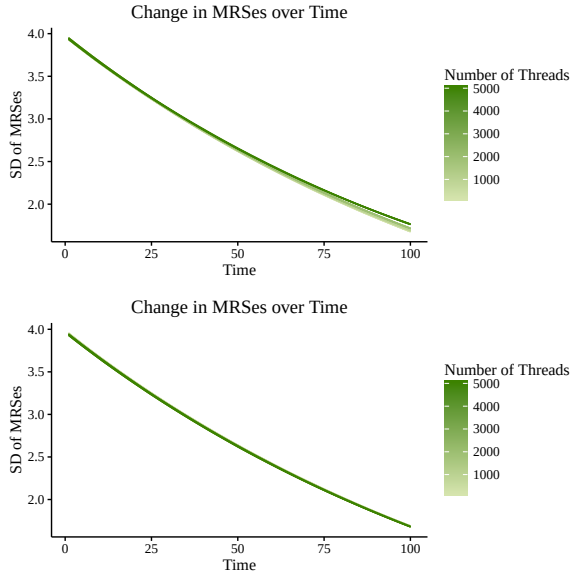

Figure 4.20: Change the standard deviation of agent MRSes as the number of threads are varied. Above is random activation, below is random-shuffle activation.

4.5 Discussion

Aside from the parallel results, in the serial case it has been demonstrated that the distribution of agent activations continue to play an essential role in ABM outcomes even as the complexity of the model increases.

In terms of model correctness, only reproducibility is compromised by parallelization, regardless of whether thread pools or fork-and-join are used. In terms of performance, the fork-and-join implementations perform quite poorly, suggesting that this model, though more complex than the toy models typically used to test agent activation regimes, still has agent behavior too simple to be efficiently parallelized through fork-and-join. Parallel activation using a thread pool can deliver modest speedups but may not be worth the effort.

A productive way to think of agent activation, particularly in economic ABMs with pairwise activation, is to analogize the agent-based model to a random graph model. In the highly biased fixed activation case, there is no giant component to the network, only

an assortment of random dyads. The random activation process is quite similar to the random graph model of Erdős and Rényi (1960). There is no direct analogue to the scale-free model (Barabási & Albert, 1999), but one could imagine perhaps some lognormal activation process producing similar results. All of these models are viewed as extremely different and the choice of which algorithm is correct to use in a given case generates serious discussion; the same should be true of agent activation.

4.5.1 Future Work

This model is extremely powerful and there are many natural extensions. The most directly relevant to this research is to introduce an exogenous Poisson activation regime. Agents have some initiative parameter, distributed uniformly or normally, represented their level of entrepreneurship. How does the exogenous Poisson model compare to the random activation model? How does it compare to the endogenous Poisson model? Is the behavior of the endogenous Poisson model a function of its exponential arrival times or of its positive feedback?

Another extension is to increase the resolution of the micro-level data produced by the model. Generating a list of proposed and successful trades between agents can produce a weighted graph of economic exchange. Are there interesting features of the network, e.g. do more central agents benefit more from trade? At a minimum, this extension would be useful as a visualization tool for demonstrating the wide-ranging effects of changing the activation regime.

One current limitation of the model is, because exchange is decentralized, there are only local prices. This is, of course, a positive feature of the model (there is no Walrasian auctioneer), but it does mean that model outputs are subjective to the agent. Future work could focus on refining model outputs, either in terms of raw commodity allocations or in terms of estimating a global price.

Finally, to make parallelization slightly easier the random number generator was implemented in a nondeterministic fashion. Future work should improve the implementation of the random number generator to improve reproducibility of model results.

Appendix A: Overview, Design Concepts and Details for Exchange Model

A.1 Overview

This is a specification of the exchange model in terms of the ODD protocol (Grimm et al., 2010). The model was originally developed and presented in Axtell (2005); I have only extended it.

A.1.1 Purpose

The purpose of the exchange model, as presented in this thesis, is as a reference model for experimenting with parallelization methods and activation regimes; in this sense its purpose is pedagogical and comparable to the PPHPC model of Fachada, Lopes, Martins, and Rosa (2015).

In its original implementation, it was an effort to use computational methods to establish a competing foundation to microeconomic theory, replacing the theoretically implausible concept of the Walrasian auctioneer (as formalized in the Arrow-Debreu model). The decentralized exchange process represented in the model was more plausible and more efficient than the idea of the auctioneer setting prices.

A.1.2 State Variables and Scales

The model contains a population of agents exchanging some number of *commodities*; the specific numbers here are key model parameters. There is no entry or exit of agents in the model. Agents possess a random (uniformly distributed) endowment of each commodity and a random (uniformly distributed) preference (α) for each commodity. The minimum

and maximum values for these two attributes are specificied by the user; with the limitations that endowment sizes belong to the set of natural numbers and preferences are floating-point numbers that should be in the range $0 < \alpha < 1$.

After α has been assigned for each commodity, they are normalized such that the sum of the αs for each commodity equals one. From their current allocation of commodities and preferences they construct a series of indifference curves for each good, i.e. the marginal rate of substitution (MRS) for each good.

The marginal rate of substitution is calculated for each agent and each commodity. For C_1, the first commodity, the marginal rate of substitution is always one; that is, all other MRSes are defined in terms of the first commodity. For all other commodities the marginal rate of substitution is given by

$$MRS(C_{n>1}) = \frac{\alpha_n S_1}{\alpha_1 S_n}$$

where α_n is the Cobb-Douglas preference for C_n and S_n is the size of the allocation of C_n.

The utility function of the agents is given by:

$$U = {S_1}^{\alpha_1} \times {S_2}^{\alpha_2} \times \ \ldots \ \times {S_N}^{\alpha_N}$$

where S is the size of the allocation and α is the preference for commodities one through N.

There are only local prices in the model. The global price is only an aggregate of individual prefrences for commodities. An individual's "wealth function," its evaluation of the size of its allocation of commodities, is the dot product of the vector of MRSes and the

vector of commodity allocations:

$$w = \begin{bmatrix} MRS_1 \\ MRS_2 \\ \cdots \\ MRS_N \end{bmatrix} \cdot \begin{bmatrix} S_1 \\ S_2 \\ \cdots \\ S_N \end{bmatrix}$$

The MRSes are the subjective marginal rates of substitution for the individual agent.

A key behavioral parameter of the model is the trade parameter ε. When two agents consider a trade (see below), it only occurs if their prospective increase in utility is greater than e^{ε}. The other important behavioral parameter is the number of trades per turn; for consistency with the terminology used in Chapter 2 this is fixed at the number of agents divided by two.

Most of the other model parameters govern the activation regime and the parallelization of the model. The exchange model currently has six activation regimes, summarized in Table 4.2. The model can be run serially or in parallel. If the model is run in parallel, the task handled in parallel can either be each agent's trade or trades within a subpopulation of agents ("fork-and-join").

A.1.3 Process Overview and Scheduling

Upon activation, two agents a^1 and a^2 each randomly select different commodities, C_x and C_y. If the ratio of their marginal rates of substitution for each commodity is greater than e^{ε}, a trade is initiated. The amount of each commodity exchanged is given by:

$$\Delta_x = \frac{\alpha_x^1 \alpha_y^2 S_x^1 S_y^2 - \alpha_y^1 \alpha_x^2 S_y^1 S_x^2}{\alpha_x^2 S_y^2 + \alpha_x^1 S_y^1}$$

$$\Delta_y = \frac{\alpha_x^1 \alpha_y^2 S_x^1 S_y^2 - \alpha_y^1 \alpha_x^2 S_y^1 S_x^2}{\alpha_y^2 S_x^2 + \alpha_y^1 S_x^1}$$

The order in which the agents is, of course, a key concern of the thesis. The activation regimes used in the analysis are summarized below.

- In the *fixed* activation regime, a^1 and a^2 are activated, then a^3 and a^4, and so on. This regime is extremely poorly suited to economic exchange models and is included only for demonstration purpose.

- In the *random* activation regime, two agents are selected at random from a uniform distribution. This regime is non-uniform; i.e., the same agent can activate more than once per turn and agents are not guaranteed an activation each turn.

- In the *uniform* activation regime, agents are selected randomly but are guaranteed exactly one activation per turn (i.e., sampling without replacement). At the implementation level this is represented as a fixed activation regime on a permutation of the agent population, with a new permutation generated at the end of each turn.

- There are three *Poisson* activation regimes. These regimes are slight adaptations of the ideal Poisson clock to be event-driven rather than time-driven. The algorithm used for all three can be seen in Algorithm 1 in Chapter 2, with the difference being how the activation parameter λ is defined. In the *Poisson-Poor* regime, $\lambda = \frac{1}{w}$, where w is the agent's wealth. In the *Poisson-Rich* regime, $\lambda = w$. In the *Poisson-Middle* regime, $\lambda = |w - \bar{w}|$.

If fork-and-join activation is used, the Poisson regimes are not available. Agents can only interact with agents inside their own subpopulation; the random number generator is bound by the size of the subpopulation. A separate model parameter governs whether the subpopulations are randomized each turn.

The model updates asynchronously. In parallel activation the global model state is kept consistent through the use of mutexes.

The model continues until a termination condition is met. The model has five termination conditions:

1. The model terminates after a fixed period of time.

2. The model terminates after the variance of all the MRSes for the various commodities falls below some threshold.

3. The model terminates after the largest variance of the MRSes for the various commodities falls below some threshold.

4. The model terminates after the sum of agent utilities falls below some amount in one turn.

5. The model terminates after the sum of agent utilities falls below some threshold.

A.2 Design Concepts

A.2.1 Emergence

The key contribution of Axtell (2005) is to show how prices can arise through a decentralized process in the absence of the Walrasian auctioneer. This is an emergent feature of the model.

A.2.2 Interaction

The economic exchange of the model consists of repeated pairwise interactions. One of the fundamental questions of the thesis is how the patterns of interactions—i.e., the activation regime—affect the model behavior.

A.2.3 Stochasticity

The model is highly stochastic. Most model parameters are randomly distributed; agent activation and commodity selection are also random. That the model outputs are so consistent in the face of so much stochasticity suggests that the underlying behavior is quite robust.

A.3 Details

A.3.1 Initialization

As mentioned above, an agent population of size A is initialized. Each agent possesses a random allocation of N commodities and Cobb-Douglas preferences for each commodity.

A.3.2 Input

The model parameters are read in from a configuration file. There is no other model input.

A.3.3 Submodels

There are no submodels of note.

Appendix B: OpenABM Entries Excluded From Review

#	Author	Date	Reason
2235	Lee	2009-01-29	No source code
2236	Voronovitsky	2009-01-24	No source code
2241	Lee	2009-01-09	No source code
2243	Zhang	2009-05-29	Proprietary software
2244	Murray-Rust	2009-07-20	Pedagogical example
2248	Jaffe	2009-09-14	Incomplete source code
2254	Abrami	2009-12-03	No source code
2263	Hufschlag	2010-04-09	Not an ABM
2272	rolanmd	2010-08-09	Proprietary software
2280	Grazzini	2010-11-28	Not an ABM
2281	Holzhauer	2010-12-01	No source code
2282	Grazzini	2010-11-28	Not an ABM
2441	Siebers	2011-02-23	Proprietary software
2466	Kim	2011-03-25	Not an ABM
2492	Bommel	2011-04-05	System not well-documented
2524	Zhang	2011-05-19	Proprietary software
2589	Truscott	2011-08-14	System not well-documented
2611	Levinson	2011-08-29	No source code
2617	Levinson	2011-08-29	No source code
2623	Nye	2011-08-30	System not well-documented
2656	Nazari	2011-10-05	Not an ABM
2717	Macedo	2011-11-07	Not an ABM

#	Author	Date	Reason
2756	Andrade	2011-11-20	Not an ABM
2870	Briner	2012-01-31	Proprietary software
2880	Bohensky	2012-02-06	System not well-documented
2905	Lawson	2012-02-27	Not an ABM
2908	Lawson	2012-02-27	Not an ABM
2965	Garip	2012-04-27	Proprietary software
3002	Shamsaee	2012-05-14	Not an ABM
3172	Schenk	2012-09-20	System not well-documented
3368	Magliocca et al.	2012-11-02	Proprietary software
3613	Boyle	2013-02-03	No source code
3760	Sibertin-Blanc et al.	2013-05-19	No source code
3792	Sibertin-Blanc et al.	2013-05-16	No source code
3807	Becu et al.	2013-05-27	Source code readability
3865	Dolado et al.	2013-07-11	No source code
3900	Hu	2013-08-09	Proprietary software
3936	Ozbas et al.	2013-09-09	Not an ABM
3939	Magliocca et al.	2013-09-09	Proprietary software
3983	Teran and Sibertin	2013-10-20	Incomplete source code
4025	Grimaldo and Paolucci	2013-11-10	System not well-documented
4036	Le Page and Bobo	2013-11-14	Incomplete source code
4079	Leighton	2014-01-06	Incomplete source code
4159	Tian	2014-03-21	Proprietary software
4161	Edali and Yasarcan	2014-03-21	Not an ABM
4163	Edali and Yasarcan	2014-03-21	Not an ABM
4166	Edali and Yasarcan	2014-03-26	Not an ABM

#	Author	Date	Reason
4220	Reardon et al.	2014-05-23	Not an ABM
4310	Smarzhevskiy	2014-08-19	Proprietary software
4486	Ceschi	2015-01-12	System not well-documented
4490	Zinn	2015-01-14	No source code
4499	Martin	2015-01-15	Not an ABM
4503	Martin and Karan	2015-01-16	Not an ABM
4557	Wren	2015-03-04	Not an ABM
4571	Voronovitsky	2015-03-11	Proprietary software
4606	Teran et al.	2015-04-27	No source code
4609	Barton et al.	2015-05-04	Source code readability
4727	Bell	2015-09-26	Proprietary software
4746	Garcia-Diaz	2015-10-16	Proprietary software
4780	Thron	2015-11-06	Proprietary software
4808	Diarisso et al.	2015-11-23	Source code readability
4850	Thron	2016-01-01	Proprietary software

Appendix C: OpenABM Review Data

The following pages present the full results of the systematic review of OpenABM models.

A note on interpreting the data: for brevity, only the first author's name is included. Columns with "T" and "F" values are binary; these represent true and false. For the "Endogenous" column, "NE" means "not endogenous", "E" means "endogenous", and "B" means both are present. For uniformity, "U" indicates uniform activation, "NU" the opposite, and "B" indicates that both options are available. For updating, "A" and "S" correspond to asynchronous and synchronous updating, respectively, while "PA" indicates partial asynchrony, "SS" the semi-synchronous state described in Appendix C, and "F" indicates that the modeling platform parallelizes using field partitioning. For selection, as in Figure 3.3, "F" means fixed selection, "R" means random selection, and "P" means Poisson selection; multiple letters indicate multiple activation regimes.

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2216	Bergin	2008-01-25	Repast	T	F	U	A	T	NE	F
2218	Janssen	2012-08-19	NetLogo	T	R	U	A	T	NE	F
2219	Janssen	2010-10-15	NetLogo	T	R	NU	A	T	NE	F
2220	Barton	2008-04-27	NetLogo	T	R	U	A	T	NE	F
2221	Janssen	2008-06-16	NetLogo	T	R	U	P	T	NE	F
2222	Janssen	2008-06-18	NetLogo	T	R	U	A	T	NE	F
2223	Janssen	2008-09-03	NetLogo	T	R	U	A	T	NE	F
2224	Barton	2008-11-22	NetLogo	F	R	U	A	T	NE	F
2225	Barton	2008-11-26	NetLogo	F	R	U	A	T	NE	F
2226	Rollins	2010-01-22	NetLogo	T	R	U	P	T	NE	F
2227	Zhong	2008-12-20	NetLogo	T	R	U	P	T	NE	F
2228	Ouyang	2008-12-15	NetLogo	T	R	U	A	T	NE	F
2229	Stotts	2008-12-15	NetLogo	T	R	U	A	T	NE	F
2230	Cherif	2008-12-15	NetLogo	T	R	U	A	T	NE	F
2232	Salau	2008-12-16	NetLogo	T	R	U	A	T	NE	F
2233	Salau	2008-12-16	NetLogo	T	R	U	P	T	NE	F
2234	Tovinen	2008-12-16	NetLogo	T	R	U	A	T	NE	F
2242	Genovese	2009-06-23	NetLogo	F	R	U	SS	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2245	Radtke	2009-08-31	MASON	T	F	U	A	T	NE	F
2247	Will	2009-08-29	FORTRAN	F	F	U	P	T	NE	F
2249	Poza	2009-09-24	NetLogo	F	R	U	A	T	NE	F
2250	Koch	2009-09-24	NetLogo	F	R	U	P	T	NE	F
2251	Radax	2013-05-02	Repast	F	F	U	B	T	NE	F
2252	Koch	2009-10-28	NetLogo	F	R	U	P	T	NE	F
2253	Dwyer	2009-11-28	NetLogo	F	R	U	A	T	NE	F
2255	Heckbert	2009-12-04	NetLogo	F	R	U	A	T	NE	F
2256	Heckbert	2009-12-04	NetLogo	F	R	U	A	T	NE	F
2257	Zhang	2009-12-07	NetLogo	F	R	U	A	T	NE	F
2258	Janssen	2010-05-04	NetLogo	T	R	U	A	T	NE	F
2259	EconGame	2010-01-26	Java	F	F	U	A	T	NE	F
2260	Delre	2010-02-11	C/C++	F	F	U	A	T	NE	F
2261	Gilbert	2010-03-03	NetLogo	F	R	U	A	T	NE	F
2262	van	2010-03-10	NetLogo	F	R	NU	A	T	NE	F
2264	Janssen	2010-05-16	NetLogo	T	R	U	A	T	NE	F
2265	Janssen	2010-05-16	NetLogo	T	R	U	A	T	NE	F
2267	Janssen	2010-06-04	NetLogo	T	R	U	A	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2268	Lettieri	2010-06-23	NetLogo	F	R	U	A	T	NE	F
2269	Savarimuthu	2010-06-28	Java	F	F	U	A	T	NE	F
2271	Aktipis	2010-07-17	NetLogo	T	R	U	SS	T	NE	F
2273	Zappala	2010-08-11	MASON	F	F	U	A	T	NE	F
2274	Janssen	2011-10-25	NetLogo	F	R	NU	A	T	NE	F
2275	Garcia	2010-09-22	NetLogo	F	R	U	A	T	NE	F
2276	Kochanski	2010-10-21	NetLogo	F	R	U	P	T	NE	F
2277	Janssen	2010-10-22	NetLogo	T	R	NU	A	T	NE	F
2278	Rebaudo	2010-10-26	NetLogo	F	R	U	P	T	NE	F
2279	Kim	2010-11-07	NetLogo	F	R	U	P	T	NE	F
2283	Schindler	2010-11-30	NetLogo	F	R	U	A	T	NE	F
2284	Janssen	2010-12-02	NetLogo	F	R	U	P	T	NE	F
2285	Haghnevis	2010-12-06	NetLogo	T	R	U	A	T	NE	F
2286	Peters	2010-12-13	NetLogo	T	R	U	A	T	NE	F
2288	Cegielski	2010-12-13	NetLogo	T	R	U	A	T	NE	F
2289	Shanafelt	2010-12-13	NetLogo	T	R	U	P	T	NE	F
2290	Haghnevis	2010-12-13	NetLogo	T	R	U	A	T	NE	F
2291	Weisbrod	2010-12-13	NetLogo	T	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2292	Barton	2010-12-13	NetLogo	T	R	U	SS	T	NE	F
2293	Wang	2010-12-14	Java	F	F	U	A	T	NE	F
2294	Bravo	2010-12-16	Multiple	F	FR	U	A	T	NE	F
2296	Gilbert	2010-12-31	NetLogo	F	R	U	P	T	NE	F
2297	Bergin	2013-02-13	NetLogo	T	R	U	P	T	NE	F
2310	Wijermans	2012-09-07	Repast	F	F	U	A	T	NE	F
2312	Balbi	2012-12-09	NetLogo	F	R	U	S	T	NE	F
2313	Kochanski	2011-02-13	NetLogo	F	R	U	P	T	NE	F
2461	Valbuena	2011-03-25	NetLogo	T	R	U	A	T	NE	F
2470	Watts	2011-03-14	NetLogo	F	R	U	P	T	NE	F
2475	Fatemi	2011-03-15	JADEX	F	R	U	A	T	NE	F
2483	Garcia	2011-03-28	NetLogo	F	R	U	A	T	NE	F
2516	Kim	2011-05-12	NetLogo	F	R	U	P	T	NE	F
2518	ipem	2011-05-27	C/C++	F	R	U	A	T	NE	F
2522	Quesada	2011-05-18	NetLogo	F	R	U	A	T	NE	F
2539	Nolan	2011-06-21	NetLogo	F	R	U	SS	T	NE	F
2549	Holtz	2011-06-30	Repast	T	R	U	A	T	NE	F
2552	Kahl	2012-12-06	NetLogo	T	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2554	Kahl	2012-03-27	NetLogo	T	R	U	P	T	NE	F
2558	Wang	2011-11-17	Java	F	F	U	P	T	NE	F
2587	Janssen	2011-08-14	NetLogo	T	R	U	A	T	NE	F
2592	Rixin	2011-08-08	NetLogo	F	R	U	A	T	NE	F
2606	Salau	2011-08-29	NetLogo	T	R	U	A	T	NE	F
2609	Rixin	2011-09-19	NetLogo	T	R	U	P	T	NE	F
2613	Levinson	2011-08-29	NetLogo	F	R	U	A	T	NE	F
2620	Nardin	2011-08-30	NetLogo	F	R	U	A	T	NE	F
2626	Murphy	2011-08-31	Java	F	R	U	S	T	NE	F
2634	Delre	2011-10-09	C/C++	F	F	U	A	T	NE	F
2639	Barton	2012-01-09	NetLogo	T	R	U	A	T	NE	F
2648	Barton	2011-11-14	NetLogo	T	R	U	A	T	NE	F
2659	Lawson	2011-10-06	NetLogo	F	R	U	S	T	NE	F
2661	Dixon	2011-10-07	NetLogo	F	FR	U	A	T	NE	F
2682	Rixin	2011-10-19	NetLogo	F	R	U	A	T	NE	F
2702	Delre	2011-10-24	C/C++	F	F	U	A	T	NE	F
2724	Baggio	2011-11-10	NetLogo	T	R	U	P	T	NE	F
2746	Xianyu	2011-11-16	Repast	F	F	U	A	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
2762	Knittel	2011-11-04	Repast	F	F	U	A	T	NE	F
2789	Watkins	2012-12-02	NetLogo	T	R	U	A	T	NE	F
2791	Radtke	2011-12-30	MASON	T	F	U	S	T	NE	F
2831	Baggio	2013-02-15	NetLogo	T	R	U	P	T	NE	F
2924	Lawson	2012-03-16	NetLogo	F	R	U	A	T	NE	F
2934	Wang	2012-03-20	Java	F	F	U	P	T	NE	F
2942	Hartshorn	2012-03-29	Java	F	F	U	P	T	NE	F
2986	Damaceanu	2012-05-03	NetLogo	F	R	U	A	T	NE	F
2988	Damaceanu	2012-05-03	NetLogo	F	R	U	A	T	NE	F
2999	Kasmire	2012-05-09	NetLogo	F	R	U	P	T	NE	F
3004	Vallino	2012-05-14	NetLogo	F	R	U	A	T	NE	F
3045	Millington	2012-07-15	NetLogo	T	R	U	A	T	NE	F
3051	Schindler	2012-06-29	NetLogo	F	R	U	P	T	NE	F
3063	Heckbert	2012-09-28	NetLogo	T	R	U	P	T	NE	F
3073	Janssen	2012-07-22	NetLogo	T	R	U	A	T	NE	F
3081	Nunes	2012-08-02	NetLogo	F	R	U	S	T	NE	F
3105	Barton	2012-09-18	NetLogo	T	R	U	A	T	NE	F
3119	Schindler	2012-08-18	NetLogo	F	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
3125	Holzhauer	2012-08-22	Repast	F	F	U	B	T	NE	F
3137	Rebaudo	2012-09-20	NetLogo	T	R	U	A	T	NE	F
3145	Squazzoni	2012-09-05	NetLogo	F	R	U	A	T	NE	F
3154	Tamburino	2012-08-07	NetLogo	T	R	U	A	T	NE	F
3163	Bergin	2012-09-14	NetLogo	T	R	NU	A	T	NE	F
3168	Millington	2012-09-28	NetLogo	T	R	U	A	T	NE	F
3175	Ibarra	2013-01-15	NetLogo	T	R	U	P	T	NE	F
3241	Baggio	2012-10-01	NetLogo	T	R	U	P	T	NE	F
3257	White	2012-10-09	Repast	F	R	U	A	T	NE	F
3274	Schindler	2012-11-05	NetLogo	F	R	U	P	T	NE	F
3294	Knoeri	2012-10-21	NetLogo	T	R	U	A	T	NE	F
3352	Brown	2012-11-30	Repast	F	R	U	A	T	NE	F
3359	Gravel-Miguel	2012-11-01	NetLogo	T	R	U	A	T	NE	F
3361	Lim	2012-11-10	NetLogo	F	R	NU	A	T	NE	F
3364	Millington	2012-11-02	NetLogo	T	R	U	A	T	NE	F
3366	Kim	2012-11-03	NetLogo	T	R	NU	A	T	NE	F
3372	Sarkar	2012-11-02	NetLogo	T	R	U	S	T	NE	F
3377	Udiani	2012-11-03	NetLogo	T	R	U	A	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
3391	Jansson	2012-11-10	Java	F	R	U	A	T	NE	F
3418	Dykstra	2012-11-28	NetLogo	F	R	U	A	T	NE	F
3420	Biondo	2012-11-29	NetLogo	F	R	U	A	T	NE	F
3549	Silverman	2012-12-20	Repast	F	R	U	A	T	NE	F
3551	Millington	2012-12-21	NetLogo	F	R	U	A	T	NE	F
3554	Millington	2012-12-21	NetLogo	F	R	U	P	T	NE	F
3556	Millington	2012-12-21	NetLogo	F	R	NU	A	T	NE	F
3575	Shiba	2013-01-09	NetLogo	F	R	U	A	T	NE	F
3577	Shiba	2013-01-09	NetLogo	F	R	U	A	T	NE	F
3580	Ibarra	2013-02-18	NetLogo	T	R	U	A	T	NE	F
3582	Premo	2013-01-09	NetLogo	F	R	U	A	T	NE	F
3597	Fernandez	2013-01-22	Repast	F	R	U	A	T	NE	F
3626	Smaldino	2013-02-08	MASON	F	R	U	A	T	NE	F
3640	Zvoleff	2013-02-23	Python	F	F	U	A	T	NE	F
3679	Dennehy	2013-03-11	NetLogo	T	R	U	P	T	NE	F
3695	Maroulis	2013-03-22	NetLogo	F	R	U	SS	T	NE	F
3705	Rodriguez	2013-03-26	NetLogo	T	R	U	A	T	NE	F
3708	Bravo	2013-03-27	NetLogo	T	R	U	A	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
3796	Nebel	2013-05-20	Repast	F	R	U	A	T	NE	F
3813	Klabunde	2013-05-29	NetLogo	T	R	U	P	T	NE	F
3817	Holtz	2013-06-04	Repast	F	R	U	A	T	NE	F
3819	Maroulis	2013-06-04	NetLogo	F	R	U	P	T	NE	F
3824	Kasmire	2013-06-07	NetLogo	F	R	U	A	T	NE	F
3826	Barton	2013-06-12	NetLogo	T	R	U	A	T	NE	F
3840	Fioretti	2013-06-22	NetLogo	F	R	U	A	T	NE	F
3842	Stoica	2013-06-23	C/C++	F	F	U	A	T	NE	F
3846	Wren	2013-06-24	NetLogo	F	R	U	A	T	NE	F
3851	Doubleday	2013-07-01	Java	T	R	U	A	T	NE	F
3854	Baggio	2013-07-02	NetLogo	T	R	U	A	T	NE	F
3860	Boyle	2013-07-06	C/C++	T	R	U	A	T	NE	F
3867	Shaffer	2013-07-11	NetLogo	F	R	U	A	T	NE	F
3872	Bert	2013-07-16	Repast	T	R	U	A	T	NE	F
3887	Janssen	2013-07-31	NetLogo	T	R	U	A	T	NE	F
3890	Kahn	2013-08-05	NetLogo	F	R	U	S	T	NE	F
3893	Klabunde	2013-08-07	NetLogo	T	R	U	A	T	NE	F
3902	Janssen	2013-08-13	NetLogo	T	R	U	A	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
3918	Lawson	2013-08-23	NetLogo	F	R	U	S	T	NE	F
3920	Hayes	2013-08-26	NetLogo	F	R	U	P	T	NE	F
3934	Janssen	2013-09-05	NetLogo	T	R	U	A	T	NE	F
3942	Sun	2013-09-10	Repast	F	F	U	A	T	NE	F
3946	Dobbie	2013-09-15	NetLogo	F	R	U	S	T	NE	F
3949	Barton	2013-09-17	NetLogo	F	R	U	A	T	NE	F
3957	Janssen	2013-09-30	NetLogo	T	R	U	A	T	NE	F
3960	Gooding	2013-10-01	NetLogo	F	R	U	A	T	NE	F
3967	Frantz	2013-10-08	MASON	F	R	U	A	T	NE	F
3976	Ibarra	2013-10-17	NetLogo	T	R	U	A	T	NE	F
3978	White	2013-10-17	Repast	F	FR	B	A	T	NE	F
3987	Daloglu	2013-10-21	Repast	T	R	U	A	T	NE	F
3993	Hayes	2013-10-24	NetLogo	F	R	U	P	T	NE	F
4015	Garcia-Diaz	2013-10-13	NetLogo	F	R	U	A	T	NE	F
4031	Galic	2013-11-12	NetLogo	T	R	U	A	T	NE	F
4039	Rubio-Campillo	2013-11-20	C/C++	T	R	U	F	F	NE	T
4042	Boone	2013-11-21	NetLogo	T	R	U	P	T	NE	F
4048	Markisic	2013-12-01	Repast	F	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
4050	Janssen	2013-12-01	NetLogo	T	R	U	A	T	NE	F
4055	Zellner	2013-12-05	NetLogo	F	R	U	A	T	NE	F
4061	Natalini	2013-12-08	NetLogo	T	R	U	A	T	NE	F
4063	Barcelo	2013-12-09	NetLogo	T	R	U	P	T	NE	F
4065	Zellner	2013-12-09	NetLogo	F	R	U	A	T	NE	F
4084	Cardona	2014-01-09	Python	F	F	U	P	T	NE	F
4087	White	2014-01-13	Repast	F	F	U	A	T	NE	F
4093	Bell	2014-01-23	Repast	T	F	U	A	T	NE	F
4110	Waldherr	2014-02-11	NetLogo	T	R	U	P	T	NE	F
4112	Sie	2014-02-11	NetLogo	F	F	U	A	T	NE	F
4122	Nakai	2014-02-16	C/C++	F	R	NU	A	T	NE	F
4128	Rebaudo	2014-02-25	NetLogo	T	R	U	A	T	NE	F
4144	Bellaubi	2014-03-08	NetLogo	T	F	U	A	T	NE	F
4154	Alizadeh	2014-03-14	Python	F	R	NU	A	T	NE	F
4180	Ruedin	2014-04-12	NetLogo	T	R	U	S	T	NE	F
4184	Hintze	2014-04-14	NetLogo	F	R	U	A	T	NE	F
4187	Janmaat	2014-04-17	NetLogo	F	R	U	A	T	NE	F
4204	Edmonds	2014-05-04	NetLogo	T	R	U	S	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
4216	Grow	2014-05-18	NetLogo	T	R	NU	A	T	NE	F
4222	Nisar	2014-05-26	NetLogo	T	R	U	A	T	NE	F
4224	Nisar	2014-05-26	NetLogo	T	R	U	A	T	NE	F
4234	Pereda	2014-06-12	NetLogo	F	R	U	A	T	NE	F
4242	Moritz	2014-06-19	NetLogo	T	R	U	P	T	NE	F
4273	Piou	2014-07-23	NetLogo	T	R	U	P	T	NE	F
4276	Schwarz	2014-07-25	Repast	T	F	U	A	T	NE	F
4293	Rasch	2014-08-02	Repast	T	F	U	A	T	NE	F
4298	Ligmann-Zielinska	2014-08-06	Python	F	F	U	S	T	NE	F
4316	Alizadeh	2014-08-24	Python	F	R	NU	A	T	NE	F
4327	Alizadeh	2014-09-09	Python	F	RP	B	A	T	B	F
4338	Castilla-Rho	2014-09-18	NetLogo	T	R	U	S	T	NE	F
4347	Brughmans	2014-09-25	NetLogo	T	R	U	P	T	NE	F
4368	Edmonds	2014-10-13	NetLogo	T	R	U	A	T	NE	F
4377	Zhao	2014-10-19	NetLogo	T	R	U	A	T	NE	F
4385	Watts	2014-10-25	NetLogo	T	R	U	P	T	NE	F
4396	Malik	2014-10-30	NetLogo	T	R	U	P	T	NE	F
4409	Atwater	2014-11-04	NetLogo	T	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
4411	Meyer	2014-11-05	NetLogo	T	R	U	A	T	NE	F
4428	Gooding	2014-11-26	NetLogo	F	R	U	A	T	NE	F
4433	Kasmire	2014-12-02	NetLogo	F	R	U	S	T	NE	F
4435	Kasmire	2014-12-02	NetLogo	F	R	U	P	T	NE	F
4437	Kasmire	2014-12-02	NetLogo	F	R	U	P	T	NE	F
4439	Kasmire	2014-12-02	NetLogo	F	R	U	P	T	NE	F
4441	Kasmire	2014-12-02	NetLogo	F	R	U	P	T	NE	F
4443	Kasmire	2014-12-02	NetLogo	F	R	U	P	T	NE	F
4447	Bergin	2014-12-11	NetLogo	T	R	U	A	T	NE	F
4458	Wang	2014-12-07	Java	F	F	U	A	F	NE	T
4464	Combs	2015-01-05	Repast	F	R	U	A	T	NE	F
4466	Ozik	2015-01-05	Repast	F	R	U	A	T	NE	F
4475	Yavas	2015-01-08	NetLogo	F	R	U	A	T	NE	F
4510	Shutters	2015-01-20	NetLogo	T	R	U	P	T	NE	F
4520	Dzutsati	2015-01-30	NetLogo	T	R	U	A	T	NE	F
4525	Boyle	2015-01-31	NetLogo	F	F	U	A	T	E	F
4536	Badham	2015-02-10	NetLogo	F	R	U	A	T	NE	F
4545	Aktipis	2015-02-19	NetLogo	T	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
4552	Scalco	2015-02-24	NetLogo	F	R	U	A	T	NE	F
4561	Paige	2015-03-05	NetLogo	T	R	U	A	T	NE	F
4564	Schmid	2015-03-07	NetLogo	T	R	U	P	T	NE	F
4581	Sasaki	2015-03-16	NetLogo	T	R	U	A	T	NE	F
4589	Camus	2015-03-25	Java	T	R	U	A	T	NE	F
4615	Rorabaugh	2015-05-17	NetLogo	F	R	U	A	T	NE	F
4627	Waring	2015-06-10	NetLogo	T	R	U	P	T	NE	F
4637	Sissa	2015-06-16	NetLogo	T	R	U	S	T	NE	F
4661	Le	2015-07-13	NetLogo	F	R	U	S	T	NE	F
4686	Lafuerza	2015-07-30	C/C++	F	F	U	A	T	NE	F
4688	Watts	2015-08-01	NetLogo	F	R	U	A	T	NE	F
4693	Fachada	2015-08-08	NetLogo	T	R	U	A	T	NE	F
4696	Kasmire	2015-08-11	NetLogo	F	R	U	P	T	NE	F
4706	Crooks	2015-08-27	MASON	T	F	U	A	T	E	F
4710	Huff	2015-09-01	NetLogo	T	R	U	P	T	NE	F
4716	Secchi	2015-09-09	NetLogo	F	R	U	A	T	NE	F
4718	Bianchi	2015-09-10	NetLogo	F	R	U	A	T	NE	F
4724	Scott	2015-09-02	NetLogo	F	R	U	P	T	NE	F

#	Author	Date	Platform	ODD	Selection	Uniform	Updating	Reproducible	Endogenous	Parallel
4734	Lawson	2015-10-06	NetLogo	F	R	U	S	T	NE	F
4744	Hales	2015-10-16	NetLogo	F	R	U	A	T	NE	F
4756	Rouchier	2015-10-21	NetLogo	F	R	NU	A	T	NE	F
4760	Sasaki	2015-10-27	NetLogo	T	R	U	SS	T	NE	F
4771	Fachada	2015-10-31	Java	T	R	U	B	T	NE	T
4778	Vinai	2015-11-06	NetLogo	F	R	U	A	T	NE	F
4795	Snitker	2015-11-16	NetLogo	F	R	U	A	T	NE	F
4828	Czaczkes	2015-12-17	NetLogo	F	R	U	A	T	NE	F
4830	Czaczkes	2015-12-17	NetLogo	F	R	U	A	T	NE	F
4841	Sasaki	2015-12-21	NetLogo	F	R	U	A	T	NE	F
4843	Biondo	2015-12-22	NetLogo	F	R	U	A	T	NE	F
4871	Edmonds	2016-01-29	NetLogo	F	R	U	A	T	NE	F
4873	Adelberg	2016-01-29	NetLogo	F	R	NU	A	T	NE	F
4880	Angourakis	2016-02-03	NetLogo	T	R	U	A	T	NE	F
4892	Smarzhevskiy	2016-02-14	NetLogo	F	R	U	P	T	NE	F
4917	Anderson	2016-03-02	Repast	F	R	U	A	T	NE	F

Appendix D: Model Code

D.1 RNG.h

```
// Copyright 2015 <Stefan McCabe>
/*
Bilateral Exchange Model
Originally written by Rob Axtell
Extended by Stefan McCabe

12/11/2015
*/

#ifndef RNG_H_
#define RNG_H_
#endif  // RNG_H_

#include <random>
#include <fstream>

class RNG {
    unsigned int seed;
    std::mt19937 rng;

    unsigned long num_agents, num_com;
    double alpha_min, alpha_max, shock_min, shock_max,
       wealth_min, wealth_max;

 public:
    RNG (bool randSeed, unsigned int s, unsigned int numagents,
        unsigned int numcom, double shockmin, double shockmax,
       double minalpha, double maxalpha, double minwealth,
       double maxwealth);
    unsigned int GetSeed() { return seed; }
    void SetSeed(unsigned int s) {
        seed = s;
        rng.seed(s);
    }
    std::mt19937 GetGenerator() { return rng; }

    int ValueInRange(int min, int max);
    unsigned long ValueInRange(unsigned long min, unsigned long
        max);
```

```
    double ValueInRange(double min, double max);

    unsigned long RandomAgent() { return ValueInRange(0L,
       num_agents-1); }
    unsigned long RandomCommodity() { return ValueInRange(0L,
       num_com-1); }
    int RandomBinary() { return ValueInRange(0, 1); }
    double RandomShock() { return ValueInRange(shock_min,
       shock_max); }
    double RandomAlpha() { return ValueInRange(alpha_min,
       alpha_max); }
    double RandomWealth() { return ValueInRange(wealth_min,
       wealth_max); }
    double RandomDouble() { return ValueInRange(0.0, 1.0); }
};

typedef RNG *RNGptr;
```

D.2 RNG.cpp

```
// Copyright 2015 <Stefan McCabe>
/*
Bilateral Exchange Model
Originally written by Rob Axtell
Extended by Stefan McCabe

12/11/2015
*/

// The key mechanism here - the use of thread_local mt19337
   pointers -
// was suggested on Stack Exchange.
// https://stackoverflow.com/questions/21237905/how-do-i-
   generate-thread-safe-uniform-random-numbers

#include "./RNG.h"
#include <thread>
#define ELPP_THREAD_SAFE
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wundef"
#pragma GCC diagnostic ignored "-Wshadow"
#pragma GCC diagnostic ignored "-Wsign-conversion"
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wctor-dtor-privacy"
```

```
#include "./easylogging++.h"
#pragma GCC diagnostic pop

int RNG::ValueInRange(int min, int max) {
    static thread_local std::mt19937* generator = nullptr;
    if (!generator) generator = new std::mt19937(std::clock() +
        std::hash<std::thread::id>()(std::this_thread::get_id()
       ));
    std::uniform_int_distribution<int> distribution(min, max);
    return distribution(*generator);
}

unsigned long RNG::ValueInRange(unsigned long min, unsigned
   long max) {
    static thread_local std::mt19937* generator = nullptr;
    if (!generator) generator = new std::mt19937(std::clock() +
        std::hash<std::thread::id>()(std::this_thread::get_id()
       ));
    std::uniform_int_distribution<unsigned long> distribution(
       min, max);
    return distribution(*generator);
}

double RNG::ValueInRange(double min, double max) {
    static thread_local std::mt19937* generator = nullptr;
    if (!generator) generator = new std::mt19937(std::clock() +
        std::hash<std::thread::id>()(std::this_thread::get_id()
       ));
    std::uniform_real_distribution<double> distribution(min,
       max);
    return distribution(*generator);
}

RNG::RNG (bool randSeed, unsigned int s, unsigned int numagents
   , unsigned int numcom, double shockmin, double shockmax,
   double minalpha, double maxalpha, double minwealth, double
   maxwealth) {
        // Seed the random number generator.
    if (!randSeed) {
        std::random_device rd;
        seed = rd();
        rng.seed(seed);
        LOG(INFO) << "Using random seed " << seed;
    } else {
        seed = s;
```

```
        LOG(INFO) << "Using fixed seed " << seed;
        rng.seed(s);
    }

        alpha_min = minalpha;
        alpha_max = maxalpha;
        shock_min = shockmin;
        shock_max = shockmax;
        num_agents = static_cast<unsigned long>(numagents);
        num_com = static_cast<unsigned long>(numcom);
        wealth_min = minwealth;
        wealth_max = maxwealth;
}
```

D.3 main.h

```
// Copyright 2015 <Stefan McCabe>
/*
Bilateral Exchange Model
Originally written by Rob Axtell
Extended by Stefan McCabe

12/11/2015
*/

#ifndef MAIN_H_
#define MAIN_H_
#endif  // MAIN_H_

#include "./RNG.h"
#include "tbb/tbb.h"
#include "tbb/concurrent_vector.h"
#include <mutex>


// Global variables specifying model parameters. See parameters
   .cfg for documentation.
bool UseRandomSeed;
unsigned int NonRandomSeed;
double Version = 1.0;
unsigned int NumberOfAgents;
unsigned int NumberOfCommodities;
unsigned int PairwiseInteractionsPerPeriod;
double alphaMin;
```

```
double alphaMax;
unsigned int wealthMin;
unsigned int wealthMax;
bool DefaultSerialExecution;
int AgentsToRandomize;
int RandomizationMethod;
int RequestedEquilibrations;
bool SameAgentInitialCondition;
double trade_eps;
double exp_trade_eps;
int termination_criterion;
long long TerminationTime;
double termination_eps;
long long CheckTerminationThreshold;
int CheckTerminationPeriod;
bool ShockPreferences;
int ShockPeriod;
double MinShock;
double MaxShock;
bool debug;
bool PrintEndowments;
bool PrintIntermediateOutput;
int IntermediateOutputPrintPeriod;
bool PrintConvergenceStats;
bool PrintFinalCommodityList;
int activationMethod;
const char* outputFilename = NULL;
bool fileAppend;
bool writeToFile;
std::ofstream outfile;
int NumberOfThreads;
bool ForkAndJoin;
bool ShuffleAfterJoin;
bool DumpAgentInformation;

RNGptr Rand;


typedef tbb::concurrent_vector<double> CommodityArray;
typedef CommodityArray *CommodityArrayPtr;

// Functions
double inline Dot(CommodityArrayPtr vector1, CommodityArrayPtr
   vector2);
void ReadConfigFile(std::string file);
```

```
void InitMiscellaneous();
void OpenFile(const char * filename);
void WriteHeader();
void WriteLine();

// Classes and methods
class MemoryObject {
    long long start;

 public:
    MemoryObject();
    void WriteMemoryRequirements();
} MemoryState;


class Data {
    int N;
    double min;
    double max;
    double sum;
    double sum2;
 public:
    Data();
    void Init();
    void AddDatum(double Datum);
    int GetN() { return N; }
    double GetMin() { return min; }
    double GetMax() { return max; }
    double GetDelta() { return max - min; }
    double GetAverage() {
        if (N > 0) {
            return sum/N;
        } else {
            return 0.0;
        }}
        double GetExpAverage() { return exp(GetAverage()); }
        double GetVariance();
        double GetStdDev() { return sqrt(GetVariance()); }
};

typedef Data *DataPtr;

class CommodityData {
    tbb::concurrent_vector<Data> data;
```

```
 public:
    CommodityData();
    void Clear();
    DataPtr GetData(size_t index) { return &data[index]; }
    double L2StdDev();
    double LinfStdDev();
};

class Agent {
    size_t id;

    CommodityArray alphas;
    CommodityArray endowment;
    CommodityArray initialMRSs;
    CommodityArray allocation;
    CommodityArray currentMRSs;

    Agent();
    double initialUtility;
    double initialWealth;
    double lambda;  // for Poisson activation
    double nextTime;  // for Poisson activation
    double initiative; // exogenous Poisson variable,
       distributed U(0,1)
    long long interactions = 0;
    long long activations = 0;
 public:
    explicit Agent(int size, size_t x);
    void Init();
    void Reset();


    std::mutex m;
    void MarkActivated() {
        //std::lock_guard<std::mutex> lock(m);
        activations++;
    }
    long long GetNumberOfActivations() {
        return activations;
    }
    void MarkSuccessfulTrade() {
        interactions++;
    }
    long long GetNumberOfTrades() {
        return interactions;
```

```
}
size_t GetId() {
    return id;
}
double GetAlpha(size_t CommodityIndex) {
    //std::lock_guard<std::mutex> lock(m);
    return alphas[CommodityIndex];
}
void SetAlpha(size_t CommodityIndex, double alpha) {
    //std::lock_guard<std::mutex> lock(m);
    alphas[CommodityIndex] = alpha;
}
void SetLambda(double lam) {
    //std::lock_guard<std::mutex> lock(m);
    lambda = lam;
}
double GetLambda() {
    //std::lock_guard<std::mutex> lock(m);
    return lambda;
}
void SetNextTime(double nextT) {
    //std::lock_guard<std::mutex> lock(m);
    nextTime = nextT;
}
double GetNextTime() {
    //std::lock_guard<std::mutex> lock(m);
    return nextTime;
}
double GetEndowment(size_t CommodityIndex) {
    //std::lock_guard<std::mutex> lock(m);
    return endowment[CommodityIndex];
}
double MRS(size_t CommodityIndex, size_t Numeraire);
void ComputeMRSs();
double GetInitialMRS(size_t CommodityIndex) {
    //std::lock_guard<std::mutex> lock(m);
    return initialMRSs[CommodityIndex];
}
double GetAllocation(size_t CommodityIndex) {
    //std::lock_guard<std::mutex> lock(m);
    return allocation[CommodityIndex];
}
void IncreaseAllocation(size_t CommodityIndex, double
   amount) {
    //std::lock_guard<std::mutex> lock(m);
```

```
        allocation[CommodityIndex] += amount;
    }
    double GetCurrentMRS(size_t CommodityIndex) {
        //std::lock_guard<std::mutex> lock(m);
        return currentMRSs[CommodityIndex];
    }
    CommodityArrayPtr GetCurrentMRSs() {
        //std::lock_guard<std::mutex> lock(m);
        return &currentMRSs;
    }
    double Utility();
    double GetInitialUtility() {
        //std::lock_guard<std::mutex> lock(m);
        return initialUtility;
    }
    double Wealth(CommodityArrayPtr prices) {
        //std::lock_guard<std::mutex> lock(m);
        return Dot(&allocation, prices);
    }
    double GetInitialWealth() {
        //std::lock_guard<std::mutex> lock(m);
        return initialWealth;
    }
    double GetInitiative() {
        return initiative;
    }
};

typedef std::shared_ptr<Agent> AgentPtr;

class AgentPopulation {
    //tbb::concurrent_vector<AgentPtr> Agents;
    tbb::concurrent_vector<AgentPtr> Agents;
    tbb::concurrent_vector<size_t> AgentIndices;
    tbb::concurrent_vector<std::pair<double,AgentPtr>>
       PoissonActivations;
    size_t AgentIndex = 0;
    bool PoissonUpToDate;
    Data InitialOwnWealthData;
    CommodityArray Volume;
    CommodityData AlphaData, EndowmentData, LnMRSsData;
    bool LnMRSsDataUpToDate;
    tbb::concurrent_vector<std::tuple<long long, double, size_t
       , double, size_t>> results;
    void ComputeLnMRSsDistribution();
```

```
double LastSumOfUtilities;
double ComputeSumOfUtilities();
double ComputeIncreaseInSumOfUtilities() {
    return ComputeSumOfUtilities() - LastSumOfUtilities;
}
double ComputeRelativeIncreaseInSumOfUtilities() {
    return ComputeIncreaseInSumOfUtilities() /
       LastSumOfUtilities;
}

void GetRandomAgentPair(AgentPtr& Agent1, AgentPtr& Agent2)
   ;
void GetUniformAgentPair(AgentPtr& Agent1, AgentPtr& Agent2
   );
void GetFixedAgentPair(AgentPtr& Agent1, AgentPtr& Agent2);
void GetPoissonAgentPair(AgentPtr& Agent1, AgentPtr& Agent2
   );
void SetPoissonAgentDistribution();
void Trade(AgentPtr a1, AgentPtr a2);

void(AgentPopulation::*GetAgentPair) (AgentPtr& Agent1,
   AgentPtr& Agent2);

std::tuple<long long, double, size_t, double, size_t>
   ParallelTrade (AgentPtr a1, AgentPtr a2);
void TradeInFork (tbb::concurrent_vector<AgentPtr> a);

void(AgentPopulation::*GetAgentPairInFork) (AgentPtr&
   Agent1, AgentPtr& Agent2, tbb::concurrent_vector<
   AgentPtr> a, tbb::concurrent_vector<AgentPtr>::iterator
   &ait);
void GetRandomAgentPairInFork(AgentPtr& Agent1, AgentPtr&
   Agent2, tbb::concurrent_vector<AgentPtr> a, tbb::
   concurrent_vector<AgentPtr>::iterator &ait);
void GetUniformAgentPairInFork(AgentPtr& Agent1, AgentPtr&
   Agent2, tbb::concurrent_vector<AgentPtr> a, tbb::
   concurrent_vector<AgentPtr>::iterator &ait);
void GetFixedAgentPairInFork(AgentPtr& Agent1, AgentPtr&
   Agent2, tbb::concurrent_vector<AgentPtr> a, tbb::
   concurrent_vector<AgentPtr>::iterator &ait);
void GetPoissonAgentPairInFork(AgentPtr& Agent1, AgentPtr&
   Agent2, tbb::concurrent_vector<AgentPtr> a, tbb::
   concurrent_vector<AgentPtr>::iterator &ait);
```

```
    void SetPoissonAgentDistributionInFork(tbb::
       concurrent_vector<AgentPtr> a, tbb::concurrent_vector<
       AgentPtr>::iterator &ait);

    void IntermediateOutput();
    bool TestConvergence();
    void DumpAgentInfo();

 public:
    explicit AgentPopulation(int size);
    bool Converged;
    long long theTime;
    long long TotalInteractions;

    void IncreaseTotalInteractions(long long x) {
        TotalInteractions += x;
    }
    void Init();
    void Reset();
    long long Equilibrate(int NumberOfEquilibrationsSoFar);
    long long ParallelEquilibrate(int
       NumberOfEquilibrationsSoFar);
    long long ForkAndJoinEquilibrate(int
       NumberOfEquilibrationsSoFar);
    void ConvergenceStatistics(CommodityArray VolumeStats);
    void CompareTwoAgents(AgentPtr Agent1, AgentPtr Agent2);
    void ShockAgentPreferences();
    std::string WriteWealthInfo();
    std::string WriteUtilityInfo();
    void WriteLine();
    std::mutex m;
};

typedef AgentPopulation *AgentPopulationPtr;
```

D.4 main.cpp

```
// Copyright 2015 <Stefan McCabe>
/*
Bilateral Exchange Model
Originally written by Rob Axtell
Extended by Stefan McCabe

12/11/2015
```

```
*/



#include "./main.h"
#include "tbb/tbb.h"
#include "tbb/concurrent_vector.h"
#include <libconfig.h++>
#define ELPP_THREAD_SAFE
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wundef"
#pragma GCC diagnostic ignored "-Wshadow"
#pragma GCC diagnostic ignored "-Wsign-conversion"
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wctor-dtor-privacy"
#include "./easylogging++.h"
#pragma GCC diagnostic pop


// Initialize the logger. This should come immediately after
   the #includes are finished.
INITIALIZE_EASYLOGGINGPP


/*============
    ==Methods==
    ============*/
    double inline Dot(CommodityArrayPtr vector1,
       CommodityArrayPtr vector2) {
        double sum = 0.0;
        for (size_t i = 0; i < static_cast<size_t>(
           NumberOfCommodities); ++i) {
            sum += (*vector1)[i] * (*vector2)[i];
        }
        return sum;
    }

    MemoryObject::MemoryObject():
    start(0)
{}  //  MemoryObject::MemoryObject()


void MemoryObject::WriteMemoryRequirements() {
    if (debug) {
```

```
        LOG(DEBUG) << "Size of Agent in memory: " << sizeof(
           Agent) << " bytes";
        LOG(DEBUG) << "Size of Data in memory: " << sizeof(Data
           ) << " bytes";
        LOG(DEBUG) << "Size of CommodityData in memory: " <<
           sizeof(CommodityData) << " bytes";
        LOG(DEBUG) << "Size of AgentPopulation in memory: " <<
           sizeof(AgentPopulation) << " bytes";
        LOG(DEBUG) << "Total bytes required (approximate): " <<
            \
        (sizeof(Rand) + sizeof(MemoryObject) + sizeof(
           CommodityData) \
            + sizeof(AgentPopulation) + sizeof(Agent) *
               static_cast<unsigned long>(NumberOfAgents)) << "
                bytes";
    }
}

Data::Data():
N(0), min(1000000.0), max(0.0), sum(0.0), sum2(0.0)
{}  //  Data::Data()

void Data::Init() {
    N = 0;
    min = 1000000.0;    //  10^6
    max = 0.0;
    sum = 0.0;
    sum2 = 0.0;
}   //  Data::Data()

void Data::AddDatum(double Datum) {
    N = N + 1;
    if (Datum < min) {
        min = Datum;
    }
    if (Datum > max) {
        max = Datum;
    }
    sum += Datum;
    sum2 += Datum * Datum;
}   //  Data::AddDatum()

double Data::GetVariance() {
    double avg, arg;
```

```
    if (N > 1) {
        avg = GetAverage();
        arg = sum2 - N * avg * avg;
        return arg / (N - 1);
    } else {
        return 0.0;
    }
}   //  Data::GetVariance()

CommodityData::CommodityData() {
    // Constructor resizes the data vector to the appropriate
       size.
    // It is initially zero-initialized, which is obviously
       problematic.
    data.resize(static_cast<size_t>(NumberOfCommodities));
}

void CommodityData::Clear() {
    //  Initialize data objects
    //
    for (auto& commodity : data) {
        commodity.Init();
    }
}   //  CommodityData::Clear()

double CommodityData::L2StdDev() {
    double sum = 0.0;

    // Instead of computing the standard deviation for each
       commodity (and calling sqrt() N times),
    // find the largest variance and from it get the std dev
    for (auto& commodity : data) {
        sum += commodity.GetVariance();
    }
    return sqrt(sum);
}   //  CommodityData::L2StdDev

double  CommodityData::LinfStdDev() {
    double var, max = 0.0;

    // Instead of computing the standard deviation for each
       commodity (and calling sqrt() N times),
    // find the largest variance and from it get the std dev
    for (auto& commodity : data) {
        var = commodity.GetVariance();
```

```
        if (var > max) {
            max = var;
        }
    }
    return sqrt(max);
}   //  CommodityData::LinfStdDev


Agent::Agent(int size, size_t x): initialUtility(0.0),
   initialWealth(0.0) {
    // Constructor resizes the data vector to the appropriate
       size.
    // It is initially zero-initialized, which is obviously
       problematic.
    // This allows me to replace arrays with vectors.
    id = x;
    alphas.resize(static_cast<size_t>(size));
    endowment.resize(static_cast<size_t>(size));
    initialMRSs.resize(static_cast<size_t>(size));
    allocation.resize(static_cast<size_t>(size));
    currentMRSs.resize(static_cast<size_t>(size));
    initiative = Rand->ValueInRange(0.0, 1.0);
    Init();
}

void Agent::Init() {
    //std::lock_guard<std::mutex> lock(m);
    size_t CommodityIndex;

    //  First generate and normalize the exponents...
    double sum = 0.0;
    for (auto& alpha : alphas) {
        alpha = Rand->RandomAlpha();
        sum += alpha;
    }
    //  Next, fill up the rest of the agent fields...
    for (auto& alpha : alphas) {
        auto i = static_cast<size_t>(&alpha - &alphas[0]);
        alpha = alpha / sum;
        endowment[i] = Rand->RandomWealth();
        allocation[i] = endowment[i];
        initialMRSs[i] = MRS(i, 0);
        currentMRSs[i] = initialMRSs[i];
    }
    initialUtility = Utility();
```

```
    initialWealth = Wealth(&initialMRSs);


}   //  Agent:Init()

void Agent::Reset() {
    //std::lock_guard<std::mutex> lock(m);
    for (auto& currentAgentMRS : currentMRSs) {
        auto i = static_cast<size_t>(&currentAgentMRS - &
           currentMRSs[0]);
        allocation[i] = endowment[i];
        currentAgentMRS = initialMRSs[i];
    }
}   //  Agent::Reset

// MRS = marginal rate of substitution
double Agent::MRS(size_t CommodityIndex, size_t Numeraire) {
    //std::lock_guard<std::mutex> lock(m);
    return (alphas[CommodityIndex] * allocation[Numeraire]) / (
       alphas[Numeraire] * allocation[CommodityIndex]);
}   //  Agent::MRS()

void Agent::ComputeMRSs() {
    //std::lock_guard<std::mutex> lock(m);
    for (auto& currentAgentMRS : currentMRSs) {
        auto i = static_cast<size_t>(&currentAgentMRS - &
           currentMRSs[0]);
        if (i == 0) {
            currentAgentMRS = 1.0;
        } else {
            currentMRSs[i] = MRS(i, 0);
        }
    }
}   //      Agent::ComputeMRSs()

double Agent::Utility() {
    //std::lock_guard<std::mutex> lock(m);
    double product = 1.0;

    for (size_t i = 0; i < allocation.size(); ++i) {
        product *= pow(allocation[i], alphas[i]);
    }

    return product;
}   //  Agent::Utility()
```

```
void AgentPopulation::ComputeLnMRSsDistribution() {
    LnMRSsData.Clear();

    for (auto& agent : Agents) {
        agent->ComputeMRSs();
        for (size_t j = 0; j < static_cast<size_t>(
           NumberOfCommodities); ++j) {
            LnMRSsData.GetData(j)->AddDatum(log(agent->
               GetCurrentMRS(j)));
        }
    }       //  for i...
    LnMRSsDataUpToDate = true;
}   //  AgentPopulation::ComputeLnMRSsDistribution()

std::string AgentPopulation::WriteUtilityInfo() { //TODO: This
   is basically a Data()
    double sum = 0.0;
    double min = 100000.0;
    double max = 0;
    double avg = 0.0;
    double x = 0.0;
    std::stringstream s;

    for (auto& agent : Agents) {
        double utility = agent->Utility();
        sum += utility;
        min = (utility < min) ? utility : min;
        max = (utility > max) ? utility : max;
    }
    avg = sum / static_cast<double>(NumberOfAgents);


    for (auto& agent : Agents) {
        x += (avg - agent->Utility()) * (avg - agent->Utility()
           );
    }

    double sd = std::sqrt(x/static_cast<double>(NumberOfAgents)
       );


    s << min << "," << max << "," << avg << "," << sd << ",";
    return s.str();
}
```

```
std::string AgentPopulation::WriteWealthInfo() { //TODO: This
   is basically a Data()
    double sum = 0.0;
    double min = 100000.0;
    double max = 0;
    double avg = 0.0;
    double x = 0.0;
    std::stringstream s;

    for (auto& agent : Agents) {
        double w = agent->Wealth(agent->GetCurrentMRSs());
        sum += w;
        min = (w < min) ? w : min;
        max = (w > max) ? w : max;
    }
    avg = sum / static_cast<double>(NumberOfAgents);

    for(auto& agent : Agents) {
        x += (avg - agent->Wealth(agent->GetCurrentMRSs())) * (
           avg - agent->Wealth(agent->GetCurrentMRSs()));
    }

    double sd = std::sqrt(x/static_cast<double>(NumberOfAgents)
       );

    s << min << "," << max << "," << avg << "," << sd << ",";
    return s.str();
}

double AgentPopulation::ComputeSumOfUtilities() {
    double sum = 0.0;

    for (auto& agent : Agents) {
        sum += agent->Utility();
    }
    return sum;
}   //  AgentPopulation::ComputeSumOfUtilities()

void AgentPopulation::GetRandomAgentPair(AgentPtr& Agent1,
   AgentPtr& Agent2) {
    Agent1 = Agents[Rand->RandomAgent()];
    do {
        Agent2 = Agents[Rand->RandomAgent()];
    } while (Agent2 == Agent1);
```

```
}   //  AgentPopulation::GetRandomAgentPair()

void AgentPopulation::GetUniformAgentPair(AgentPtr& Agent1,
   AgentPtr& Agent2) {
    if (NumberOfAgents % 2 > 0 && AgentIndex == 0) {
        LOG(WARNING) << "Warning: Uniform activation requires
           an even number of agents.";
    }
    std::lock_guard<std::mutex> lock(m);
    Agent1 = Agents[AgentIndices[AgentIndex++]];
    Agent2 = Agents[AgentIndices[AgentIndex++]];
    if (AgentIndex >= static_cast<size_t>(NumberOfAgents)) {
        // if (debug) { LOG(DEBUG) << "Rolling over uniform
           indices..."; }
        AgentIndex = 0;
        std::shuffle(AgentIndices.begin(), AgentIndices.end(),
           Rand->GetGenerator());
    }
}

void AgentPopulation::GetFixedAgentPair(AgentPtr& Agent1,
   AgentPtr& Agent2) {
    if (NumberOfAgents % 2 > 0 && AgentIndex == 0) {
        LOG(WARNING) << "Warning: Fixed activation requires an
           even number of agents.";
    }
    std::lock_guard<std::mutex> lock(m);
    Agent1 = Agents[AgentIndices[AgentIndex++]];
    Agent2 = Agents[AgentIndices[AgentIndex++]];
    if (AgentIndex >= static_cast<size_t>(NumberOfAgents)) {
        // if (debug) { LOG(DEBUG) << "Rolling over uniform
           indices..."; }
        AgentIndex = 0;
    }
}

void AgentPopulation::GetPoissonAgentPair(AgentPtr& Agent1,
   AgentPtr& Agent2) {
    std::lock_guard<std::mutex> lock(m);
    if (!PoissonUpToDate) {
        SetPoissonAgentDistribution();
    }
    size_t AgentIndex1 = AgentIndex++;
    size_t AgentIndex2 = AgentIndex;
```

```
    Agent1 = PoissonActivations[AgentIndex1].second;
    Agent2 = PoissonActivations[AgentIndex2].second;

    size_t swap = 1;
    while (Agent1 == Agent2 || Agent2 == nullptr) {
        ++swap; // get some value close to the "correct" one
        Agent2 = PoissonActivations[Rand->ValueInRange(
           static_cast<int>(AgentIndex2-swap),static_cast<int>(
           AgentIndex2+swap))].second;
    }
    ++AgentIndex;
    if (AgentIndex >= static_cast<size_t>(NumberOfAgents)) {
        PoissonUpToDate = false;
    }
    if (Agent1 == nullptr) {
        LOG(WARNING) << "null pointer in Agent1" << std::endl;
        std::terminate();
    }
    if (Agent2 == nullptr) {
        LOG(WARNING) << "null pointer in Agent2" << std::endl;
        std::terminate();
    }
    if (Agent1 == NULL) {
        LOG(WARNING) << "null pointer in Agent1" << std::endl;
        std::terminate();
    }
    if (Agent2 == NULL) {
        LOG(WARNING) << "null pointer in Agent2" << std::endl;
        std::terminate();
    }
}


void AgentPopulation::SetPoissonAgentDistribution() {
    // reset data structures
    PoissonActivations.clear();
    PoissonActivations.shrink_to_fit(); // memory leaks are bad
    tbb::concurrent_vector<std::pair<double,AgentPtr>>().swap(
       PoissonActivations);
    // more ritual to stave off memory leaks
    AgentIndex = 0;

    double totalWealth = 0.0;
    double totalDistanceFromMean = 0.0;
    double denom;
```

```
double totalLambda = 0.0;

//determine mean wealth
for (auto &a : Agents) {
    totalWealth += a->Wealth(a->GetCurrentMRSs());
}
double meanWealth = totalWealth / static_cast<double>(
   NumberOfAgents);

//determine total distance from the mean
for (auto &a : Agents) {
    double distFromMean = std::abs(a->Wealth(a->
       GetCurrentMRSs()) - meanWealth);
    totalDistanceFromMean += distFromMean;
}

//update lambdas based on distance from the mean
//this is where the Poisson activation methods are
   differentiated
for (auto &a : Agents) {
    double lam;
    switch (activationMethod) {
        case 2:  // furthest from mean activate faster
        //LOG(WARNING) << "WARNING: This Poisson method is
           still buggy.";
        denom = std::abs(a->Wealth(a->GetCurrentMRSs()) -
           meanWealth);
        if (denom == 0) {
            denom = 0.0001;
        }
        lam = totalDistanceFromMean / denom;
        a->SetLambda(lam);
        totalLambda += lam;
        break;
        case 3: // poor activate faster
        denom = a->Wealth(a->GetCurrentMRSs());
        if (denom == 0) {
            denom = 0.0001;
        }
        lam = 1 / denom;
        a->SetLambda(lam);
        totalLambda += lam;
        break;
        case 4: // rich activate faster
        denom = a->Wealth(a->GetCurrentMRSs());
```

```
            if (denom == 0) {
                denom = 0.0001;
            }
            lam = denom;
            a->SetLambda(lam);
            totalLambda += lam;
            break;
            case 5: // closest to mean activate faster
            lam = std::abs(a->Wealth(a->GetCurrentMRSs()) -
               meanWealth);
            a->SetLambda(lam);
            totalLambda += lam;
            break;
            default:
            LOG(WARNING) << "Error: Accessing Poisson
               activation out of scope";
            std::terminate();
        }

    }

    //normalize lambdas
    for (auto &a : Agents) {
        auto i = static_cast<size_t>(&a - &Agents[0]);
        double lam = a->GetLambda() * static_cast<double>(
           NumberOfAgents) * 1.25/totalLambda; // was 1.1
        if (lam == 0) {
            lam = 1.0 / static_cast<double>(NumberOfAgents);
        }
        //done normalizing lambdas

        //schedule agents based on lambda
        a->SetLambda(lam);
        a->SetNextTime(-1 * log(Rand->RandomDouble()) / a->
           GetLambda());
        while (a->GetNextTime() < 1 ) {
            PoissonActivations.push_back(std::make_pair(a->
               GetNextTime(), a)); // double-check the order
               here
            a->SetNextTime(a->GetNextTime() + -1 * log(Rand->
               RandomDouble()) / a->GetLambda());
        }
    }

    //sort the activations vector
```

```
    std::sort(PoissonActivations.begin(), PoissonActivations.
       end(), [](const std::pair<double, AgentPtr> &left, const
        std::pair<double, AgentPtr> &right) {
        return left.first < right.first;
    });

    PoissonUpToDate = true;
}

void AgentPopulation::GetRandomAgentPairInFork(AgentPtr& Agent1
   , AgentPtr& Agent2, tbb::concurrent_vector<AgentPtr> a, tbb
   ::concurrent_vector<AgentPtr>::iterator &ait) {
    auto s = a.size();
    Agent1 = a[Rand->ValueInRange(0L,s-1)];
    do {
        Agent2 = a[Rand->ValueInRange(0L,s-1)];
    } while (Agent2 == Agent1);
}

void AgentPopulation::GetUniformAgentPairInFork(AgentPtr&
   Agent1, AgentPtr& Agent2, tbb::concurrent_vector<AgentPtr> a
   , tbb::concurrent_vector<AgentPtr>::iterator &ait) {
    if (NumberOfAgents % 2 > 0 && AgentIndex == 0) {
        LOG(WARNING) << "Warning: Uniform activation requires
           an even number of agents.";
    }

    Agent1 = *ait;
    ait++;
    Agent2 = *ait;
    ait++;
    if (ait == a.end()) {
        ait == a.begin();
    }
}

void AgentPopulation::GetFixedAgentPairInFork(AgentPtr& Agent1,
    AgentPtr& Agent2, tbb::concurrent_vector<AgentPtr> a, tbb::
   concurrent_vector<AgentPtr>::iterator &ait) {
    if (NumberOfAgents % 2 > 0 && AgentIndex == 0) {
        LOG(WARNING) << "Warning: Fixed activation requires an
           even number of agents.";
    }

    Agent1 = *ait;
```

```
    ait++;
    Agent2 = *ait;
    ait++;
    if (ait == a.end()) {
        ait == a.begin();
    }
}

void AgentPopulation::GetPoissonAgentPairInFork(AgentPtr&
   Agent1, AgentPtr& Agent2, tbb::concurrent_vector<AgentPtr> a
   , tbb::concurrent_vector<AgentPtr>::iterator &ait) {
    LOG(WARNING) << "Warning: NYI";
    return;
}

void AgentPopulation::SetPoissonAgentDistributionInFork(tbb::
   concurrent_vector<AgentPtr> a, tbb::concurrent_vector<
   AgentPtr>::iterator &ait) {
    SetPoissonAgentDistribution();
}


AgentPopulation::AgentPopulation(int size):
AlphaData(), EndowmentData(), LnMRSsData(), LnMRSsDataUpToDate(
   true), LastSumOfUtilities(0.0), GetAgentPair(NULL) {
    Volume.resize(static_cast<size_t>(size));
    AgentIndices.resize(static_cast<size_t>(NumberOfAgents));
    for (size_t i = 0; i < AgentIndices.size(); ++i) {
        AgentIndices[i] = i;
    }

    if (activationMethod == 1) { std::shuffle(AgentIndices.
       begin(), AgentIndices.end(), Rand->GetGenerator()); }

    AgentIndex = 0;
    for (size_t i = 0; i < static_cast<size_t>(NumberOfAgents);
        ++i) {
        Agents.emplace_back(new Agent{NumberOfCommodities, i});
        //Agents.push_back(Agent{NumberOfCommodities, i});
    }

    PoissonUpToDate = false;
    switch (activationMethod) {
        case -1:
        GetAgentPair = &AgentPopulation::GetFixedAgentPair;
```

```
GetAgentPairInFork = &AgentPopulation::
   GetFixedAgentPairInFork;
LOG(INFO) << "Using fixed activation";
LOG(INFO) << "WARNING: Do not use fixed activation.";
break;
case 0:
GetAgentPair = &AgentPopulation::GetRandomAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetRandomAgentPairInFork;
LOG(INFO) << "Using random activation";
break;
case 1:
GetAgentPair = &AgentPopulation::GetUniformAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetUniformAgentPairInFork;
LOG(INFO) << "Using uniform activation";
break;
case 2:
GetAgentPair = &AgentPopulation::GetPoissonAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetPoissonAgentPairInFork;
LOG(INFO) << "Using Poisson activation (λ = 1/|wealth -
    mean(wealth)|)";
LOG(WARNING) << "WARNING: This Poisson method is still
   buggy.";
break;
case 3:
GetAgentPair = &AgentPopulation::GetPoissonAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetPoissonAgentPairInFork;
LOG(INFO) << "Using Poisson activation (λ = 1/wealth)";
break;
case 4:
GetAgentPair = &AgentPopulation::GetPoissonAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetPoissonAgentPairInFork;
LOG(INFO) << "Using Poisson activation (λ = wealth)";
break;
case 5:
GetAgentPair = &AgentPopulation::GetPoissonAgentPair;
GetAgentPairInFork = &AgentPopulation::
   GetPoissonAgentPairInFork;
LOG(INFO) << "Using Poisson activation (λ = |wealth -
   mean(wealth)|)";
break;
```

```
        default:
        LOG(ERROR) << "Invalid activation method (or NYI)";
        std::terminate();
        break;
    }
}   // Constructor...

void AgentPopulation::Init() {
    size_t CommodityIndex;


    AlphaData.Clear();
    EndowmentData.Clear();
    LnMRSsData.Clear();

    //  Cycle through the agents...
    for (auto& ActiveAgent : Agents) {
        ActiveAgent->Init();

        //  Next, fill up the rest of the agent fields...
        for (CommodityIndex = 0; CommodityIndex < static_cast<
           size_t>(NumberOfCommodities); ++CommodityIndex) {
            AlphaData.GetData(CommodityIndex)->AddDatum(
               ActiveAgent->GetAlpha(CommodityIndex));
            EndowmentData.GetData(CommodityIndex)->AddDatum(
               ActiveAgent->GetEndowment(CommodityIndex));
            LnMRSsData.GetData(CommodityIndex)->AddDatum(log(
               ActiveAgent->GetInitialMRS(CommodityIndex)));
        }   //  for (CommodityIndex...
        InitialOwnWealthData.AddDatum(ActiveAgent->
           GetInitialWealth());
    }   //  for (AgentIndex...

    LnMRSsDataUpToDate = true;
    LastSumOfUtilities = ComputeSumOfUtilities();

    //  Finally, display stats on the instantiated population
       ...
    if (PrintEndowments) {
        LOG(INFO) << "Initial endowments:";
        for (CommodityIndex = 0; CommodityIndex < static_cast<
           size_t>(NumberOfCommodities); ++CommodityIndex) {
```

```
            LOG(INFO) << "Commodity " << CommodityIndex << ": <
               exp.> = " << AlphaData.GetData(CommodityIndex)->
               GetAverage() << "; s.d. = " << AlphaData.GetData
               (CommodityIndex)->GetStdDev() <<
            "; <endow.> = " << EndowmentData.GetData(
               CommodityIndex)->GetAverage() << "; s.d. = " <<
               EndowmentData.GetData(CommodityIndex)->GetStdDev
               () << "; <MRS> = " <<
            LnMRSsData.GetData(CommodityIndex)->GetExpAverage()
                << "; s.d. = " << LnMRSsData.GetData(
               CommodityIndex)->GetStdDev();
        }
    }
    LOG(INFO) << "Average initial wealth (@ own prices) = " <<
       InitialOwnWealthData.GetAverage() << "; standard
       deviation = " << InitialOwnWealthData.GetStdDev();
    LOG(INFO) << "Initial sum of utilities = " <<
       LastSumOfUtilities;
}   //  AgentPopulation::Init()

void AgentPopulation::Reset() {
    for (auto& agent : Agents) {
        agent->Reset();
    }
}   //  AgentPopulation::Reset

void AgentPopulation::IntermediateOutput() {
    LOG(INFO) << "Through time " << theTime << ", " <<
       TotalInteractions << " total exchanges; ";

    switch (termination_criterion) {
        case -2:
        if (!LnMRSsDataUpToDate) {
            ComputeLnMRSsDistribution();
        }
        LOG(INFO) << "current L2 s.d. in MRS = " << LnMRSsData.
           L2StdDev();
        break;
        case -1:
        if (!LnMRSsDataUpToDate) {
            ComputeLnMRSsDistribution();
        }
        LOG(INFO) << "current L2 s.d. in MRS = " << LnMRSsData.
           L2StdDev();
        break;
```

```
        case 0:
        if (!LnMRSsDataUpToDate) {
            ComputeLnMRSsDistribution();
        }
        LOG(INFO) << "current max s.d. in MRS = " << LnMRSsData
           .LinfStdDev();
        break;
        case 1:
        LOG(INFO) << "relative increase in ΣU = " <<
           ComputeRelativeIncreaseInSumOfUtilities();
        break;
        case 2:
        LOG(INFO) << "increase in ΣU = " <<
           ComputeIncreaseInSumOfUtilities();
        break;
        default:
        LOG(ERROR) << "Invalid termination criterion";
        std::terminate();
        break;
    }   // switch...
}

bool AgentPopulation::TestConvergence() {
    switch (termination_criterion) {
        case -2:
        if (theTime >= TerminationTime) {
            return true;
        }
        break;
        case -1:  //  Termination based on L2 norm of MRS
           distribution
        ComputeLnMRSsDistribution();
        if (LnMRSsData.L2StdDev() < termination_eps) {
            return true;
        }
        break;
        case 0:  //  Termination based on L∞ norm of MRS
           distribution
        ComputeLnMRSsDistribution();
        if (LnMRSsData.LinfStdDev() < termination_eps) {
            return true;
        }
        break;
        case 1:  //  Termination based on relative increase in
           V
```

```
        if (ComputeRelativeIncreaseInSumOfUtilities() <
           termination_eps) {
            return true;
        }
        break;
        case 2:  //  Termination based on absolute increase in
           V
        if (ComputeIncreaseInSumOfUtilities() < termination_eps
           ) {
            return true;
        }
        break;
        default:
        LOG(ERROR) << "Invalid termination criterion";
        std::terminate();
        break;
    }   //  switch...
    return false;
}
void AgentPopulation::Trade (AgentPtr a1, AgentPtr a2) {
    AgentPtr LargerMRSAgent, SmallerMRSAgent;
    size_t id1 = a1->GetId();
    size_t id2 = a2->GetId();
    size_t Commodity1, Commodity2;
    double MRSratio12, MRSratio;
    double LAgentalpha1, LAgentalpha2, LAgentx1, LAgentx2,
       SAgentalpha1, SAgentalpha2, SAgentx1, SAgentx2;
    double num, denom, delta1, delta2;
    double Agent1PreTradeUtility, Agent2PreTradeUtility;

    if (debug) {
        Agent1PreTradeUtility = a1->Utility();
        Agent2PreTradeUtility = a2->Utility();
    }
    //  Next, select the commodities to trade...
    if (NumberOfCommodities == 2) {
        Commodity1 = 0;
        Commodity2 = 1;
    } else {
        Commodity1 = Rand->RandomCommodity();
        do {
            Commodity2 = Rand->RandomCommodity();
        } while (Commodity2 == Commodity1);
    }
            //  Compare MRSs...
```

```
MRSratio12 = a1->MRS(Commodity2, Commodity1) / a2->MRS(
   Commodity2, Commodity1);
if (MRSratio12 > 1.0) {
    MRSratio = MRSratio12;
} else {
    MRSratio = 1.0/MRSratio12;
}


if (MRSratio >= exp_trade_eps)  {  //  do exchange
    if (MRSratio12 > 1.0) {
        LargerMRSAgent = a1;
        SmallerMRSAgent = a2;
    } else {
        LargerMRSAgent = a2;
        SmallerMRSAgent = a1;
    }
    //  Here are the guts of bilateral Walrasian exchange
    SAgentalpha1 = SmallerMRSAgent->GetAlpha(Commodity1);
    SAgentalpha2 = SmallerMRSAgent->GetAlpha(Commodity2);
    SAgentx1 = SmallerMRSAgent->GetAllocation(Commodity1);
    SAgentx2 = SmallerMRSAgent->GetAllocation(Commodity2);
    LAgentalpha1 = LargerMRSAgent->GetAlpha(Commodity1);
    LAgentalpha2 = LargerMRSAgent->GetAlpha(Commodity2);
    LAgentx1 = LargerMRSAgent->GetAllocation(Commodity1);
    LAgentx2 = LargerMRSAgent->GetAllocation(Commodity2);

    num = (SAgentalpha1 * LAgentalpha2 * LAgentx1 *
       SAgentx2) - (LAgentalpha1 * SAgentalpha2 * LAgentx2
       * SAgentx1);
    denom = LAgentalpha1 * LAgentx2 + SAgentalpha1 *
       SAgentx2;
    delta1 = num / denom;
    denom = LAgentalpha2 * LAgentx1 + SAgentalpha2 *
       SAgentx1;
    delta2 = num / denom;

    //std::cout << delta1 << std::endl;
    SmallerMRSAgent->IncreaseAllocation(Commodity1, delta1)
       ;
    SmallerMRSAgent->IncreaseAllocation(Commodity2, -delta2
       );
    LargerMRSAgent->IncreaseAllocation(Commodity1, -delta1)
       ;
```

```
        LargerMRSAgent->IncreaseAllocation(Commodity2, delta2);

        SmallerMRSAgent->MarkSuccessfulTrade();
        LargerMRSAgent->MarkSuccessfulTrade();

        std::lock_guard<std::mutex> lock(m); // lock the global
            updates
        ++TotalInteractions;
        Volume[Commodity1] += delta1;
        Volume[Commodity2] += delta2;

    }

    if (debug) {
        if (a1->Utility() < Agent1PreTradeUtility) {
            LOG(WARNING) << "!!!Utility decreasing trade by
               agent #1!!!  Actual utility change = " << a1->
               Utility() - Agent1PreTradeUtility;
        }
        if (a2->Utility() < Agent2PreTradeUtility) {
            LOG(WARNING) << "!!!Utility decreasing trade by
               agent #2!!!  Actual utility change = " << a2->
               Utility() - Agent2PreTradeUtility;
        }
    }
}

std::tuple<long long, double, size_t, double, size_t>
   AgentPopulation::ParallelTrade (AgentPtr a1, AgentPtr a2) {
    AgentPtr LargerMRSAgent, SmallerMRSAgent;
    size_t id1 = a1->GetId();
    size_t id2 = a2->GetId();
    size_t Commodity1, Commodity2;
    double MRSratio12, MRSratio;
    double LAgentalpha1, LAgentalpha2, LAgentx1, LAgentx2,
       SAgentalpha1, SAgentalpha2, SAgentx1, SAgentx2;
    double num, denom;
    double Agent1PreTradeUtility, Agent2PreTradeUtility;

    long long interaction = 0;
    double delta1 = 0.0;
    double delta2 = 0.0;

    if (debug) {
        Agent1PreTradeUtility = a1->Utility();
```

```
    Agent2PreTradeUtility = a2->Utility();
}
//  Next, select the commodities to trade...
if (NumberOfCommodities == 2) {
    Commodity1 = 0;
    Commodity2 = 1;
} else {
    Commodity1 = Rand->RandomCommodity();
    do {
        Commodity2 = Rand->RandomCommodity();
    } while (Commodity2 == Commodity1);
}
        //  Compare MRSs...

MRSratio12 = a1->MRS(Commodity2, Commodity1) / a2->MRS(
   Commodity2, Commodity1);
if (MRSratio12 > 1.0) {
    MRSratio = MRSratio12;
} else {
    MRSratio = 1.0/MRSratio12;
}


if (MRSratio >= exp_trade_eps)  {  //  do exchange
    if (MRSratio12 > 1.0) {
        LargerMRSAgent = a1;
        SmallerMRSAgent = a2;
    } else {
        LargerMRSAgent = a2;
        SmallerMRSAgent = a1;
    }
    //  Here are the guts of bilateral Walrasian exchange
    SAgentalpha1 = SmallerMRSAgent->GetAlpha(Commodity1);
    SAgentalpha2 = SmallerMRSAgent->GetAlpha(Commodity2);
    SAgentx1 = SmallerMRSAgent->GetAllocation(Commodity1);
    SAgentx2 = SmallerMRSAgent->GetAllocation(Commodity2);
    LAgentalpha1 = LargerMRSAgent->GetAlpha(Commodity1);
    LAgentalpha2 = LargerMRSAgent->GetAlpha(Commodity2);
    LAgentx1 = LargerMRSAgent->GetAllocation(Commodity1);
    LAgentx2 = LargerMRSAgent->GetAllocation(Commodity2);

    num = (SAgentalpha1 * LAgentalpha2 * LAgentx1 *
       SAgentx2) - (LAgentalpha1 * SAgentalpha2 * LAgentx2
       * SAgentx1);
```

```
        denom = LAgentalpha1 * LAgentx2 + SAgentalpha1 *
           SAgentx2;
        delta1 = num / denom;
        denom = LAgentalpha2 * LAgentx1 + SAgentalpha2 *
           SAgentx1;
        delta2 = num / denom;

        SmallerMRSAgent->IncreaseAllocation(Commodity1, delta1)
           ;
        SmallerMRSAgent->IncreaseAllocation(Commodity2, -delta2
           );
        LargerMRSAgent->IncreaseAllocation(Commodity1, -delta1)
           ;
        LargerMRSAgent->IncreaseAllocation(Commodity2, delta2);

        SmallerMRSAgent->MarkSuccessfulTrade();
        LargerMRSAgent->MarkSuccessfulTrade();

        ++interaction;
    }

    if (debug) {
        if (a1->Utility() < Agent1PreTradeUtility) {
            LOG(WARNING) << "!!!Utility decreasing trade by
               agent #1!!!  Actual utility change = " << a1->
               Utility() - Agent1PreTradeUtility;
        }
        if (a2->Utility() < Agent2PreTradeUtility) {
            LOG(WARNING) << "!!!Utility decreasing trade by
               agent #2!!!  Actual utility change = " << a2->
               Utility() - Agent2PreTradeUtility;
        }
    }
    return std::make_tuple(interaction, delta1, Commodity1,
       delta2, Commodity2);
}

void AgentPopulation::TradeInFork (tbb::concurrent_vector<
   AgentPtr> a) {
    long long interactionsInFork, timeInFork;
    bool convergedInFork;
    AgentPtr Agent1, Agent2;
    tbb::concurrent_vector<AgentPtr>::iterator ait = a.begin();
    tbb::concurrent_vector<double> VolumeInFork;
    VolumeInFork.resize(NumberOfCommodities);
```

```
//int t = std::max(std::thread::hardware_concurrency(),
   static_cast<unsigned int>(NumberOfThreads));
int t = NumberOfThreads;
if (t == 0) {
    t = std::thread::hardware_concurrency();
}
//std::cout << t << std::endl;

interactionsInFork = 0;
for (int i = 0; i < PairwiseInteractionsPerPeriod/t; ++i) {
    //std::cout << i << std::endl;
    (this->*GetAgentPairInFork) (Agent1, Agent2, a, ait);
    Agent1->MarkActivated();
    Agent2->MarkActivated();

    AgentPtr LargerMRSAgent, SmallerMRSAgent;
    size_t id1 = Agent1->GetId();
    size_t id2 = Agent2->GetId();
    size_t Commodity1, Commodity2;
    double MRSratio12, MRSratio;
    double LAgentalpha1, LAgentalpha2, LAgentx1, LAgentx2,
       SAgentalpha1, SAgentalpha2, SAgentx1, SAgentx2;
    double num, denom, delta1, delta2;
    double Agent1PreTradeUtility, Agent2PreTradeUtility;


    if (debug) {
        Agent1PreTradeUtility = Agent1->Utility();
        Agent2PreTradeUtility = Agent2->Utility();
    }

    //  Next, select the commodities to trade...
    if (NumberOfCommodities == 2) {
        Commodity1 = 0;
        Commodity2 = 1;
    } else {
        Commodity1 = Rand->RandomCommodity();
        do {
            Commodity2 = Rand->RandomCommodity();
        } while (Commodity2 == Commodity1);
    }
        //  Compare MRSs...
```

```
    MRSratio12 = Agent1->MRS(Commodity2, Commodity1) /
       Agent2->MRS(Commodity2, Commodity1);
    if (MRSratio12 > 1.0) {
        MRSratio = MRSratio12;
    } else {
        MRSratio = 1.0/MRSratio12;
    }


if (MRSratio >= exp_trade_eps)  {  //  do exchange
    if (MRSratio12 > 1.0) {
        LargerMRSAgent= Agent1;
        SmallerMRSAgent= Agent2;
    } else {
        LargerMRSAgent= Agent2;
        SmallerMRSAgent= Agent1;
    }

    //  Here are the guts of bilateral Walrasian exchange
    SAgentalpha1 = SmallerMRSAgent->GetAlpha(Commodity1);
    SAgentalpha2 = SmallerMRSAgent->GetAlpha(Commodity2);
    SAgentx1 = SmallerMRSAgent->GetAllocation(Commodity1);
    SAgentx2 = SmallerMRSAgent->GetAllocation(Commodity2);
    LAgentalpha1 = LargerMRSAgent->GetAlpha(Commodity1);
    LAgentalpha2 = LargerMRSAgent->GetAlpha(Commodity2);
    LAgentx1 = LargerMRSAgent->GetAllocation(Commodity1);
    LAgentx2 = LargerMRSAgent->GetAllocation(Commodity2);

    num = (SAgentalpha1 * LAgentalpha2 * LAgentx1 *
       SAgentx2) - (LAgentalpha1 * SAgentalpha2 * LAgentx2
       * SAgentx1);
    denom = LAgentalpha1 * LAgentx2 + SAgentalpha1 *
       SAgentx2;
    delta1 = num / denom;
    denom = LAgentalpha2 * LAgentx1 + SAgentalpha2 *
       SAgentx1;
    delta2 = num / denom;

    SmallerMRSAgent->IncreaseAllocation(Commodity1, delta1)
       ;
    SmallerMRSAgent->IncreaseAllocation(Commodity2, -delta2
       );
    LargerMRSAgent->IncreaseAllocation(Commodity1, -delta1)
       ;
    LargerMRSAgent->IncreaseAllocation(Commodity2, delta2);
```

```
            SmallerMRSAgent->MarkSuccessfulTrade();
            LargerMRSAgent->MarkSuccessfulTrade();

            interactionsInFork++;
            VolumeInFork[Commodity1] += delta1;
            VolumeInFork[Commodity2] += delta2;
        }
    }

        std::lock_guard<std::mutex> lock(m); // lock the global
           updates
        //std::cout << "acquired lock" << std::endl;
        TotalInteractions += interactionsInFork;

        for (auto& vol : Volume) {
            auto i = static_cast<size_t>(&vol - &Volume[0]);
            vol += VolumeInFork[i];
        }
        return;
    }

    long long AgentPopulation::ForkAndJoinEquilibrate(int
       NumberOfEquilibrationsSoFar) {
        Converged = false;
        theTime = 0;
        TotalInteractions = 0;

        size_t split = static_cast<size_t>(NumberOfThreads);
        if (split == 0) {
            split = std::thread::hardware_concurrency();
        }
        //std::cout << split << std::endl;

        tbb::concurrent_vector<std::pair<size_t, size_t>>
           populations;

        // split the population
        for (size_t i = 0; i < split; ++i) {
            std::pair<size_t, size_t> population = std::make_pair(i
               *(NumberOfAgents/split), (i+1)*(NumberOfAgents/split
               ));
            populations.push_back(population);
        }
```

```
tbb::concurrent_vector<std::thread> threadPool;

do {
    LnMRSsDataUpToDate = false;  // Since these data are
       gonna change...

    ++theTime;

    if ((ShockPreferences) && (theTime % ShockPeriod == 0))
        {
        ShockAgentPreferences();
    }
    // std::cout << "Fork" << std::endl;
    tbb::parallel_for(static_cast<size_t>(0), split,
       static_cast<size_t>(1), [=](size_t i) {
        //std::cout << "reached" << std::endl;
        auto pop = tbb::concurrent_vector<AgentPtr>(Agents.
           begin() + populations[i].first, Agents.begin() +
            populations[i].second);
        TradeInFork(pop);
    });
    if (ShuffleAfterJoin) {
        std::shuffle(Agents.begin(), Agents.end(), Rand->
           GetGenerator());
    }

        //  Check for termination...
    if ((theTime > CheckTerminationThreshold) && (theTime %
        CheckTerminationPeriod == 0))  {
        Converged = TestConvergence();
    }

    //  Display stats if the time is right...
    if (PrintIntermediateOutput) {
        if (theTime % IntermediateOutputPrintPeriod == 0) {
            IntermediateOutput();
        }   //  theTime...

    //  Store the sum of utilities if it will be needed
       next period for either termincation check or
       printing
        if (termination_criterion > 0) {
```

```
                if (((theTime > CheckTerminationThreshold) &&
                   ((theTime + 1) % CheckTerminationPeriod ==
                   0)) || ((PrintIntermediateOutput) && ((
                   theTime + 1) % IntermediateOutputPrintPeriod
                    == 0))) {
                    LastSumOfUtilities = ComputeSumOfUtilities
                       ();
                }
            }

            if (writeToFile) { WriteLine(); }
        }
    } while (!Converged);



    //  Agents are either equilibrated or user has asked for
       termination; display stats for the former case
    if (!Converged) {
        LOG(INFO) << "Terminated by user!";
        return 0;
    } else {    //  the economy has converged...
        LOG(INFO) << "Equilibrium achieved at time " << theTime
            << " via " << TotalInteractions << " interactions";
        LOG(INFO) << "Equilibration #" <<
           NumberOfEquilibrationsSoFar << " ended";
        if (PrintConvergenceStats) {
            ConvergenceStatistics(Volume);
        }
        if(DumpAgentInformation) {
            DumpAgentInfo();
        }
        return TotalInteractions;
    }
}

long long AgentPopulation::ParallelEquilibrate(int
   NumberOfEquilibrationsSoFar) {
    Converged = false;
    theTime = 0;
    TotalInteractions = 0;
    //AgentPtr Agent1, Agent2;

    LOG(INFO) << "Equilibration #" <<
       NumberOfEquilibrationsSoFar << " starting";
```

```
//  Next, initialize some variables...
for (auto& vol : Volume) {
    vol = 0.0;
}

// Start up the exchange process here...
do {
    LnMRSsDataUpToDate = false;  // Since these data are
       gonna change...

    ++theTime;

    if ((ShockPreferences) && (theTime % ShockPeriod == 0))
        {
        ShockAgentPreferences();
    }


    results.clear();
    results.resize(PairwiseInteractionsPerPeriod);

    tbb::parallel_for(static_cast<size_t>(0), static_cast<
       size_t>(PairwiseInteractionsPerPeriod), static_cast<
       size_t>(1), [this](size_t i) {
        AgentPtr Agent1, Agent2;
        (this->*GetAgentPair) (Agent1, Agent2);
        std::lock_guard<std::mutex> lock1(Agent1->m);
        std::lock_guard<std::mutex> lock2(Agent2->m);
        Agent1->MarkActivated();
        Agent2->MarkActivated();
        results[i] = ParallelTrade(Agent1, Agent2);
    });

    for (auto &r : results) {
        TotalInteractions += std::get<0>(r);
        Volume[std::get<2>(r)] += std::get<1>(r);
        Volume[std::get<4>(r)] += std::get<3>(r);
    }

    //  Check for termination...
    if ((theTime > CheckTerminationThreshold) && (theTime %
        CheckTerminationPeriod == 0))  {
        Converged = TestConvergence();
    }
```

```
        //  Display stats if the time is right...
        if (PrintIntermediateOutput && (theTime %
           IntermediateOutputPrintPeriod == 0)) {
            IntermediateOutput();
        }

        //  Store the sum of utilities if it will be needed
           next period for either termincation check or
           printing
        if (termination_criterion > 0) {
            if (((theTime > CheckTerminationThreshold) && ((
               theTime + 1) % CheckTerminationPeriod == 0)) ||
               ((PrintIntermediateOutput) && ((theTime + 1) %
               IntermediateOutputPrintPeriod == 0))) {
                LastSumOfUtilities = ComputeSumOfUtilities();
            }
        }

        if (writeToFile) { WriteLine(); }

    }  while (!Converged);

    //  Agents are either equilibrated or user has asked for
       termination; display stats for the former case
    if (!Converged) {
        LOG(INFO) << "Terminated by user!";
        return 0;
    } else {    //  the economy has converged...
        LOG(INFO) << "Equilibrium achieved at time " << theTime
            << " via " << TotalInteractions << " interactions";
        LOG(INFO) << "Equilibration #" <<
           NumberOfEquilibrationsSoFar << " ended";
        if (PrintConvergenceStats) {
            ConvergenceStatistics(Volume);
        }
        if(DumpAgentInformation) {
            DumpAgentInfo();
        }
        return TotalInteractions;
    }
}

long long AgentPopulation::Equilibrate(int
   NumberOfEquilibrationsSoFar) {
```

```
Converged = false;
theTime = 0;
TotalInteractions = 0;

LOG(INFO) << "Equilibration #" <<
   NumberOfEquilibrationsSoFar << " starting";
//  Next, initialize some variables...
for (auto& vol : Volume) {
    vol = 0.0;
}

//  Start up the exchange process here...
do {
    LnMRSsDataUpToDate = false;  //  Since these data are
       gonna change...

    ++theTime;

    if ((ShockPreferences) && (theTime % ShockPeriod == 0))
        {
        ShockAgentPreferences();
    }

    for (int i = 1; i <= PairwiseInteractionsPerPeriod; ++i
       ) {
        AgentPtr Agent1, Agent2;
        (this->*GetAgentPair) (Agent1, Agent2);
        Agent1->MarkActivated();
        Agent2->MarkActivated();
        Trade(Agent1,Agent2);
    }   //  for i...

    //  Check for termination...
    if ((theTime > CheckTerminationThreshold) && (theTime %
        CheckTerminationPeriod == 0))  {
        Converged = TestConvergence();
    }

    //  Display stats if the time is right...
    if (PrintIntermediateOutput && (theTime %
       IntermediateOutputPrintPeriod == 0)) {
        IntermediateOutput();
    }
```

```
        //  Store the sum of utilities if it will be needed
            next period for either termincation check or
            printing
        if (termination_criterion > 0) {
            if (((theTime > CheckTerminationThreshold) && ((
               theTime + 1) % CheckTerminationPeriod == 0)) ||
               ((PrintIntermediateOutput) && ((theTime + 1) %
               IntermediateOutputPrintPeriod == 0))) {
                LastSumOfUtilities = ComputeSumOfUtilities();
            }
        }
        if (writeToFile) { WriteLine(); }

    }  while (!Converged);

    //  Agents are either equilibrated or user has asked for
        termination; display stats for the former case
    if (!Converged) {
        LOG(INFO) << "Terminated by user!";
        return 0;
    } else {    //  the economy has converged...
        LOG(INFO) << "Equilibrium achieved at time " << theTime
            << " via " << TotalInteractions << " interactions";
        LOG(INFO) << "Equilibration #" <<
            NumberOfEquilibrationsSoFar << " ended";
        if (PrintConvergenceStats) {
            ConvergenceStatistics(Volume);
        }

        if(DumpAgentInformation) {
            DumpAgentInfo();
        }
        return TotalInteractions;
    }
}   //  AgentPopulation::Equilibrate

void AgentPopulation::DumpAgentInfo() {
    const char* filename = "dump.csv";
    std::ofstream dumpfile;

    dumpfile.open(filename, std::ios::trunc);

    for (auto &a : Agents) {
```

```
        dumpfile << a->GetId() << "," << a->
            GetNumberOfActivations() << "," << a->
            GetNumberOfTrades() << std::endl;
    }
    dumpfile.close();
}

void AgentPopulation::ConvergenceStatistics(CommodityArray
    VolumeStats) {
    Data InitialMarketWealthData, FinalMarketWealthData,
        DeltaMarketWealthData;
    Data FinalOwnWealthData, DeltaOwnWealthData, DeltaUtility;
    double price, AgentsInitialWealth, AgentsFinalWealth;

    for (auto& agent : Agents) {
        //  First compute agent wealth one commodity at a time
            ...
        AgentsInitialWealth = 0.0;
        AgentsFinalWealth = 0.0;
        for (size_t j = 0; j < static_cast<size_t>(
            NumberOfCommodities); ++j) {
            price = LnMRSsData.GetData(j)->GetExpAverage();
            AgentsInitialWealth += agent->GetEndowment(j) *
                price;
            AgentsFinalWealth += agent->GetAllocation(j) *
                price;
        }
        InitialMarketWealthData.AddDatum(AgentsInitialWealth);
        FinalMarketWealthData.AddDatum(AgentsFinalWealth);
        DeltaMarketWealthData.AddDatum(AgentsFinalWealth -
            AgentsInitialWealth);

        //  The following line is a minor optimization...
            variable name should include 'own' to be mnemonic...
        AgentsFinalWealth = agent->Wealth(agent->GetCurrentMRSs
            ());
        FinalOwnWealthData.AddDatum(AgentsFinalWealth);
        DeltaOwnWealthData.AddDatum(AgentsFinalWealth - agent->
            GetInitialWealth());
        DeltaUtility.AddDatum(agent->Utility() - agent->
            GetInitialUtility());
    }   //  for i...
```

```
    LOG(INFO) << "Average initial wealth (@ market prices) = "
       << InitialMarketWealthData.GetAverage() << "; standard
       deviation = " << InitialMarketWealthData.GetStdDev();
    LOG(INFO) << "Average final wealth (@ market prices)   = "
       << FinalMarketWealthData.GetAverage() << "; standard
       deviation = " << FinalMarketWealthData.GetStdDev();
    LOG(INFO) << "Average change in market wealth          = "
       << DeltaMarketWealthData.GetAverage() << "; standard
       deviation = " << DeltaMarketWealthData.GetStdDev();
    LOG(INFO) << "Average final wealth (@ own prices)      = "
       << FinalOwnWealthData.GetAverage() << "; standard
       deviation = " << FinalOwnWealthData.GetStdDev();
    LOG(INFO) << "Average change in own wealth             = "
       << DeltaOwnWealthData.GetAverage() << "; standard
       deviation = " << DeltaOwnWealthData.GetStdDev();
    LOG(INFO) << "Minimum increase in utility              = "
       << DeltaUtility.GetMin() << "; maximum increase = " <<
       DeltaUtility.GetMax();
    LOG(INFO) << "Final sum of utilities                   = "
       << ComputeSumOfUtilities();
    if (PrintFinalCommodityList) {
        for (size_t i = 0; i < static_cast<size_t>(
           NumberOfCommodities); ++i) {
            LOG(INFO) << "Commodity " << i << ": volume = " <<
               VolumeStats[i] << "; avg. MRS = " << LnMRSsData.
               GetData(i)->GetExpAverage() <<
            "; s.d. = " << LnMRSsData.GetData(i)->GetStdDev();
        }
    }
}   //  AgentPopulation::ConvergenceStatistics()

void AgentPopulation::CompareTwoAgents(AgentPtr Agent1,
   AgentPtr Agent2) {
    for (size_t j = 0; j < static_cast<size_t>(
       NumberOfCommodities); ++j) {
        if (Agent1->GetAlpha(j) != Agent2->GetAlpha(j)) {
            LOG(WARNING) << "Bad alpha copying!";
        }
        if (Agent1->GetEndowment(j) != Agent2->GetEndowment(j))
            {
            LOG(WARNING) << "Bad endowment copying!";
        }
        if (Agent1->GetAllocation(j) != Agent2->GetAllocation(j
           )) {
            LOG(WARNING) << "Bad allocation copying!";
```

```
        }
        if (Agent1->GetInitialMRS(j) != Agent2->GetInitialMRS(j
           )) {
            LOG(WARNING) << "Bad InitialMRSs copying!";
        }
        if (Agent1->GetCurrentMRS(j) != Agent2->GetCurrentMRS(j
           )) {
            LOG(WARNING) << "Bad CurrentMRSs copying!";
        }
    }
    if (Agent1->GetInitialUtility() != Agent2->
       GetInitialUtility()) {
        LOG(WARNING) << "Bad InitialUtility copying!";
    }
    if (Agent1->GetInitialWealth() != Agent2->GetInitialWealth
       ()) {
        LOG(WARNING) << "Bad InitialWealth copying!";
    }
}   //  AgentPopulation::CompareTwoAgents()

void AgentPopulation::ShockAgentPreferences() {
    if (debug) { LOG(DEBUG) << "Shocking agent preferences...";
        }
    double oldPref, newPref, pref;

    size_t CommodityToShock = Rand->RandomCommodity();
    bool sign = Rand->RandomBinary();
    double shock = Rand->RandomShock();
    if (debug) {
        if (sign) {
            LOG(DEBUG) << "Shocking commodity " <<
               CommodityToShock << " * " << shock;
        } else {
            LOG(DEBUG) << "Shocking commodity " <<
               CommodityToShock << " * 1/" << shock;
        }
    }
    for (auto& ActiveAgent : Agents) {
        oldPref = ActiveAgent->GetAlpha(CommodityToShock);
        if (sign) {
            newPref = oldPref * shock;
        } else {
            newPref = oldPref/shock;
        }
        ActiveAgent->SetAlpha(CommodityToShock, newPref);
```

```
        for (size_t CommodityIndex = 0; CommodityIndex <
           static_cast<size_t>(NumberOfCommodities); ++
           CommodityIndex) {
            pref = ActiveAgent->GetAlpha(CommodityIndex);
            ActiveAgent->SetAlpha(CommodityIndex, pref/(1.0 -
               oldPref + newPref));
        }   //  for (CommodityIndex...
    }   //  for (AgentIndex...
}   //  AgentPopulation::ShockAgentPreferenes()

void AgentPopulation::WriteLine() {
    // This can obviously be made more elegant.
    outfile << theTime <<"," << TotalInteractions << "," <<
       WriteWealthInfo() << WriteUtilityInfo();
    outfile << LnMRSsData.L2StdDev() << "," << LnMRSsData.
       LinfStdDev() << "," << InitialOwnWealthData.GetMin() <<
       "," << InitialOwnWealthData.GetMax() << "," ;
    outfile << alphaMin << "," << alphaMax << "," <<
       activationMethod << ",";
    outfile << NumberOfAgents << "," << NumberOfCommodities <<
       "," << trade_eps << ",";
    outfile << termination_criterion << "," << termination_eps
       << "," << TerminationTime << "," <<
       CheckTerminationThreshold;

    outfile << std::endl;
}

/*================= End of Methods =================*/
void OpenFile(const char * filename) {
    bool fileExists = false;

    std::ifstream file_to_check (filename);
    if(file_to_check.is_open()) {
        fileExists = true;
    }
    file_to_check.close();

    if(debug) {
        LOG(DEBUG) << "Opening file " << filename;
    }

    if (fileAppend) {
        outfile.open(filename, std::ios::app);
```

```
        if (!fileExists) {
            WriteHeader();
        }
    } else {
        outfile.open(filename, std::ios::trunc);
        WriteHeader();
    }
}

void WriteHeader() {
    outfile << "time,interactions,currentwealth.min,
       currentwealth.max,currentwealth.avg,currentwealth.sd,";
    outfile << "utility.min,utility.max,utility.avg,utility.sd,
       ";
    outfile << "L2.sd.MRS,max.sd.MRS,initialwealth.min,
       initialwealth.max,";
    outfile << "alpha.min,alpha.max,activation.method,num.
       agents,num.commodities,";
    outfile << "trade.eps,termination_criterion,termination.eps
       ,termination.time,termination.threshold";
    outfile << std::endl;
}

void InitMiscellaneous() {
    LOG(INFO) << "Model version: " << Version;
    LOG(INFO) << "Number of agents: " << NumberOfAgents;
    LOG(INFO) << "Number of commodities: " <<
       NumberOfCommodities;
    LOG(INFO) << "Trade epsilon: " << trade_eps;
    LOG(INFO) << "Time period: " << 2 *
       PairwiseInteractionsPerPeriod;
    LOG(INFO) << "Termination period check rate: " <<
       CheckTerminationPeriod;
    LOG(INFO) << "Termination period check threshold: " <<
       CheckTerminationThreshold;
    switch (termination_criterion) {
        case -2:
        LOG(INFO) << "Termination criterion: After " <<
           TerminationTime << " time steps";
        break;
        case -1:
        LOG(INFO) << "Termination criterion: L2 norm of
           standard deviation of agent MRSes";
        break;
        case 0:
```

```
        LOG(INFO) << "Termination criterion: Maximum standard
           deviation of agent MRSes";
        break;
        case 1:
        LOG(INFO) << "Termination criterion: relative increase
           in sum of agent utilities";
        break;
        case 2:
        LOG(INFO) << "Termination criterion: increase in the
           sum of agent utilities";
        break;
        default:
        LOG(ERROR) << "Invalid termination criterion";
        std::terminate();
        break;
    }
    LOG(INFO) << "Termination threshold: " << termination_eps;
    if (DefaultSerialExecution) {
        LOG(INFO) << "Parallel activation: FALSE";
    } else {
        LOG(INFO) << "Parallel activation: TRUE";
        LOG(INFO) << "Agent randomization size: " <<
           AgentsToRandomize;
    }
    LOG(INFO) << "Number of equilibrations: " <<
       RequestedEquilibrations;
    LOG(INFO) << "Vary agent initial conditions: " << !
       SameAgentInitialCondition;

    MemoryState.WriteMemoryRequirements();
} // InitMiscellaneous()

void ReadConfigFile(std::string file) {
    // This function sets all relevant model parameters by
       reading from a config file (libconfig).
    // If there's some issue with the formatting or reading of
       the config file, it catches the exception
    // and terminates the program. The config file *must* be
       proper for the model to run. See parameters.cfg
    // in the repository for an example.

    libconfig::Config config;
    try {
        LOG(INFO) << "Loading configuration from " << file << "
           ...";
```

```
config.readFile(file.c_str());
LOG(INFO) << "Loaded configuration from " << file;
if (config.lookupValue("debug.enabled", debug) && debug
   ) { LOG(DEBUG) << "debug: " << debug; }
if (config.lookupValue("number.commodities",
   NumberOfCommodities) && debug) { LOG(DEBUG) << "
   NumberOfCommodities: " << NumberOfCommodities; }
if (config.lookupValue("number.agents", NumberOfAgents)
    && debug) { LOG(DEBUG) << "NumberOfAgents: " <<
   NumberOfAgents; }
if (config.lookupValue("rand.use_seed", UseRandomSeed)
   && debug) { LOG(DEBUG) << "UseRandomSeed: " <<
   UseRandomSeed; }
if (config.lookupValue("rand.seed", NonRandomSeed) &&
   debug) { LOG(DEBUG) << "NonRandomSeed: " <<
   NonRandomSeed; }
if (config.lookupValue("interactions_per_period",
   PairwiseInteractionsPerPeriod) && debug) { LOG(DEBUG
   ) << "PairwiseInteractionsPerPeriod: " <<
   PairwiseInteractionsPerPeriod; }
if (config.lookupValue("alpha.min", alphaMin) && debug)
    { LOG(DEBUG) << "alphaMin: " << alphaMin; }
if (config.lookupValue("alpha.max", alphaMax) && debug)
    { LOG(DEBUG) << "alphaMax: " << alphaMax; }
if (config.lookupValue("wealth.min", wealthMin) &&
   debug) { LOG(DEBUG) << "wealthMin: " << wealthMin; }
if (config.lookupValue("wealth.max", wealthMax) &&
   debug) { LOG(DEBUG) << "wealthMax: " << wealthMax; }
if (config.lookupValue("parallel.disabled",
   DefaultSerialExecution) && debug) { LOG(DEBUG) << "
   DefaultSerialExecution: " << DefaultSerialExecution;
    }
if (config.lookupValue("parallel.number_of_threads",
   NumberOfThreads) && debug) { LOG(DEBUG) << "
   NumberOfThreads: " << NumberOfThreads; }
if (config.lookupValue("parallel.fork_and_join",
   ForkAndJoin) && debug) { LOG(DEBUG) << "ForkAndJoin:
    " << ForkAndJoin; }
if (config.lookupValue("parallel.shuffle_after_join",
   ShuffleAfterJoin) && debug) { LOG(DEBUG) << "
   ShuffleAfterJoin: " << ShuffleAfterJoin; }
if (config.lookupValue("num_equilibrations",
   RequestedEquilibrations) && debug) { LOG(DEBUG) << "
   RequestedEquilibrations: " <<
   RequestedEquilibrations; }
```

```
if (config.lookupValue("
   same_conditions_each_equilibration",
   SameAgentInitialCondition) && debug) { LOG(DEBUG) <<
    "SameAgentInitialCondition: " <<
   SameAgentInitialCondition; }
if (config.lookupValue("trade.eps", trade_eps) && debug
   ) { LOG(DEBUG) << "trade_eps: " << trade_eps; }
if (config.lookupValue("termination.criterion",
   termination_criterion) && debug) { LOG(DEBUG) << "
   termination_criterion: " << termination_criterion; }
if (config.lookupValue("termination.time",
   TerminationTime) && debug) { LOG(DEBUG) << "
   TerminationTime: " << TerminationTime; }
if (config.lookupValue("termination.eps",
   termination_eps) && debug) { LOG(DEBUG) << "
   termination_eps: " << termination_eps; }
if (config.lookupValue("termination.threshold",
   CheckTerminationThreshold) && debug) { LOG(DEBUG) <<
    "CheckTerminationThreshold: " <<
   CheckTerminationThreshold; }
if (config.lookupValue("termination.period",
   CheckTerminationPeriod) && debug) { LOG(DEBUG) << "
   CheckTerminationPeriod: " << CheckTerminationPeriod;
    }
if (config.lookupValue("shock.enabled",
   ShockPreferences) && debug) { LOG(DEBUG) << "
   ShockPreferences: " << ShockPreferences; }
if (config.lookupValue("shock.period", ShockPeriod) &&
   debug) { LOG(DEBUG) << "ShockPeriod: " <<
   ShockPeriod; }
if (config.lookupValue("shock.min", MinShock) && debug)
    { LOG(DEBUG) << "MinShock: " << MinShock; }
if (config.lookupValue("shock.max", MaxShock) && debug)
    { LOG(DEBUG) << "MaxShock: " << MaxShock; }
if (config.lookupValue("debug.print_endowments",
   PrintEndowments) && debug) { LOG(DEBUG) << "
   PrintEndowments: " << PrintEndowments; }
if (config.lookupValue("debug.print_intermediate_output
   ", PrintIntermediateOutput) && debug) { LOG(DEBUG)
   << "PrintIntermediateOutput: " <<
   PrintIntermediateOutput; }
```

```
        if (config.lookupValue("debug.
           intermediate_output_print_period",
           IntermediateOutputPrintPeriod) && debug) { LOG(DEBUG
           ) << "IntermediateOutputPrintPeriod: " <<
           IntermediateOutputPrintPeriod; }
        if (config.lookupValue("debug.print_convergence_stats",
            PrintConvergenceStats) && debug) { LOG(DEBUG) << "
           PrintConvergenceStats: " << PrintConvergenceStats; }
        if (config.lookupValue("debug.
           print_final_commodity_list", PrintFinalCommodityList
           ) && debug) { LOG(DEBUG) << "PrintFinalCommodityList
           : " << PrintFinalCommodityList; }
        if (config.lookupValue("debug.dump_agent_information",
           DumpAgentInformation) && debug) { LOG(DEBUG) << "
           DumpAgentInformation: " << DumpAgentInformation; }
        if (config.lookupValue("activation.method",
           activationMethod) && debug) { LOG(DEBUG) << "
           Activation Method: " << activationMethod; } // TODO:
            make this an enum
        if (config.lookupValue("file.filename", outputFilename)
            && debug) { LOG(DEBUG) << "Output filename: " <<
           outputFilename; }
        if (config.lookupValue("file.write_to_file",
           writeToFile) && debug) { LOG(DEBUG) << "Write to
           file?: " << writeToFile; }
        if (config.lookupValue("file.append", fileAppend) &&
           debug) { LOG(DEBUG) << "Append to file?: " <<
           fileAppend; }
        exp_trade_eps = exp(trade_eps);

    } catch (...) {
        LOG(ERROR) << "Error reading config file";
        std::terminate();
    }
} //ReadConfigFile

int main(int argc, char** argv) {
    // Preliminaries: Parse flags, etc.
    std::string usage = "An agent-based model of bilateral
       exchange. Usage:\n";
    usage += argv[0];
    //gflags::SetUsageMessage(usage);
    //gflags::ParseCommandLineFlags(&argc, &argv, true);
    LOG(INFO) << "Opening log file...";
```

```
// Read the config file passed through the -file flag, or
   read the default parameters.cfg.
std::string param_file = "";
if (argv[1] != NULL) {
    param_file = argv[1];
}
if (param_file.empty()) {
    param_file = "parameters.cfg";
}
ReadConfigFile(param_file);

outputFilename = "data.csv"; // workaround for libconfig++
   issue, TODO: correct

if (writeToFile) { OpenFile(outputFilename); }

if (!DefaultSerialExecution && NumberOfThreads > 0) {
    tbb::task_scheduler_init init(NumberOfThreads);
}

Rand = new RNG(UseRandomSeed, NonRandomSeed, NumberOfAgents
   , NumberOfCommodities, MinShock, MaxShock, alphaMin,
   alphaMax, wealthMin, wealthMax);

//  Initialize the model and print preliminaries to the log
   .
InitMiscellaneous();
AgentPopulationPtr PopulationPtr = new AgentPopulation(
   NumberOfCommodities);


PopulationPtr->Init();

//  Equilibrate the agent economy once...
int EquilibrationNumber = 1;
long long sum;

if (DefaultSerialExecution) {
    sum = PopulationPtr->Equilibrate(EquilibrationNumber);
} else {
    if (ForkAndJoin) {
        sum = PopulationPtr->ForkAndJoinEquilibrate(
           EquilibrationNumber);
    } else {
```

```
            sum = PopulationPtr->ParallelEquilibrate(
               EquilibrationNumber);
        }
    }
    long long sum2 = sum*sum;
    //  Equilibrate again if the user has requested this...
    long long interactions;

    EquilibrationNumber = 2;
    while (EquilibrationNumber <= RequestedEquilibrations) {
        if (SameAgentInitialCondition) {
            PopulationPtr->Reset();
        } else {
            PopulationPtr->Init();
        }
        interactions = PopulationPtr->Equilibrate(
           EquilibrationNumber);
        sum += interactions;
        sum2 += interactions*interactions;
        ++EquilibrationNumber;
    }

    double avg = static_cast<double>(sum)/(EquilibrationNumber
       -1);
    LOG(INFO) << "Average number of interactions: " << avg;
    if (EquilibrationNumber > 2) {
        LOG(INFO) << "std. dev.: " << sqrt((sum2 - (
           EquilibrationNumber - 1) * avg * avg)/(
           EquilibrationNumber - 2));
    }


    if (outfile.is_open()) { outfile.close(); }
}  // main()
```

D.5 Dockerfile

```
#FROM ubuntu:14.04
FROM ubuntu:15.10

RUN apt-get clean
RUN rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
RUN echo "deb http://dl.bintray.com/sbt/debian /" | tee -a /etc
   /apt/sources.list.d/sbt.list
```

```
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --
   recv 642AC823
RUN apt-get update
RUN apt-get install -yqq time git curl build-essential
   checkinstall autotools-dev wget cmake software-properties-
   common libtbb-dev vim
RUN apt-get clean
RUN rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
RUN mkdir -p model/serial-activation-suite



COPY RNG.cpp model/
COPY RNG.h model/
COPY main.cpp model/
COPY main.h model/
COPY parameters.cfg model/
COPY easylogging++.h model/
COPY ctpl_stl.h model/
COPY Makefile model/
COPY serial-activation-suite/ model/serial-activation-suite
COPY parallel-activation-suite/ model/parallel-activation-suite
COPY forkjoin-activation-suite/ model/forkjoin-activation-suite
COPY forkjoin-threading-suite/ model/forkjoin-threading-suite
WORKDIR /model/

RUN wget http://www.hyperrealm.com/libconfig/libconfig-1.5.tar.
   gz
RUN tar xvf libconfig-1.5.tar.gz
RUN wget https://github.com/gflags/gflags/archive/v2.1.2.tar.gz
RUN tar xvf v2.1.2.tar.gz
WORKDIR /model/libconfig-1.5
RUN ./configure && make && make install
WORKDIR /model/gflags-2.1.2/build
RUN cmake .. && make && make install
RUN ldconfig

WORKDIR /model/serial-activation-suite
CMD ["bash"]

#CMD /
#CMD go build zi-traders.go
#CMD echo "Built Go model"
#CMD bash -c "/usr/bin/time -f '1,%e,%U,%S' ./zi-traders"
```

```
#docker-machine create -d virtualbox --virtualbox-boot2docker-
   url file://$HOME/Dropbox/boot2docker-v1.9.1-fix1.iso --
   virtualbox-memory 1536 --virtualbox-disk-size 10000
   fixedjava
#docker-machine create -d virtualbox --virtualbox-boot2docker-
   url https://github.com/tianon/boot2docker-legacy/releases/
   download/v1.10.0-rc1/boot2docker.iso --virtualbox-memory
   1536 --virtualbox-disk-size 10000 --virtualbox-cpu-count 2
   fixedjava
#docker-machine create -d virtualbox --virtualbox-boot2docker-
   url https://github.com/tianon/boot2docker-legacy/releases/
   download/v1.10.0-rc1/boot2docker.iso --virtualbox-memory
   1024 --virtualbox-disk-size 10000 --virtualbox-cpu-count 2
   fixedjava
```

Bibliography

Aldous, D. (2013). Interacting particle systems as stochastic social dynamics. *Bernoulli*, *19*(4), 1122–1149. doi: 10.3150/12-BEJSP04

Alizadeh, M., & Cioffi-Revilla, C. (2015). Activation regimes in opinion dynamics: Comparing asynchronous updating schemes. *Journal of Artificial Societies and Social Simulation*, *18*(3). doi: 10.18564/jasss.2733

Angus, S. D., & Hassani-Mahmooei, B. (2015). "Anarchy" reigns: A quantitative analysis of agent-based modelling publication practices in JASSS, 2001-2012. *Journal of Artificial Societies and Social Simulation*, *18*(4). Retrieved from `http://jasss.soc.surrey.ac.uk/18/4/16.html` doi: 10.18564/jasss.2952

Auble, B. D. (2015). *Narrative agents as a reporting mechanism for agent-based models* (Master's thesis). George Mason University, Fairfax, VA.

Axtell, R. L. (2000a). *Effects of interaction topology and activation regime in several multi-agent systems* (Working Paper No. 12). Center on Social & Economic Dynamics: Brookings Institution.

Axtell, R. L. (2000b). *Why agents?: On the varied motivations for agent computing in the social sciences* (Working Paper No. 17). Center on Social & Economic Dynamics: Brookings Institution.

Axtell, R. L. (2005). The complexity of exchange. *The Economic Journal*, *115*(504), F193–F210. doi: 10.1111/j.1468-0297.2005.01001.x

Axtell, R. L., Axelrod, R., Epstein, J. M., & Cohen, M. D. (1996). Aligning simulation models: A case study and results. *Computational and Mathematical Organization Theory*, *1*(2), 123–141. doi: 10.1007/BF01299065

Baetens, J., Van der Weeën, P., & De Baets, B. (2012). Effect of asynchronous updating on the stability of cellular automata. *Chaos, Solitons & Fractals*, *45*(4), 383–394. doi: 10.1016/j.chaos.2012.01.002

Barabási, A.-L., & Albert, R. (1999). Emergence of scaling in random networks. *Science*, *286*(5439), 509–512. doi: 10.1126/science.286.5439.509

Beberg, A. L., Ensign, D. L., Jayachandran, G., Khaliq, S., & Pande, V. S. (2009). Folding@Home: Lessons from eight years of volunteer distributed computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (pp. 1–8). Washington, DC: IEEE Computer Society. doi: 10.1109/IPDPS.2009.5160922

Bennett, D. A., & Tang, W. (2006). Modelling adaptive, spatially aware, and mobile agents: Elk migration in Yellowstone. *International Journal of Geographical Information Science*, *20*(9), 1039–1066. doi: 10.1080/13658810600830806

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. (arXiv: 1410.0846v1 [cs.SE]) doi: 10.1145/2723872.2723882

Caron-Lormier, G., Humphry, R. W., Bohan, D. A., Hawes, C., & Thorbek, P. (2008). Asynchronous and synchronous updating in individual-based models. *Ecological Modelling*, *212*(3-4), 522–527. doi: 10.1016/j.ecolmodel.2007.10.049

Chen, S.-H. (2012). Varieties of agents in agent-based computational economics: A historical and an interdisciplinary perspective. *Journal of Economic Dynamics and Control*, *36*(1), 1–25. doi: 10.1016/j.jedc.2011.09.003

Cioffi-Revilla, C. (2014). *Introduction to computational social science: Principles and applications*. London: Springer-Verlag.

Coakley, S., Smallwood, R., & Holcombe, M. (2006). Using X-machines as a formal basis for describing agents in agent-based modelling. *Simulation Series*, *38*(2), 33.

Collier, N. T., & North, M. J. (2013). Parallel agent-based simulation with Repast for High Performance Computing. *Simulation*, *89*(10), 1215–1235. doi: 10.1177/0037549712462620

Comer, K. W. (2014). *Who goes first? An examination of the impact of activation on outcome behavior in agent-based models* (Ph.D. dissertation). George Mason University, Fairfax, VA.

Cordasco, G., Chiara, R. D., Mancuso, A., Mazzeo, D., Scarano, V., & Spagnuolo, C. (2013). Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason. *Simulation*, *89*(10), 1236–1253. doi: 10.1177/0037549713489594

Cordasco, G., Milone, F., Spagnuolo, C., & Vicidomini, L. (2014). Exploiting D-Mason on parallel platforms: A novel communication strategy. In *Euro-Par 2014: Parallel Processing Workshops* (pp. 407–417). Cham: Springer International.

Cox, R. (n.d.). *Bell Labs and CSP threads.* Retrieved 2015-09-18, from `https://swtch.com/~rsc/thread/`

Crooks, A. T., Castle, C., & Batty, M. (2008). Key challenges in agent-based modelling for geo-spatial simulation. *Computers, Environment and Urban Systems*, *32*(6), 417–430. doi: 10.1016/j.compenvurbsys.2008.09.004

Crooks, A. T., & Castle, C. J. E. (2012). The integration of agent-based modelling and geographical information for geospatial simulation. In A. J. Heppenstall, A. T. Crooks, L. M. See, & M. Batty (Eds.), *Agent-based models of geographical systems* (pp. 219–251). Dordrecht: Springer.

Crooks, A. T., Croitoru, A., Lu, X., Wise, S., Irvine, J., & Stefanidis, A. (2015). Walk this way: Improving pedestrian agent-based models through scene activity analysis. *ISPRS International Journal of Geo-Information*, *4*(3), 1627–1656. doi: 10.3390/ijgi4031627

Crooks, A. T., & Hailegiorgis, A. B. (2014). An agent-based modeling approach applied to the spread of cholera. *Environmental Modelling & Software*, *62*, 164–177. doi: 10.1016/j.envsoft .2014.08.027

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, *18*(8), 453–457. doi: 10.1145/360933.360975

D'Souza, R. M., Lysenko, M., & Rahmani, K. (2007). Sugarscape on steroids: Simulating over a million agents at interactive rates. In *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence* (Vol. 20). Argonne, IL.

Epstein, J. M. (2002). Modeling civil violence: An agent-based computational approach. *Proceedings of the National Academy of Sciences*, *99*(Supplement 3), 7243–7250. doi: 10.1073/ pnas.092080199

Epstein, J. M., & Axtell, R. L. (1996). *Growing artificial societies: Social science from the bottom up*. Washington, DC: Brookings Institution Press.

Erdős, P., & Rényi, A. (1960). On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences* (pp. 17–61).

Fachada, N., Lopes, V. V., Martins, R. C., & Rosa, A. C. (2015). Towards a standard model for research in agent-based modeling and simulation. *PeerJ Computer Science*, *1*, e36. doi: 10.7717/peerj-cs.36

Fachada, N., Lopes, V. V., Martins, R. C., & Rosa, A. C. (2016). Parallelization strategies for spatial agent-based models. *International Journal of Parallel Programming*, 1–33. doi: 10.1007/s10766-015-0399-9

Fatès, N., & Chevrier, V. (2010). How important are updating schemes in multi-agent systems? An illustration on a multi-turmite model. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems* (Vol. 1, pp. 533–540). Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Filatova, T., Parker, D., & van der Veen, A. (2009). Agent-based urban land markets: agent's pricing behavior, land prices and urban land use change. *Journal of Artificial Societies and Social Simulation*, *12*(1), 3. Retrieved from `http://jasss.soc.surrey.ac.uk/12/1/3.html`

Galán, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., del Olmo, R., López-Paredes, A., & Edmonds, B. (2009). Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, *12*(1), 1. Retrieved from `http://jasss.soc.surrey.ac.uk/ 12/1/1.html`

Geanakoplos, J. (2003). Nash and Walras equilibrium via Brouwer. *Economic Theory*, *21*(2-3), 585–603. doi: 10.1007/s001990000076

Geanakoplos, J., Axtell, R., Farmer, D. J., Howitt, P., Conlee, B., Goldstein, J., ... Yang, C.-Y. (2012). Getting at systemic risk via an agent-based model of the housing market. *American Economic Review*, *102*(3), 53–58. doi: 10.1257/aer.102.3.53

Gilbert, N., & Troitzsch, K. (2005). *Simulation for the social scientist* (2nd ed.). Maidenhead, England: Open University Press.

Gode, D. K., & Sunder, S. (1993). Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of Political Economy*, *101*(1), 119–137.

Grimm, V., Berger, U., DeAngelis, D. L., Polhill, J. G., Giske, J., & Railsback, S. F. (2010). The ODD protocol: A review and first update. *Ecological Modelling*, *221*(23), 2760–2768. doi: 10.1016/j.ecolmodel.2010.08.019

Herlihy, M., & Shavit, N. (2012). *The art of multiprocessor programming* (Revised ed.). Amsterdam: Morgan Kaufmann.

Hirsch, M. D., Papadimitriou, C. H., & Vavasis, S. A. (1989). Exponential lower bounds for finding Brouwer fix points. *Journal of Complexity*, *5*(4), 379–416. doi: 10.1016/0885-064X(89)90017-4

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677. doi: 10.1145/359576.359585

Hoare, C. A. R. (1985). *Communicating sequential processes*. Englewood Cliffs, N.J: Prentice-Hall International.

Huberman, B. A., & Glance, N. S. (1993). Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*, *90*(16), 7716–7718.

Jackson, J., Forest, B., & Sengupta, R. (2008). Agent-based simulation of urban residential dynamics and land rent change in a gentrifying area of Boston. *Transactions in GIS*, *12*(4), 475–491. doi: 10.1111/j.1467-9671.2008.01109.x

Jaffry, S. W., & Treur, J. (2011). Modelling trust for communicating agents: Agent-based and population-based perspectives. In P. Jędrzejowicz, N. T. Nguyen, & K. Hoang (Eds.), *Computational Collective Intelligence. Technologies and Applications* (pp. 366–377). Berlin: Springer.

Jooybar, H., Fung, W. W., O'Connor, M., Devietti, J., & Aamodt, T. M. (2013). GPUDet: A deterministic GPU architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 1–12). Houston, TX: ACM Press. doi: 10.1145/2451116.2451118

Katzgraber, H. G. (2010). *Random numbers in scientific computing: An introduction.* (arXiv: 1005.4117v1 [physics.comp-ph])

Kindratenko, V. (Ed.). (2014). *Numerical computations with GPUs*. New York: Springer.

Laver, M., & Sergenti, E. (2012). *Party competition: An agent-based model*. Princeton: Princeton University Press.

Lee, E. A. (2006). The problem with threads. *Computer*, *39*(5), 33–42. doi: 10.1109/MC.2006.180

Lee, E. A. (2009). Computing needs time. *Communications of the ACM*, *52*(5), 70–79. doi: 10.1145/1506409.1506426

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. (2005). MASON: A multiagent simulation environment. *Simulation: Transactions of The Society for Modeling and Simulation International*, *81*(7), 517–527. doi: 10.1177/0037549705058073

Lytinen, S. L., & Railsback, S. F. (2012). The evolution of agent-based simulation platforms: A review of NetLogo 5.0 and ReLogo. In *Proceedings of the 21st European Meeting on Cybernetics and Systems Research.* Vienna: Bertalanffy Center for the Study of Systems. Retrieved from `http://www2.econ.iastate.edu/tesfatsi/NetLogoReLogoReview.LytinenRailsback2012.pdf`

Mathevet, R., Bousquet, F., Le Page, C., & Antona, M. (2003). Agent-based simulations of interactions between duck population, farming decisions and leasing of hunting rights in the Camargue (Southern France). *Ecological Modelling*, *165*(2-3), 107–126. doi: 10.1016/S0304-3800(03)00098-X

Matsumoto, M., & Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, *8*(1), 3–30. doi: 10.1145/272991.272995

Miller, J. H., & Page, S. E. (2007). *Complex adaptive systems: An introduction to computational models of social life*. Princeton, NJ: Princeton University Press.

Mitchell, M. (2011). *Complexity: A guided tour*. Oxford: Oxford University Press.

Mooij, W. M., Bennetts, R. E., Kitchens, W. M., & DeAngelis, D. L. (2002). Exploring the effect of drought extent and interval on the Florida snail kite: Interplay between spatial and temporal scales. *Ecological Modelling*, *149*(1-2), 25–39. doi: 10.1016/S0304-3800(01)00512-9

Newth, D., & Cornforth, D. (2009). Asynchronous spatial evolutionary games. *Biosystems*, *95*(2), 120–129. doi: 10.1016/j.biosystems.2008.09.003

North, M. J., Collier, N., & Murphy, J. T. (2016). *High performance agent-based modeling*. Boca Raton, FL: CRC Press.

North, M. J., Collier, N. T., & Vos, J. R. (2006). Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, *16*(1), 1–25. doi: 10.1145/1122012.1122013

Nowak, M. A., Bonhoeffer, S., & May, R. M. (1994). Spatial games and the maintenance of cooperation. *Proceedings of the National Academy of Sciences*, *91*(11), 4877–4881.

Nowak, M. A., & May, R. M. (1992). Evolutionary games and spatial chaos. *Nature*, *359*(6398), 826–829. doi: 10.1038/359826a0

Osgood, N. (2009). Lightening the performance burden of individual-based models through dimensional analysis and scale modeling. *System Dynamics Review*, *25*(2), 101–134. doi: 10.1002/sdr.417

Page, S. E. (1997). On incentives and updating in agent based models. *Computational Economics*, *10*(1), 67–87. doi: 10.1023/A:1008625524072

Papadimitriou, C. H. (1994). On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, *48*(3), 498–532. doi: 10.1016/S0022-0000(05)80063-7

Radax, W., & Rengs, B. (2010). Timing matters: Lessons from the CA literature on updating. In *Proceedings of the 3rd World Congress of Social Simulation.* Kassel, Germany. (arXiv: 1008.0941v1 [cs.MA])

Railsback, S. F., & Grimm, V. (2012). *Agent-based and individual-based modeling: A practical introduction*. Princeton, NJ: Princeton University Press.

Richardson, R., Richiardi, M., & Wolfson, M. (2015). *We ran one billion agents. Scaling in simulation models.* (Working Paper No. 142). Laboratorio Riccardo Revelli: Collegio Carlo Alberto. Retrieved 2015-10-04, from `http://www.laboratoriorevelli.it/_pdf/wp142.pdf`

Richmond, P., Coakley, S., & Romano, D. M. (2009). A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems* (Vol. 2, pp. 1125–1126). Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.

Rouly, O. C. (2015). *Towards emergent social complexity* (Ph.D. dissertation). George Mason University, Fairfax, VA.

Rousset, A., Herrmann, B., Lang, C., & Philippe, L. (2014). A survey on parallel and distributed multi-agent systems. In L. Lopes et al. (Eds.), *Euro-Par 2014: Parallel Processing Workshops* (pp. 371–382). Cham: Springer International Publishing.

Rubio-Campillo, X., Cela, J. M., & Cardona, F. X. H. (2013). The development of new infantry tactics during the early eighteenth century: A computer simulation approach to modern military history. *Journal of Simulation*, *7*(3), 170–182. doi: 10.1057/jos.2012.25

Schelling, T. C. (1971). Dynamic models of segregation. *The Journal of Mathematical Sociology*, *1*(2), 143–186. doi: 10.1080/0022250X.1971.9989794

Schönfisch, B., & de Roos, A. (1999). Synchronous and asynchronous updating in cellular automata. *Biosystems*, *51*(3), 123–143. doi: 10.1016/S0303-2647(99)00025-8

Simon, H. A. (1996). *The sciences of the artificial* (3rd ed.). Cambridge, Mass: MIT Press.

Sutcliffe, A. G., Wang, D., & Dunbar, R. I. M. (2015). Modelling the role of trust in social relationships. *ACM Transactions on Internet Technology*, *15*(4), 1–24. doi: 10.1145/2815620

Wilcox, A. R. (1973). Indices of qualitative variation and political measurement. *The Western Political Quarterly*, *26*(2), 325. doi: 10.2307/446831

Wilensky, U. (2015, April). *NetLogo.* Northwestern University: Center for Connected Learning and Computer-Based Modeling. Retrieved from `http://ccl.northwestern.edu/netlogo/`

Curriculum Vitae

Stefan D. McCabe received a Bachelor of Arts in Government and International Politics from George Mason University in 2013.