An Object-Oriented Parallel Social Simulation Framework

Mohammad H. Abuomar

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2015

Committee

Vikram Jandhyala

Arun Sathanur

Swagato Chakraborty

Program Authorized to Offer Degree:

Electrical Engineering

© Copyright 2015

Mohammad H. Abuomar

University of Washington

Abstract

An Object-Oriented Parallel Social Simulation Framework

Mohammad H. Abuomar

Chair of the Supervisory Committee:

Professor Vikram Jandhyala

Electrical Engineering

Agent-Based Modeling (ABM) is a powerful technique in social simulations that employs a bottom-up approach for opinion-dynamics study. In this work, we develop a social simulation framework based on ABM. The framework supports multiple features of interest to social scientists and experimenters including automatic parameter study, weighted and unweighted graphs support, and multiple output formats. We also showcase the framework by studying select social models from the literature. The framework allows us to explore the social models and understand similarities and differences among them all with little to no effort from the experiment designer.

TABLE OF CONTENTS

Chapter 1. Introduction 8

1.1 Agent-based modeling (abm) 8
1.2 benefits of abm 9
1.3 applications of abm 10
1.4 challenges in abm 10

Chapter 2. social interaction models 12

2.1 Opinion Dynamics Models 12
2.2 Classical model 13
2.3 FJ Model 13
2.4 Time-Variable Model 15
2.5 Bounded Confidence (bc) Model 15
2.6 Relative Agreement (RA) model 17
2.7 Qualitative Models 18

Chapter 3. Developed Social Simulation Framework 19

3.1 Motivation 19
3.1.1 Need for Fidelity in Modeling 19
3.1.2 Evolvement support 19
3.1.3 Automatic parameter exploration 20
3.1.4 Achieving separation of concerns 20
3.1.5 Being up-to-date 20
3.2 Features 21
3.2.1 Developed in Java 21
3.2.2 Parallel execution 21
3.2.3 Parameter study 21
3.2.4 Initial Conditions 21
3.2.5 Agent Interactions 22

3.2.6 Opinion Dynamics Models 22
3.2.7 Output Reporting 22
3.3 Architecture 23
3.4 Benchmarking 24
3.5 Parameter study 25
3.5.1 Constant parameter 25
3.5.2 Parameter sweep 26
3.5.3 Random Distribution 26
3.6 Agent Interactions 27
3.6.1 Random Interactions 27
3.6.2 Graph-Based Interactions 27
Chapter 4. Detailed example experiment 28
4.1 Relative Agreement in Depth 28
4.2 Experiment 29
4.2.1 Objective 29
4.2.2 Method 29
4.3 Experiment flow 30
4.3.1 Input specification 30
4.3.2 Starting experiment 32
4.3.3 High-level flow 33
4.3.4 Simulation Factory 33
4.3.5 Parameter Space Setup 34
4.3.6 Simulation 34
4.3.7 Output reporting 35
Chapter 5. Results 37
5.1 Relative agreement model 37
5.1.1 Population uncertainty and consensus 37
5.1.2 Number of iterations and consensus 39
5.1.3 Dynamics control and consensus 41

5.1.4 Initial opinion distribution and consensus ... 42
5.1.5 Population Uncertainty in graph-based experiments ... 45
5.2 Bounded Confidence Model ... 47
5.2.1 Effect of confidence range (ε) ... 48
5.2.2 Effect of initial opinion ... 50
5.3 FJ model ... 51
Chapter 6. Conclusions and future work ... 54
6.1 Conclusions ... 54
6.1.1 Relative weights similarity: ... 54
6.1.2 Susceptibility to change similarity: ... 54
6.2 Future work ... 55
Appendix: Code listing for core simulation framework ... 56

LIST OF FIGURES

Figure 1: Social Simulation Framework architecture. 23
Figure 2: The effect of number of threads on experiment time. 25
Figure 3: Agent categories according to opinion range 28
Figure 4: Measure of Extreme Opinion Spread as a function of Population Uncertainty and Proportion of Agents with Extreme Opinion. 36
Figure 5: Output of Relative Agreement Experiment with Population Uncertainty 10 Times Uncertainty of Agents with Extreme Opinion. 38
Figure 6: Output of for Relative Agreement Experiment with Equal Population and Agent with Extreme Opinion Uncertainty. 39
Figure 7: Output of Relative Agreement Experiment where all Agents have 0.1 as Uncertainty for 9 Iterations. 40
Figure 8: Output of Relative Agreement Experiment with Control Parameter=0.7. 42
Figure 9: Output of Relative Agreement Experiment with Gaussian(0.0, 0.2) Agent Initial opinion. 43
Figure 10: Output of Relative Agreement Experiment with Gaussian(0.3, 0.2) Agent Initial opinion. 44
Figure 11: Output of Graph-based Relative Agreement Experiment with Gaussian(0.8, 0.1) Population Uncertainty. 46
Figure 12: Output of Graph-based Relative Agreement Experiment with Gaussian(0.4, 0.1) Population Uncertainty. 47
Figure 13: Output of Bounded Confidence Experiment with confidence Range of 0.5... 48
Figure 14: Output of Bounded Confidence Experiment with confidence Range of 0.01. 49
Figure 15: Output of Bounded Confidence Experiment with Uniform(0,1) Agent Initial Opinion. 50
Figure 16: Output of FJ Experiment with Uniform(0.5, 0.7) Agent Susceptibility. 52
Figure 17: Output of FJ Experiment with Uniform(0.1, 0.3) Agent Susceptibility. 53

LIST OF TABLES

Table 1: Input specification for Relative Agreement Simulation 30
Table 2: Input Specification for Relative Agreement Experiment with Population Uncertainty 10 Times Uncertainty of Agents with Extreme Opinion. 37
Table 3: Input Specification for Relative Agreement Experiment with Equal Population and Agent with Extreme Opinion Uncertainty. 38
Table 4: Input Specification for Relative Agreement Experiment where all Agents have 0.1 as Uncertainty for 9 Iterations. 40
Table 5: Input specification for Relative Agreement Experiment with Control Parameter=0.7. 41
Table 6: Input Specification for Relative Agreement Experiment with Gaussian(0.0, 0.2) Agent Initial opinion. 43
Table 7: Input Specification for Relative Agreement Experiment with Gaussian(0.3, 0.2) Agent Initial opinion. 44
Table 8: Input Specification for Graph-based Relative Agreement Experiment with Gaussian(0.8, 0.1) Population Uncertainty. 45
Table 9: Input Specification for Graph-based Relative Agreement Experiment with Gaussian(0.4, 0.1) Population Uncertainty. 46
Table 10: Input Specification for Bounded Confidence Experiment with confidence Range of 0.5.. 48
Table 11: Input Specification for Bounded Confidence Experiment with confidence Range of 0.01.. 49
Table 12: Input Specification for Bounded Confidence Experiment with Uniform(0,1) Agent Initial Opinion. 50
Table 13: Input Specification for FJ Experiment with Uniform(0.5, 0.7) Agent Susceptibility. 51
Table 14: Input Specification for FJ Experiment with Uniform(0.1, 0.3) Agent Susceptibility. 52

ACKNOWLEDGEMENTS

I’d like to thank my wife and family who were very supportive and continuously encouraging. Also, I’d like to give very special thanks to Dr. Arun Sathanur for his guidance.

DEDICATION

To my grandfather who taught me how to be a man.

Chapter 1. INTRODUCTION

Agent-Based Modeling (ABM) is a powerful simulation technique that employs a bottom-up approach for system modeling. It starts by modeling the system being simulated in terms of smaller units called agents. Agents interact according to a representative model. The model should capture the properties of the system with high fidelity. As agents interact on the micro-level, system properties are monitored on the macro-level.

In this dissertation we present a powerful ABM social simulation framework that is built using Object-Oriented software design practices and also supports parallel processing. We will start by giving a brief introduction about ABM, its benefits, applications, and challenges in the introduction chapter. In Chapter two we talk about social interaction models. These are models that govern interactions between agents in a social simulations modeling interaction between humans in society. In chapter three we talk about the framework itself. We discuss our motivations for developing the framework, as well as supported features, and go over a detailed example illustrating how the framework works. In chapter four we go over results obtained by using the framework to study select social interaction models. Finally, we close by conclusions and future work.

1.1 AGENT-BASED MODELING (ABM)

ABM is a bottom-up approach for simulations. In this approach the system is described in terms of small units (agents). Agents are defined by properties capturing system related quantities. For example in a social system; agent typically has an opinion property representing

the agent's opinion on a given topic. Agents interact during the simulation according to a system-relevant dynamics model. Models are domain specific and vary in terms of complexity and applications. After interaction occurs an agent updates his properties according to the model. As agents update their properties, we monitor population level properties over time.

1.2 BENEFITS OF ABM

ABM technique is a powerful technique for simulations in general and social simulations in particular, due to the following:

- ***Natural System Description***: A community is described naturally in terms of the individuals comprising it according to predefined properties (by the interaction model).
- ***Enables studying emergent phenomena***: Agent properties are initialized at the beginning of the simulation along-side with model parameters, then as agents interact population-level properties are observed and emergent phenomena can be monitored.
- ***Extensibility***: Interaction model can be extended to incorporate new parameters and explore more behaviors without having to redevelop from scratch.
- ***Allowing heterogeneity***: Agents can interact according to different models (for example in different situations) within the same simulation framework provided a proper mapping exists between models.

1.3 APPLICATIONS OF ABM

ABM has many applications among which:

- ***Diffusion***: Agent behavior is influenced by surrounding environment (agents, population-level properties, as well as model parameters). For example; technology adoption, and elections modeling.
- ***Market study***: Modeling agent behavior in market situations like auctions, stock markets, etc.
- ***Organization study***: Study of operational risks, ream formation, and policy generation.
- ***Flow Dynamics***: Flow applications like traffic patterns.

1.4 CHALLENGES IN ABM

Although ABM has many benefits, it doesn't come for free. The following are a set of challenges that come up while using ABM.

- ***Model fidelity***: model must capture all study-relevant features. Overlooking a relevant feature might yield misleading results.
- ***Verification and validation***: result interpretation is not trivial and requires expertise with the model being simulated as well as crosschecking with other techniques studies like statistical results.
- ***Computational cost***: ABM simulations are more computationally demanding that population level techniques since population-level optimizations are not possible. For example; a linear model can take advantage from using linear algebra

optimization techniques like Singular Value Decomposition to simplify computations, while this is not possible in ABM.

- ***Skills requirements***: Designing a performing simulation framework requires expertise beyond that of a modeling scientist mainly related to effective programming, load balancing, and scalable software design.

Chapter 2. SOCIAL INTERACTION MODELS

In this chapter we talk about various social interaction models. We start by an overview about social interaction models and a high-level categorization. Then we discuss select models from the literature.

2.1 OPINION DYNAMICS MODELS

These models focus on studying opinion diffusion in social settings. Focus is given to agent properties change by interaction with other agents, and how that affects the overall population. For example we can study how opinion diffusion in a given population affect adoption of a certain new technology.

Social opinion dynamics models can be categorized into:

- ***Quantitative***: These models describe agent interactions using mathematical equations and model parameters. For example; when two agents interact there is an equation that describe the opinion change for each agent after the interaction.
- ***Qualitative***: These models describe agent interactions in terms of interaction specifications. For example; an agent might switch to adopt a given technology if all his connections adopt the technology.

Here we focus on quantitative models only.

2.2 CLASSICAL MODEL

Next state is a transformation from current state by a stochastic matrix A. It expresses the change in opinion using a linear mapping given by [Gilbert2005]:

$$x(t+1) = A.x(t)$$

$x(t+1)$: Opinion vector at time $t+1$.

$x(t)$: Opinion vector at time t .

A: Opinion mapping stochastic matrix; all elements are non-negative and the sum of each row is 1. This indicates that opinion at time $t+1$ is simply a weighted average of opinion in the previous step. In general opinion vector at time t_n can be given by:

$$x(t_n) = A^n.\ x(0)$$

2.3 FJ MODEL

This model is named after the inventors Friedken and Johnson [Friedken1990]. This model emphasizes the difference between two components of an agent's opinion. According to FJ; an agent opinion consists of self and influenced components. Self-opinion represents lack of susceptibility to other agents in the system, while influenced-opinion represents the agent's susceptibility to other agents. Mathematically, agent i opinion at time $t+1$ is given by:

$$x_i(t+1) = (1-\alpha_i)\ x_i(0) + \alpha_i \Sigma_j\ \omega_{ij}\ x_j(t)$$

- α_i: susceptibility to get influenced by other agents.
- $(1-\alpha_i)$: lack of susceptibility to other agents.
- ω_{ij}: weight of agent *j* over agent *i* (implying a connection from *j* to *i*).

We see that self-opinion is a function in lack of susceptibility to other agents multiplied by initial opinion and that it's constant. Influenced-opinion is a function of susceptibility to other agents multiplied by a weighted average of connected agents' opinions. The weight ω_{ij} is a connection parameter for the connection from *j* to *i* representing connection strength. In real life we can think of connection weight as the strength of a relationship between two entities. One important detail here is that the connection direction is from *j* to *i*, so the weight here is a representation of how much does agent *j* influence (affect) agent *i*. The difference between FJ model and the classical model is the emphasis on the two-component nature of agent opinion, and the corresponding susceptibility parameter.

The linear nature of FJ model makes it computationally efficient for matrix-based simulations. In [Sathanur2013] FJ model is used as the basis for online social network analysis. We also see commonalities between system representation in FJ model and Helmholtz Green Function in Electromagnetics.

A nonlinear variant of FJ model is represented in [Gabbay2012]. In this model system is represented by the equation:

$$\frac{dx_i}{dt} = -\gamma(x_i - \mu_i) + \sum_{j=1}^{N} \kappa_{ij}\, h(x_j - x_i)$$

The first term in the right side represents the self-bias force; γ model parameter representing agent commitment, x_i agent opinion, and μ_i agent natural bias. Second term represents group influence force; κ_{ij} is the coupling strength (similar to w_{ij}), and $h(x_j - x_i)$ is the coupling function (nonlinear equivalent to the linear summation proposed by FJ).

2.4 TIME-VARIABLE MODEL

$$x(t+1) = A(t).x(t)$$

Where $A(t)$ is time-dependent. This model is good for describing change of transformation as time goes; for example opinion hardening or softening over time (an agent becomes less or more affected by others as time goes).

Reaching consensus depends on the time variation of weights of A. If the variations reach a point where there are enough positive weights, consensus will be reached. In the case of opinion hardening consensus might not be reached at all [Hegselmann2002].

To the best of my knowledge there are not much publications for time-varying models in the context of opinion modeling and influence due to complexity. Time-varying models however are used extensively in financial research to model an agent's properties change over time; for example credit worth [Hegselmann2002].

2.5 BOUNDED CONFIDENCE (BC) MODEL

The Bounded Confidence (BC) model [Hegselmann2002] can be viewed as a restriction on FJ model in which an agent is only susceptible to opinions of a subset of agents defined by a confidence zone $I(i, x_i(t))$. Another view of this model is that an agent's opinion is the average of compatible opinions (defined by $I(i, x_i(t))$). The model can be represented by:

$$x_i(t+1) = \left(\frac{1}{\left|I\left(i, x_i(t)\right)\right|}\right).\Sigma\, x_j(t) \text{ } for \text{ } j \text{ } belongs \text{ } to \text{ } I(i, x_i(t))$$

The confidence zone can be symmetric; i.e. confidence interval is equal both ways from i to j and vice versa. The confidence zone is then represented as follows:

$$I(i, x_i(t)) = \{ 1 \leq j \leq n, |x_j - x_i| < \varepsilon_i \}$$

Uniform level of confidence is the special case and happens when $\varepsilon_i = \varepsilon$. The asymmetric case is when the confidence intervals are not of equal sizes. In this case the confidence zone is represented by:

$$I\big(i, x_i(t)\big) = \big\{ 1 \leq j \leq n, \quad \varepsilon_l \leq x_j - x_i \leq \varepsilon_r \big\} \, for \, \varepsilon_r \neq \varepsilon_l$$

Consensus in this model is reached when the confidence interval is equal to or greater than 40% of the overall opinion span. For example $x_i(t) \in [0,1]$, a confidence interval ε of 0.4 or larger will always lead to consensus.

One key feature for this model is that consensus is reached within finite number of steps if it will be reached. For the other cases, local consensus can be reached within the consensus intervals of each agent group belonging to a set $I(i, x_i(t))$. These intervals represent clusters of agents with the same opinion (local consensus).

The original BC model allows opinion to take any real value. A discretized version of BC is proposed in [Furtunato2004]. The proposal quantizes possible opinions $x[i] \in \{1, 2, \ldots., Q\}$. Using this model on two types of topology; one in which an agent can connect with any agent within the confidence zone, and a scale-free Barabasi-Albert Network, author found that there is always a corner value Q_c such that $Q < Q_c$ will always result in consensus.

Similar models were proposed by Deffuant et al [Deffuant2000] and Dittmer [Dittmer2001]. A mathematical proof of BC model equilibrium in both the global and interval consensus cases can be found in [Blondel2009].

2.6 RELATIVE AGREEMENT (RA) MODEL

The Relative Agreement (RA) model is an extension to the BC model [Deffuant2002]. In RA model two interacting agents are affected by each other's opinion as well as uncertainty. A key change in RA is that after interaction not only is the opinion updated, but also the uncertainty. The influence of agent i on agent j (agreement) is directly proportional to opinion overlap, h_{ij}, and negatively proportional to agent $i's$ uncertainty u_i. After the interaction agent $j's$ opinion and uncertainty are updated as follows:

$$x_j(t+1) = x_j(t) + \mu \left(\frac{h_{ij}(t)}{u_i(t)} - 1 \right) (x_i(t) - x_j(t))$$

$$u_j(t+1) = u_j(t) + \mu \left(\frac{h_{ij}(t)}{u_i(t)} - 1 \right) (u_i(t) - u_j(t))$$

The opinion overlap $h_{ij}(t)$ is obtained by:

$$h_{ij}(t) = \min\big(x_i(t) + u_i(t)\,, x_j(t) + u_j(t)\big) - \max(x_i(t) - u_i(t)\,, x_j(t) - u_j(t))$$

μ is the model parameter which amplitude controlling the speed of the dynamics.

According to this model, if $h_{ij}(t) \leq u_i(t)$ i will have no influence on j. Unlike BC model, in RA model, influence (change in opinion and uncertainty) is continuous. In BC model, once an agent falls out of agreement interval, his opinion is no longer influential. Convergence under this model follows similar patterns as in the BC, with different parameter space. This model is also examined in details, including a java-based implementation in [Meadows2012].

The original model introduced by Deffuant allows global agent interaction in which any two agents can interact with equal probability. In [Meadows2013] analysis of RA model in more complex types of network topologies was introduced. [Deffuant2005] used RA to model innovation diffusion and its dynamic properties. The effect of adding noise, modeling free will,

is explored in [Pineda2009]. Free will was modeled as the probability of an agent picking a different opinion from the opinion space and not be affected by the RA model equations.

2.7 QUALITATIVE MODELS

These models depend on autonomous agents. An agent has a state, a view of the surrounding world (other agents), and updates both state and view by interactions with other agents in the system [Gilbert2005]. During interactions an agent will have a strategy to maximize gain.

In PsychSim [Marsell2004] each agent has a state of mind representation as well as an update mechanism through interaction with other agents. State of mind is represented by a decision-theoretic view of the world providing beliefs about the environment and models of other agents. PsychSim provides a descriptive framework for agent-based simulations in which agents can be defined by providing their state of mind and update mechanisms. We won't cover the implementation details of such system, since we're more concerned about quantitative models.

Next Chapter, we'll discuss the architecture of the developed social simulation framework. We'll also give an overview of the supported features and experiment flows.

Chapter 3. DEVELOPED SOCIAL SIMULATION FRAMEWORK

In this chapter we'll discuss the developed social simulation framework. We will also discuss features and supported experiment flows.

3.1 MOTIVATION

3.1.1 *Need for Fidelity in Modeling*

An expressive model dynamics language is essential for describing social interaction models. Existing simulation toolkits require learning a limited vocabulary simulation language that the tool makers developed. These languages often make it harder to express rich models with high fidelity. On the other hand general programming languages such as Java are the most rich and agile. They are constantly developed and improved both in terms of expressiveness and performance.

3.1.2 *Evolvement support*

A social simulation framework is expected to make it easy to evolve the model. Adding new parameters, and studying parameter-space in more scenarios and combinations should be allowed to evolve and become richer. At the same time the framework itself should be extensible supporting new models, new output formats, and easier interfacing.

3.1.3 *Automatic parameter exploration*

Parameter study including parameter sweep, random distribution specification, Monte Carlo Simulations, and their combinations should be supported with almost no manual steps. For example we should be able to specify model parameters, assign some of them to be constant, sweep other parameters in a specified range, and define others using a Random distribution.

3.1.4 *Achieving separation of concerns*

Separation of different simulation aspects using clear interfaces and definitions is a very desirable feature. Different aspects of a social simulation framework:

- Input representation and parsing; e.g. Graph representation and parsing.
- Experiment setup and parameter specification.
- Experiment execution.
- Output representation.

3.1.5 *Being up-to-date*

Computational tools are constantly developing. Performance enhancements, software bug fixes, and security flaws remedies are parts of almost every upgrade to any modern computational framework. A good social simulation framework, must be up-to-date in terms of the computational tools used and the performance achieved.

3.2 FEATURES

Features are added to support the use-cases in our motivation to develop the framework including.

3.2.1 *Developed in Java*

Java is a very rich general programming language that empowers a wide range of applications. The language is continuously developing to address all development needs. It features a very rich vocabulary allowing specification of social models with high fidelity.

3.2.2 *Parallel execution*

Framework supports executing multiple simulations concurrently. The number of concurrent simulations can be tuned to allowing maximizing resource utilization.

3.2.3 *Parameter study*

Each parameter can be independently set to constant, sweep, or random distribution. Parameter sweep is accomplished by uniform sampling over a specified range. Random distributions can be specified to be any of: Uniform, Exponential, Gaussian, or Log-Normal distributions.

3.2.4 *Initial Conditions*

Experiment consists of multiple simulations. In each simulation initial conditions are set by specifying a random distribution for the initial condition parameter. For example: agent opinions can be set to values using any of the supported random distributions.

3.2.5 *Agent Interactions*

Two types of agent interactions are supported. In random interactions; all agents are allowed to interact in random order with equal probability. In graph-based interactions; only connected agents are allowed to interact. Input graphs can be weighted or not.

3.2.6 *Opinion Dynamics Models*

Currently the following models are supported, support can be added for more models as needed easily.

- Relative Agreement (RA),
- Bounded Confidence (BC),
- FJ

3.2.7 *Output Reporting*

Output can be exported in the following formats to a CSV file making it more interoperable because of the wide-support of CSV.

- Linear representation: inputs vs. outputs for each simulation. Allows exporting multi-dimensional parameters and outputs.
- 2D representation for two-parameter studies.
- Output histograms.
- Time series representation: parameter values for all agents over time.

3.3 ARCHITECTURE

The following diagram shows the overall architecture of the simulation framework

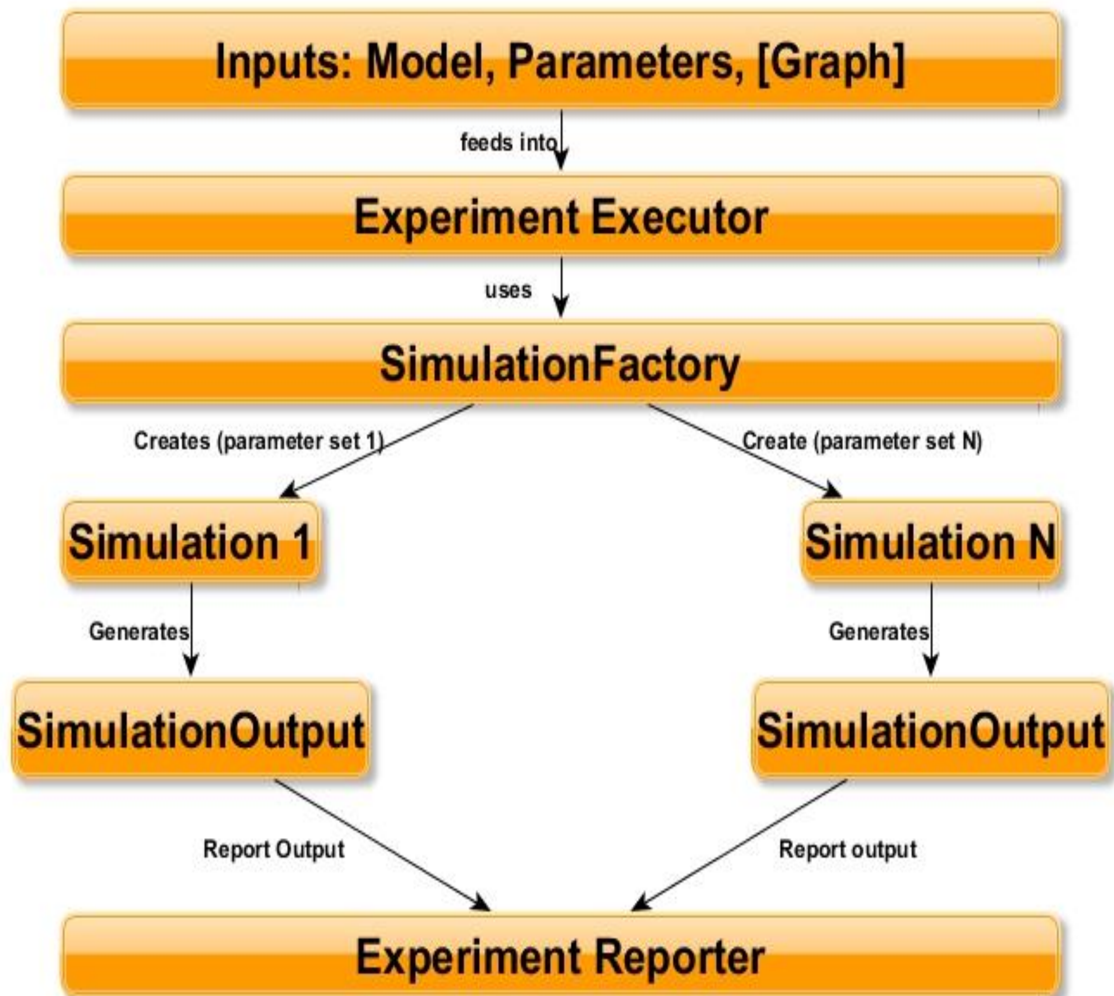

Figure 1: Social Simulation Framework architecture.

Experiments start by specifying inputs including; model being studied, model parameters, and network graph for graph-based simulations. These inputs are then fed into an experiment executor, which orchestrates the flow of the experiment. The Experiment Executor uses a Simulation Factory that generates all simulation units needed to complete the experiment. Each simulation runs in its own thread and the number of concurrent simulations is configurable to

maximize resource utilization. Experiment flow is discussed in more details in the next chapter. In the remaining sections of this chapter we discuss some of the aspects of the framework at a high level and provide benchmarking information.

3.4 BENCHMARKING

The framework was benchmarked against a published java-based implementation [Meadows2012]. Both the published Java-based implementation and the framework performed a Relative Agreement experiment consisting of 2,500 simulations. The framework finished in less than 23 seconds on average, while the other implementation finished in 300 seconds on average. We see almost 15x performance improvement. This improvement can be attributed to restructuring the experiment into smaller units that can be performed in parallel. Also memory usage optimization, by removing objects that take longer time to traverse without adding much information contributed to the speedup.

To study the effect of the number of concurrent simulations we performed the same experiment on a 4 processing-core machine with different number of threads and recorded the experiment time for each setting. As we can see in Figure 2; we don’t gain any speedup beyond 4 threads. This corresponds to the total number of logical processing cores on the machine. This finding goes with the intuition that adding more threads beyond what the hardware can physically parallelize makes no difference in performance. In Practice increasing the number of threads too much beyond what the hardware can support actually results in performance degradation because of the overhead for managing different thread contexts without extra gain from hardware to support the extra threads.

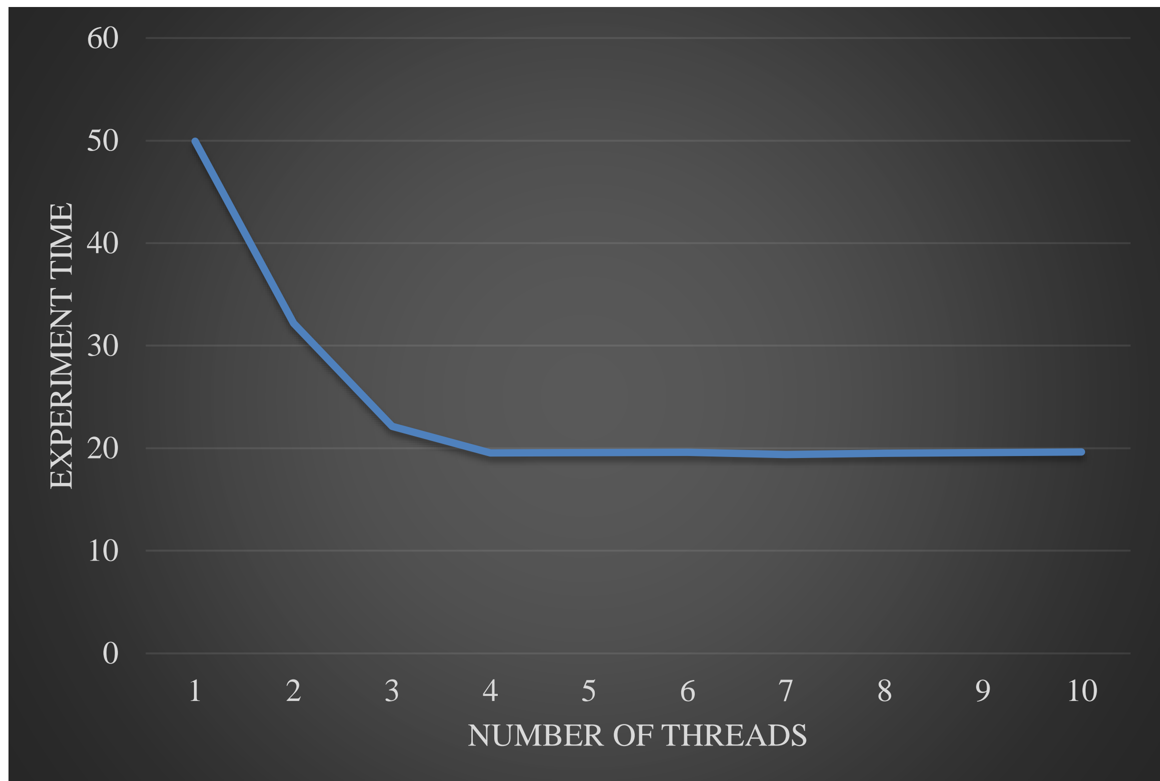

Figure 2: The effect of number of threads on experiment time.

3.5 Parameter study

3.5.1 *Constant parameter*

These parameters remain constant throughout the experiment. Typically this parameter configuration is used when we don't want to study the effect of varying a particular parameter, or when we want to study other parameter varying effects while the particular constant parameter has a fixed value. An example of a constant parameter can be the Dynamics Control parameter in the Relative Agreement model. Typically experiments are performed with a constant value for this parameter.

3.5.2 *Parameter sweep*

In this parameter configuration we're interested in studying a uniform sample of the parameter within a given range. Sampling rate is fixed and a Min, Max, and Number of Steps define the range. For example to study the effect of Confidence Range in the Bounded Confidence model between 0.01 and 2.0, we can sweep the Confidence range parameter with Min = 0.01, and Max = 2.0. Number of Steps in the sweep defines the resolution of study. If we're interested in capturing a highly non-linear parameter increasing the Number of Steps is advisable because it can show properties that might be missed with a lower resolution. On the other hand for linear parameters where there is a clear trend extra resolution won't provide much insight into the parameter effects on the model.

3.5.3 *Random Distribution*

This configuration is very useful in Monte Carlo simulations where we want to study the effect of a particular parameter profile rather than actual value. The profile is described by means of a random distribution, and we can also control the number of samples. For example; we can study the effect of a Gaussian profile for Confidence Range parameter in the Bounded Confidence model by using 1000 samples of a Gaussian distribution with Mean = 0.0, and Standard Deviation = 0.3. We can also use the random distributions to specify initial conditions for example agent opinions in any of the social dynamics models discussed. Currently the framework supports Uniform, Exponential, Gaussian, and Lognormal random distributions.

3.6 AGENT INTERACTIONS

3.6.1 *Random Interactions*

This mode of interaction is used for models that don't require an explicit connection between sets of agents. We specify the number of agents as an input to the experiment. Agent initial conditions are specified using a random distribution as discussed earlier, then agents are allowed to interact randomly with equal probability.

3.6.2 *Graph-Based Interactions*

In this mode a graph is specified as an input to the experiment. The graph can be weighted or unweighted depending on the model requirements. For example; Bounded Confidence model doesn't have a parameter to describe connections between agents and hence weight specification can't be used. On the other hand, FJ model has a weight parameter describing connection between any two connected agents and hence and weight is required. During the experiment an agent can only interact with his direct connections. In this mode of interaction agent influence propagates to the rest of the network through connections over time.

In the next chapter we'll discuss a detailed example for experiment flow and show how the framework performs the experiment.

Chapter 4. DETAILED EXAMPLE EXPERIMENT

In this chapter we'll discuss a detailed Relative Agreement Experiment. This experiment was introduced in [Deffuant2002] and re-implemented in [Meadows2012]. We show the experiment flow and the operation undertaken by the framework to perform in parallel and optimized fashion.

4.1 RELATIVE AGREEMENT IN DEPTH

The experiment we discuss here was first introduced in [Deffuant2002]. The main objective is to measure spread of extreme opinion among population based on Relative Agreement model.

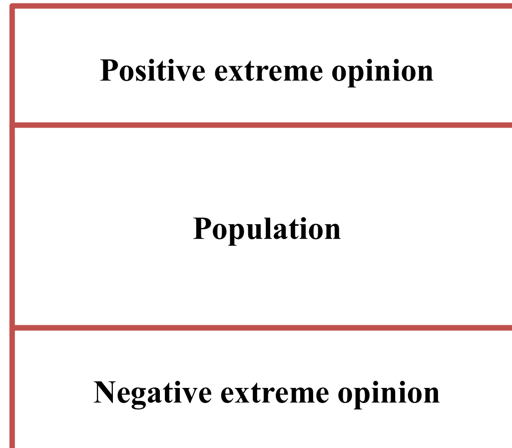

Figure 3: Agent categories according to opinion range

Deffuant et al in categorize people in a community (agents in the population) into three categories according to their opinion Range. The center category represents agents with moderate opinion range. This category represents population agents. Extreme opinion ranges in

both positive and negative ends of the opinion range represent the extreme categories. These correspond to the extreme right and extreme left in a community. Agents in the population range are characterized by higher uncertainty that those in the extreme range.

Measure of extreme opinion spread is determined using the percentage of agents who flipped from moderate to either of the extreme opinion ranges. For a percentage of agents converting to the positive extreme region: p'^2_+ and a percentage of agents converting to the negative extreme region: p'^2_- the Measure of Extreme opinion Spread can be expressed as:

$$y = p'^2_+ + p'^2_-$$

A value for y greater than 0 indicates extreme opinion spread.

4.2 EXPERIMENT

4.2.1 *Objective*

Study the relationship between population uncertainty and proportion of agents with extreme opinion in a community.

4.2.2 *Method*

- A population of 200 agents interacting randomly.
- Parameter sweep of population uncertainty from 0 to 2.0 with 50 steps.
- Parameter sweep of proportion of agents with extreme opinion from 0.0 to 0.3 with 50 steps.
- Report the measure of extreme opinion spread for each simulation.

4.3 EXPERIMENT FLOW

In this section we discuss how the framework operates starting from input specification and ending with output representation.

4.3.1 *Input specification*

Table 1: Input specification for Relative Agreement Simulation

Number of Agents	200
Initial Opinion Distribution	Uniform Distribution between -1 and 1.
Population Uncertainty	Sweep; Min:0.0, Max:2.0, 50 steps
Proportion of agents with Extreme Opinion	Sweep; Min: 0.0, Max:0.3, 50 steps
Population opinion range	Constant (0.8, -0.8)
Uncertainty of agents with extreme opinion	Constant 0.1
Control (Mu)	0.2
Number of Samples	50
Output	Measure of extreme opinion spread for each combination of population uncertainty and proportion of agents with extreme opinion

Input is specified according to parameters in table 1. This input is translated into code according to the listings below.

```
// Initialization

Set<SimulationParameter> params = new
HashSet<SimulationParameter>();

// Constant Parameters

params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.MaxOpinion, 1.0)); // MinOpinion = -1 * MaxOpinion

params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.NumberOfSamples, 50.0));
params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.ExtremeOpinionBoundary, 0.8)); // between 0.8 and -
0.8

params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.ExtremeOpinionAgentUncertainty, 0.1));

// Constant parameters Continued

params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.MU, 0.2));

params.add(SimulationParameter.newConstantParameter(SimulationPa
rameterName.NumberOfAgents, 200.0));

// Varying parameters.

params.add(SimulationParameter.newVaryingParemter(SimulationPara
meterName.PopulationUncertainty, 0.0, 2.0, 50.0));

params.add(SimulationParameter.newVaryingParemter(SimulationPara
meterName.ExtremeOpinionAgentProportion, 0.0, 0.3,50.0));

// Initial conditions - Uniform agent opinion

RandomDistributionGeneratorFactory factory = new
RandomDistributionGeneratorFactory() {

     @Override

     public RandomDistributionGenerator getGenerator() {

          return new UniformRandomGenerator (-1.0, 1.0);

     }

};

params.add(SimulationParameter.newAgentDistributionParameter(Sim
ulationParameterName.AgentOpinionGeneratorFactory, factory));
```

```
// Initial conditions - Uniform agent opinion
RandomDistributionGeneratorFactory factory = new
RandomDistributionGeneratorFactory() {

     @Override

     public RandomDistributionGenerator getGenerator() {

          return new UniformRandomGenerator (-1.0, 1.0);

     }

};

params.add(SimulationParameter.newAgentDistributionParameter(Sim
ulationParameterName.AgentOpinionGeneratorFactory, factory));

// Output format: 50x50 Matrix with rows corresponding to
Population Uncertainty and columns corresponding to Proportion
of agents with extreme opinion.

ExperimentReporter reporter = new
MatrixReporter(SimulationParameterName.PopulationUncertainty,
50, SimulationParameterName.ExtremeOpinionAgentProportion, 50,
“outputFileName.csv”);
```

4.3.2 *Starting experiment*

After inputs are specified, the experiment is started using the following listing:

```
// Start Relative Agreement experiment using input parameters,
output reporter, and allow 4 concurrent simulations.

// Executor is generic: same code for all types of simulations,
simulation type is specified at initialization time.

ExperimentExecutor<RelativeAgreementSimulation> executor = new
ExperimentExecutor<RelativeAgreementSimulation>(RelativeAgreemen
tSimulation.class, params, reporter, 4);

executor.execute();
```

After starting the experiment nothing is required from experiment designer and simulation framework takes care of performing the experiment and exporting the output.

4.3.3 *High-level flow*

The Experiment Executor handles the high-level flow of the experiment and the listing below specifies it:

```
// Thread pool to run using input number of concurrent
simulations.
ExecutorService threadPool =
Executors.newFixedThreadPool(concurrentSimulationsNum);
SimulationFactory<T> factory = new
SimulationFactory<T>(modelClass, inputParameters, reporter);
// Execute all simulations.
Simulation sim;
while ((sim = factory.create()) != null) {
     pool.execute(sim); // Only concurrentSimulationsNum is
     running at any given time.
}
// Wait for all simulations to finish
pool.shutdown();
// report output
reporter.createReport();
```

4.3.4 *Simulation Factory*

Simulation Factory plays an important role in the experiment. It's responsible for creating all simulations in the experiment. Simulation represents the experiment building block within which all model parameters are constant. In order to create simulations the Simulation Factory prepares the parameter space and takes care of sampling it assigning a fixed set of parameters to a single simulation until parameter space is exhausted. It's also responsible for creating graph objects by parsing input graph files. Finally, it links a simulation to an experiment reporter by passing a reference of the Experiment Reporter to each created Simulation.

4.3.5 *Parameter Space Setup*

Parameter space is prepared from varying parameters in two steps:

1. Varying parameters (sweep, random distribution) expansion:
    - Sweep (population uncertainty, Proportion of agents with extreme opinion): create a list of values starting with Min, and ending at Max with the provided number of steps.
    - Random Distribution (None for this example): List values with the specified number of samples from input Random Distribution.
2. Space is generated by Cartesian product of all Varying parameters (50x50=2500).

4.3.6 *Simulation*

Simulation Factory prepares parameters by using a single sample from the varying parameter space and attaching al constant parameters to the sample, simulation parameter set is created. Simulation parameters alongside with Experiment reporter are passed to the created Simulation object. Simulation types are specified in an object hierarchy. Behavior common to all simulations independent of models is shared in a Simulation abstract type. These behaviors include parsing common parameters, starting simulation, reporting output, etc. Each social interaction model encapsulates its own logic in a special sub-type. For example we have a special type for the RelativeAgreementSimulation. Each model simulation also uses its own agent type. For example we have RelativeAgreementAgent.

At the beginning of the simulation agents are initialized using input random distribution for the initial conditions. In this case we have a Uniform distribution for agent opinion between +/- 1.0. The number of samples input is used to remove the effect of specific values on simulation output while keeping the trend. In this experiment we have 50 samples. This means

that each simulation will be performed 50 times and the final output is the average of the output from the 50 runs. After final output is obtained, simulation reports the output to the Experiment Reporter. At any given point in time we have 4 concurrent simulations according to the input Number of Threads. Note that this number is configurable. The value 4 is used for this experiment to utilize the 4 logical processor on the machine used for the experiment.

4.3.7 *Output reporting*

Experiment Reporter collects the output of each simulation as they finish execution. At the end of the experiment when all simulations finished the final output is exported to a CSV file. The 2D output required for this experiment is exported using a Matrix reporter that outputs values for measure of extreme opinion spread for each combination of population uncertainty and proportion of extreme opinion spread.

By examining the output in figure 4; we see that in a certain range where population uncertainty is relatively low, and proportion of agents with extreme opinion is also low extreme opinion isn't spread. We can also see that beyond a population Uncertainty of about 0.3, the proportion of agents with extreme opinion controls the extreme opinion spread measure. As expected the higher the proportion the more likely to see extreme opinion spreading.

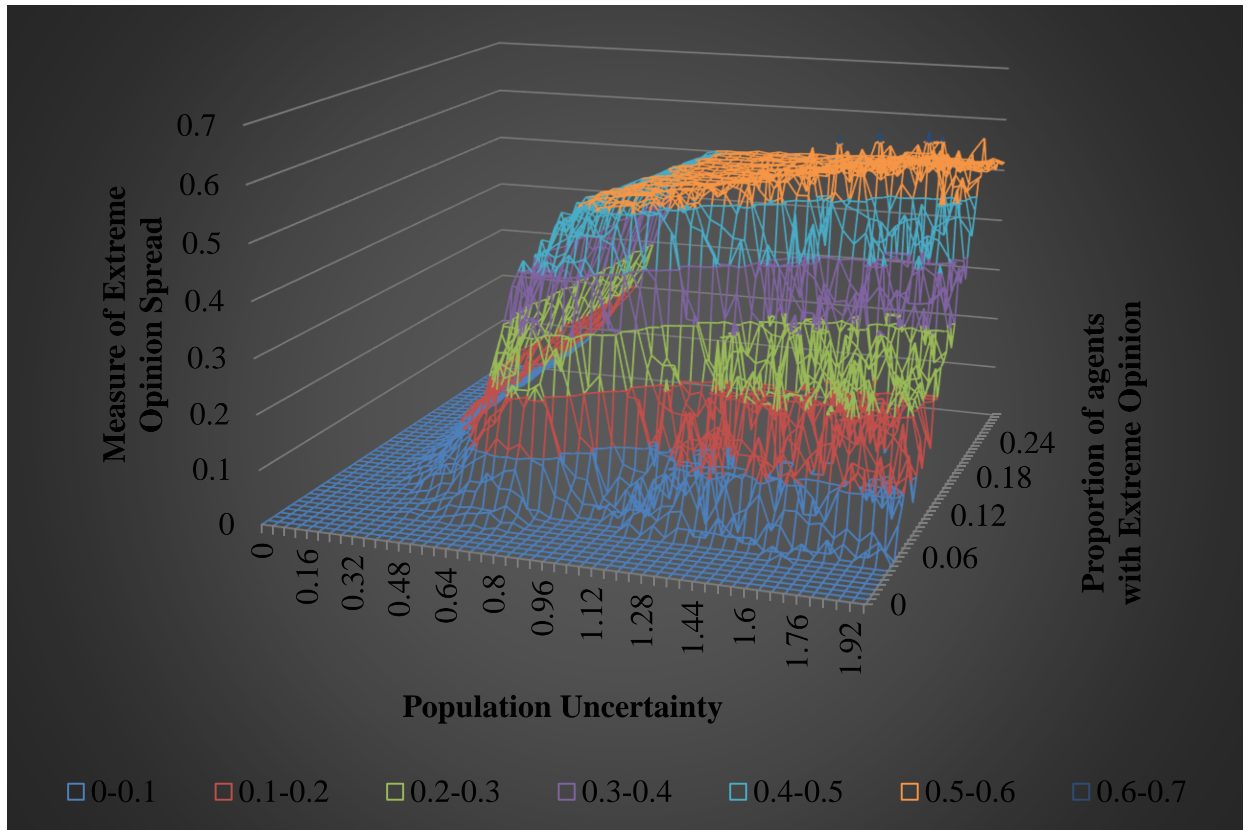

Figure 4: Measure of Extreme Opinion Spread as a function of Population Uncertainty and Proportion of Agents with Extreme Opinion.

In the next chapter we show output of various experiments performed using the developed social simulation framework.

Chapter 5. RESULTS

In this chapter we show results of studying Relative Agreement, Bounded Confidence, and FJ social interaction models. For each model we study the effect of model parameters as well as initial conditions.

5.1 RELATIVE AGREEMENT MODEL

5.1.1 *Population uncertainty and consensus*

To show the effect of population uncertainty on consensus we perform two experiments one where population uncertainty is 10 times the uncertainty of agents with extreme opinion, and another where the uncertainty is the same between all agents. We notice that when population uncertainty is high, consensus (defined as opinion convergence) is reach with two groups one if each extreme opinion range. In the case of equal uncertainty we see multiple groups forming.

Table 2: Input Specification for Relative Agreement Experiment with Population Uncertainty 10 Times Uncertainty of Agents with Extreme Opinion.

Agents	200
Initial Opinion Distribution	Uniform(-0.99, 0.99)
Population opinion range	(0.8, -0.8)
Population Uncertainty	1.0
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.2
Iterations	3

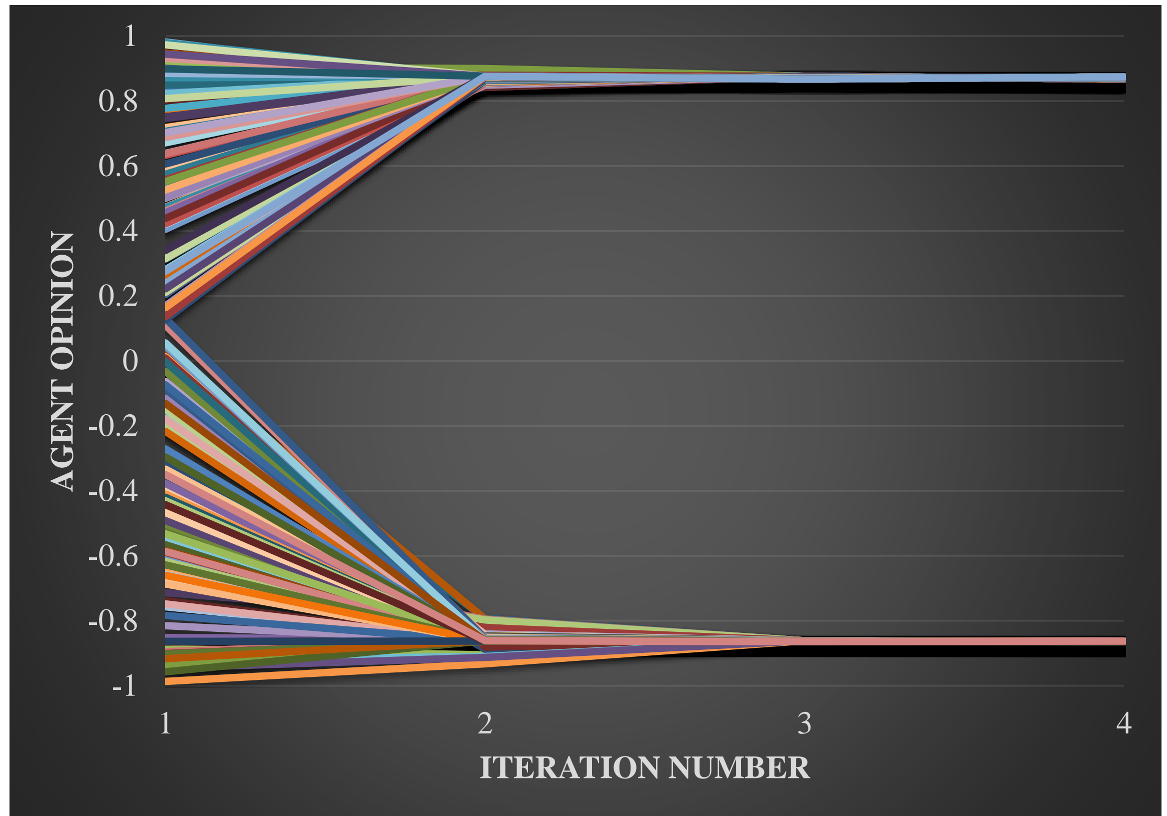

Figure 5: Output of Relative Agreement Experiment with Population Uncertainty 10 Times Uncertainty of Agents with Extreme Opinion.

Table 3: Input Specification for Relative Agreement Experiment with Equal Population and Agent with Extreme Opinion Uncertainty.

Agents	200
Initial Opinion Distribution	Uniform(-0.99, 0.99)
Population opinion range	(0.8, -0.8)
Population Uncertainty	0.1
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.2
Iterations	3

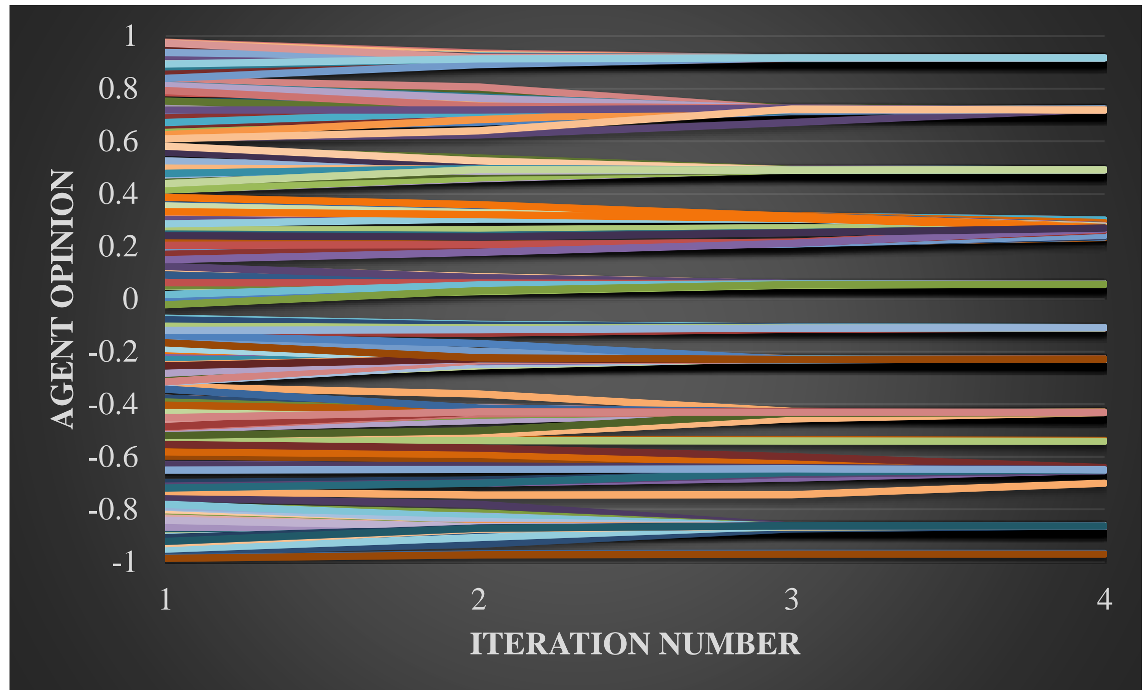

Figure 6: Output of for Relative Agreement Experiment with Equal Population and Agent with Extreme Opinion Uncertainty.

5.1.2 *Number of iterations and consensus*

In this experiment we study the effect of more interactions in the same population where convergence was reached at a high number of groups. The output shows that by allowing agents to interact for longer time, we see some groups merge and form less number of groups. However, after a certain time threshold more interactions don’t yield any new merges. This experiment was performed to model convergence in opinion during meetings according to the Relative Agreement Model. Although some groups merge after the meeting runs for longer time, consensus is not reached.

Table 4: Input Specification for Relative Agreement Experiment where all Agents have 0.1 as Uncertainty for 9 Iterations.

Agents	200
Initial Opinion Distribution	Uniform(-0.99, 0.99)
Population opinion range	(0.8, -0.8)
Population Uncertainty	0.1
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.2
Iterations	9

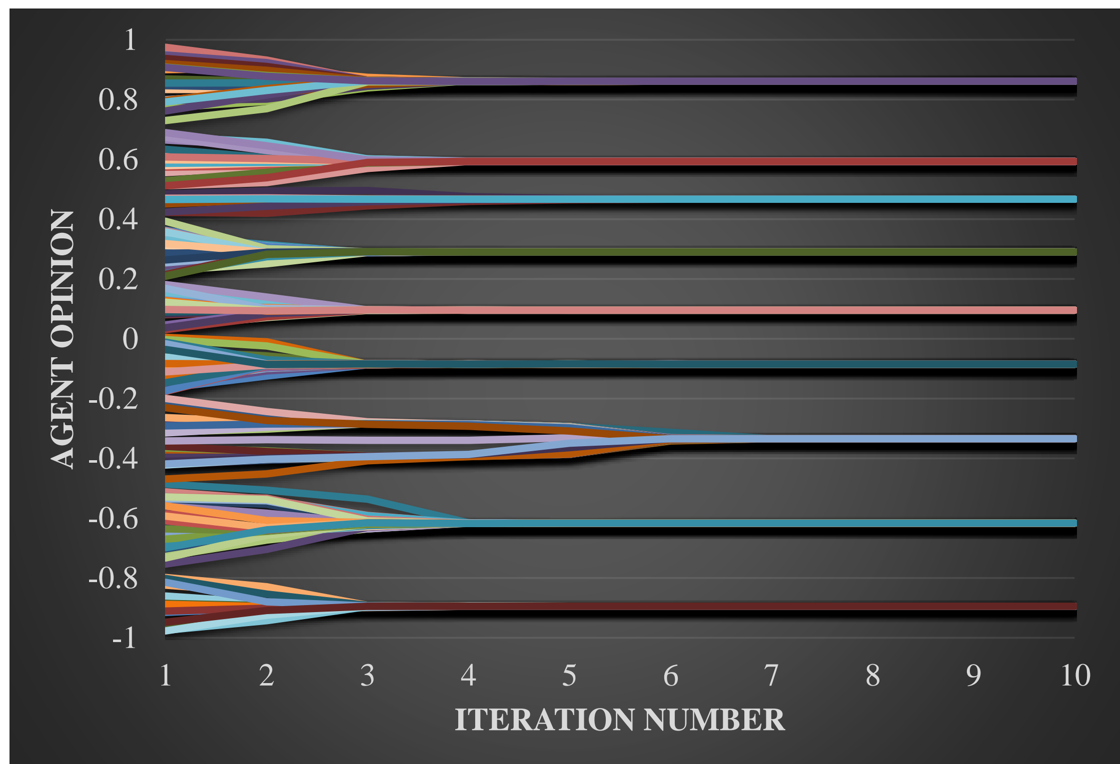

Figure 7: Output of Relative Agreement Experiment where all Agents have 0.1 as Uncertainty for 9 Iterations.

5.1.3 *Dynamics control and consensus*

In this experiment we explore the effect of high dynamics control parameter on reaching consensus. Dynamic control parameter corresponds to the amplitude of model progression. For example; an online social network with active agents who are very engaged control dynamics parameter is higher than a similar network where agents aren't as involved.

We see that when Control dynamics is higher consensus groups are formed faster. We also see that in situations where uncertainty is equal for all agents this means larger number of groups are formed at steady state because the gap increases faster preventing chances of merge.

Table 5: Input specification for Relative Agreement Experiment with Control Parameter=0.7.

Agents	200
Initial Opinion Distribution	Uniform(-0.99, 0.99)
Population opinion range	(0.8, -0.8)
Population Uncertainty	0.1
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.7
Iterations	4

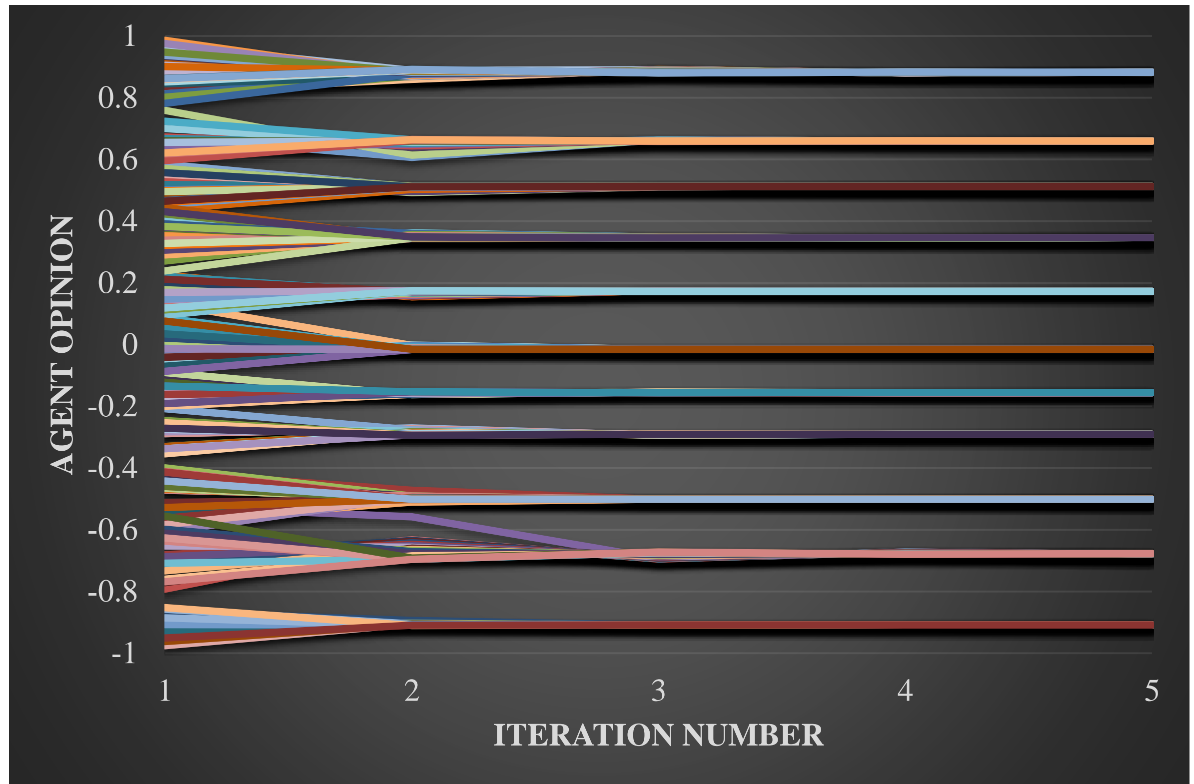

Figure 8: Output of Relative Agreement Experiment with Control Parameter=0.7.

5.1.4 *Initial opinion distribution and consensus*

To study the relationship between initial opinion distribution and final consensus when consensus is high for all population. We find that a single consensus group is formed at steady-state. We also find that the opinion of the formed group is the mean of the input initial distribution.

Table 6: Input Specification for Relative Agreement Experiment with Gaussian(0.0, 0.2) Agent Initial opinion.

Agents	200
Initial Opinion Distribution	Gaussian(0.0, 0.2)
Population opinion range	(0.8, -0.8)
Population Uncertainty	1.0
Uncertainty of agents with extreme opinion	1.0
Control (Mu)	0.2
Iterations	3

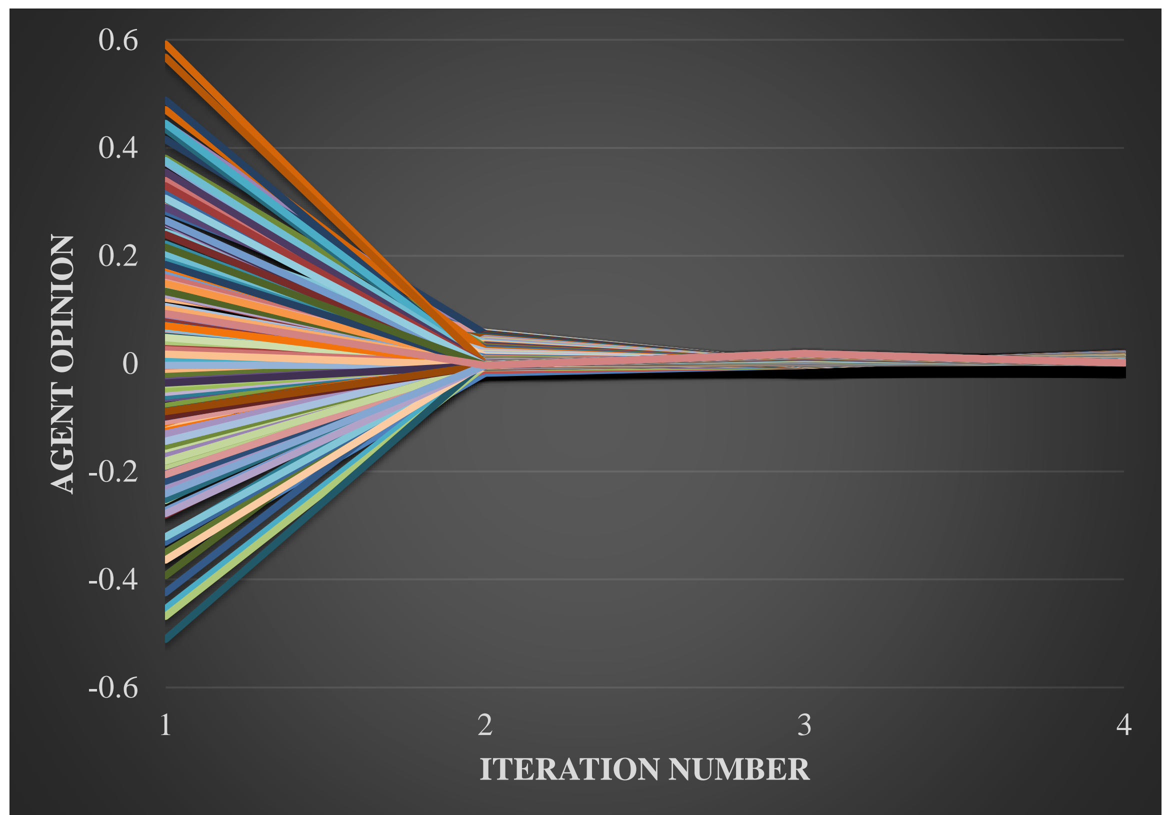

Figure 9: Output of Relative Agreement Experiment with Gaussian(0.0, 0.2) Agent Initial opinion.

Table 7: Input Specification for Relative Agreement Experiment with Gaussian(0.3, 0.2) Agent Initial opinion.

Agents	200
Initial Opinion Distribution	Gaussian(0.0, 0.2)
Population opinion range	(0.8, -0.8)
Population Uncertainty	1.0
Uncertainty of agents with extreme opinion	1.0
Control (Mu)	0.2
Iterations	3

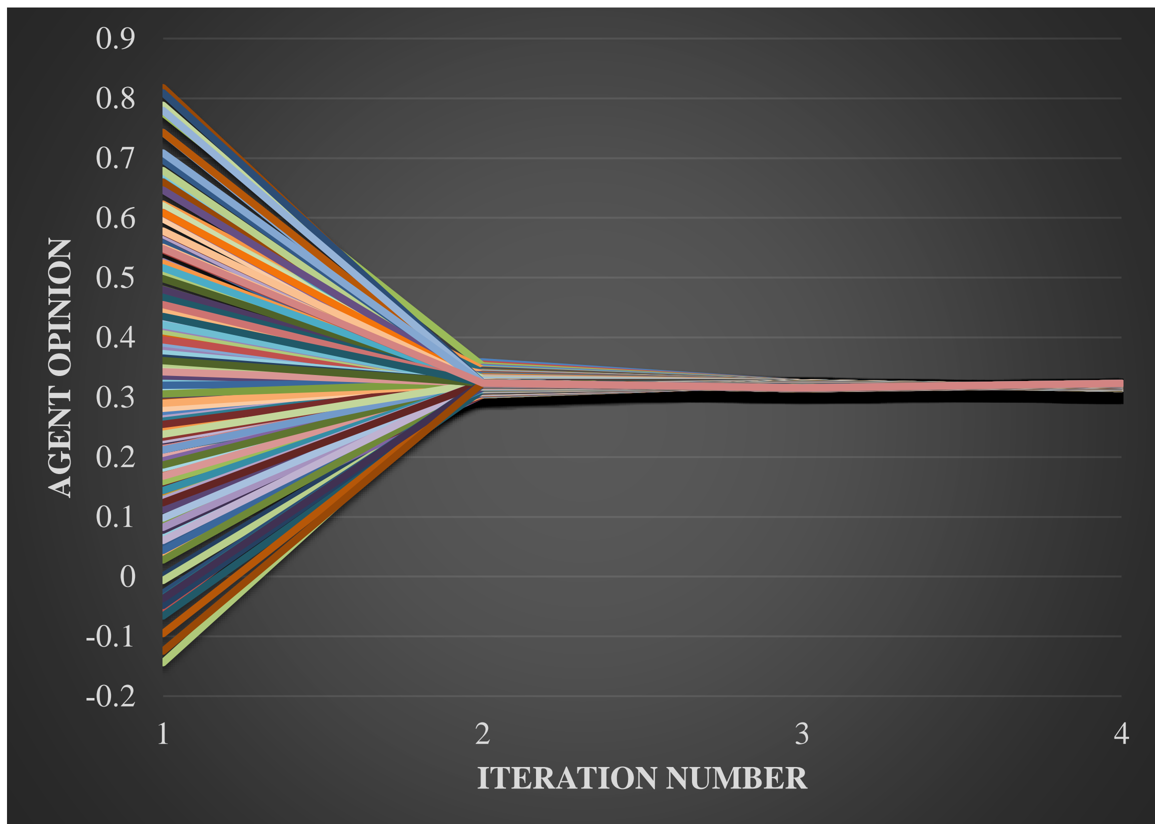

Figure 10: Output of Relative Agreement Experiment with Gaussian(0.3, 0.2) Agent Initial opinion.

5.1.5 *Population Uncertainty in graph-based experiments*

We study the effect of initial condition of population uncertainty in graph-based simulations. The graph used here is obtained from [SNAP2008]. We see that with higher ranges of population uncertainty the probability of spread of extreme opinion spread is higher.

Table 8: Input Specification for Graph-based Relative Agreement Experiment with Gaussian(0.8, 0.1) Population Uncertainty.

Graph Source	SNAP.
Agents	7,115
Initial Opinion Distribution	Uniform(-1, 1)
Population Uncertainty	Gaussian(0.8, 0.1), 1000 samples
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.2

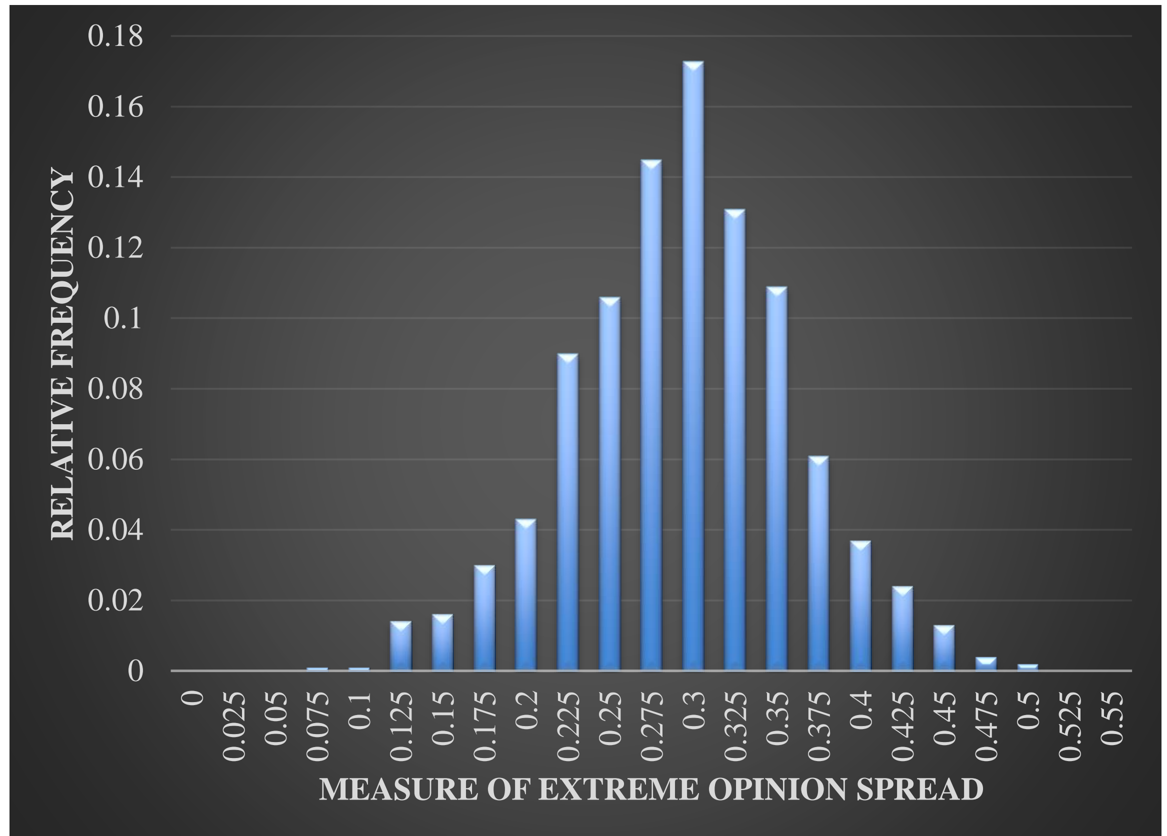

Figure 11: Output of Graph-based Relative Agreement Experiment with Gaussian(0.8, 0.1) Population Uncertainty.

Table 9: Input Specification for Graph-based Relative Agreement Experiment with Gaussian(0.4, 0.1) Population Uncertainty.

Graph Source	SNAP.
Agents	7,115
Initial Opinion Distribution	Uniform(-1, 1)
Population Uncertainty	Gaussian(0.4, 0.1), 1000 samples
Uncertainty of agents with extreme opinion	0.1
Control (Mu)	0.2

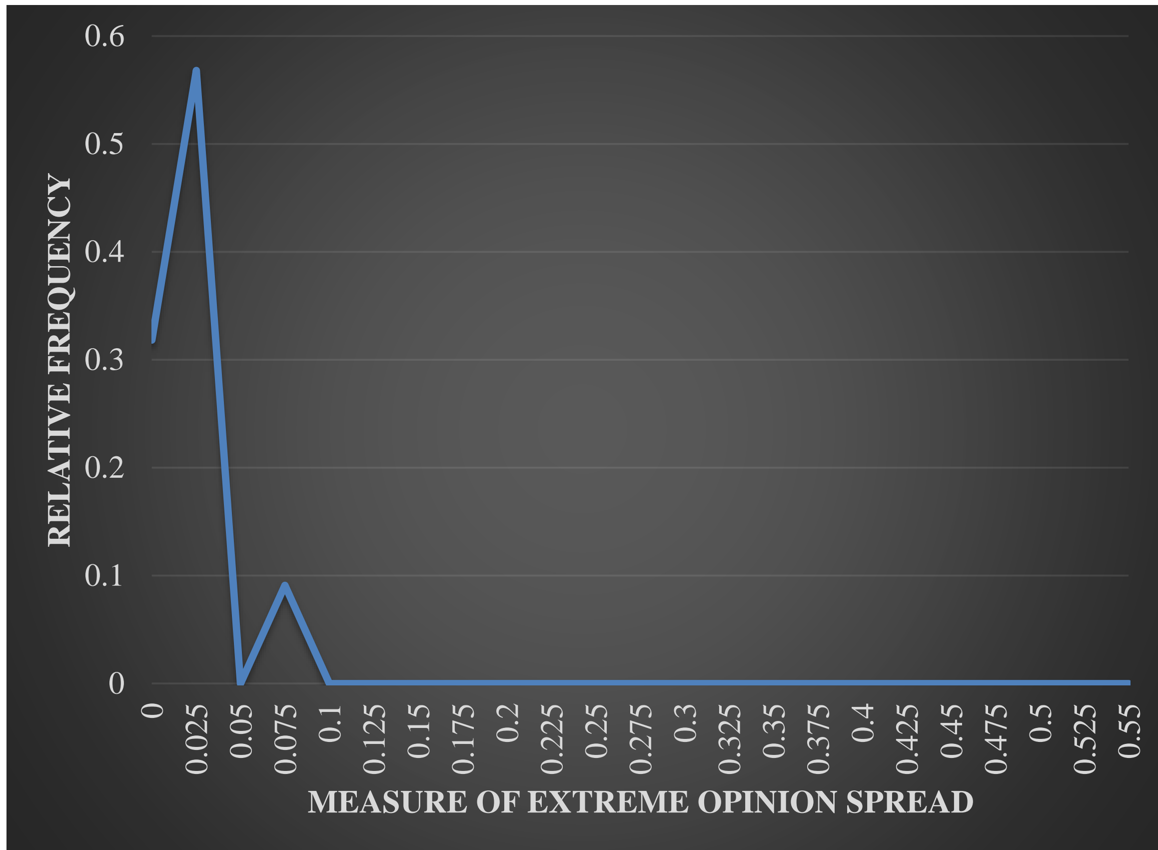

Figure 12: Output of Graph-based Relative Agreement Experiment with Gaussian(0.4, 0.1) Population Uncertainty.

5.2 Bounded Confidence Model

In the uniform case Bounded Confidence model is characterized by the confidence range parameter. We see that in this model opinion convergence always happens, and that the time required to reach consensus is controlled by the confidence range parameter. The larger the confidence range parameter is, the less time is required to reach consensus.

5.2.1 *Effect of confidence range (ε)*

Table 10: Input Specification for Bounded Confidence Experiment with confidence Range of 0.5.

Number of Agents	200
Initial Opinion Distribution	Uniform(-1,1)
Confidence Range(ε)	0.5
Iterations	10

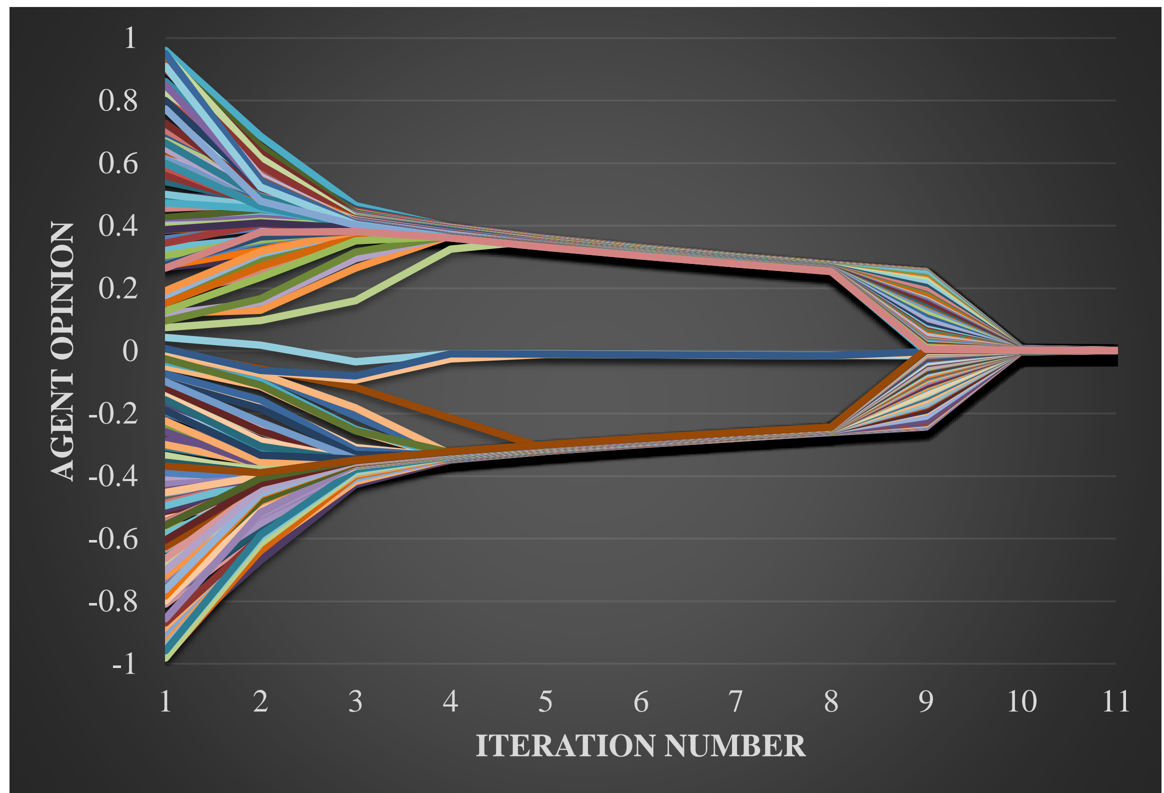

Figure 13: Output of Bounded Confidence Experiment with confidence Range of 0.5.

Table 11: Input Specification for Bounded Confidence Experiment with confidence Range of 0.01.

Number of Agents	200
Initial Opinion Distribution	Uniform(-1,1)
Confidence Range(ε)	0.01
Iterations	90

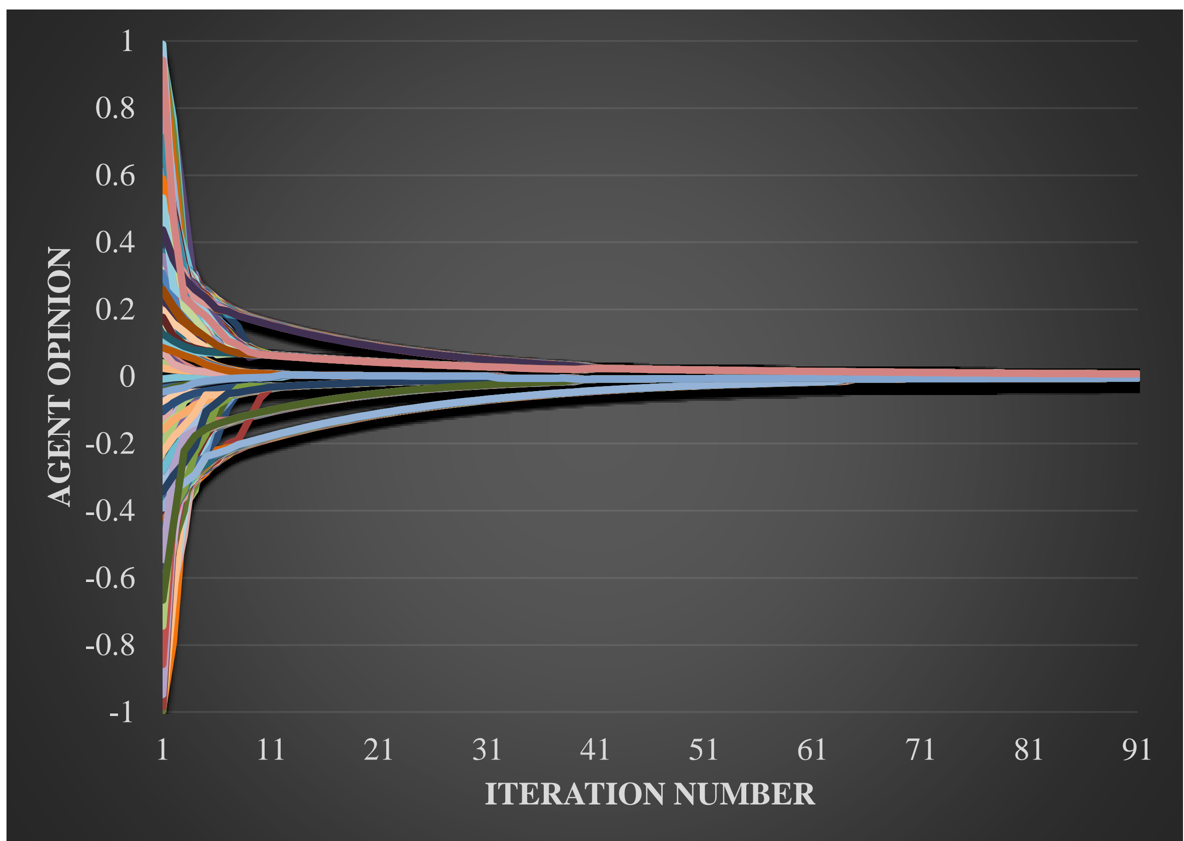

Figure 14: Output of Bounded Confidence Experiment with confidence Range of 0.01.

5.2.2 *Effect of initial opinion*

Similar to Relative Agreement model when all agents have the same uncertainty, we see that convergence is reached at a value corresponding to the mean of the input distribution.

Table 12: Input Specification for Bounded Confidence Experiment with Uniform(0,1) Agent Initial Opinion.

Number of Agents	200
Initial Opinion Distribution	Uniform(0,1)
Confidence Range(ε)	0.5
Iterations	90

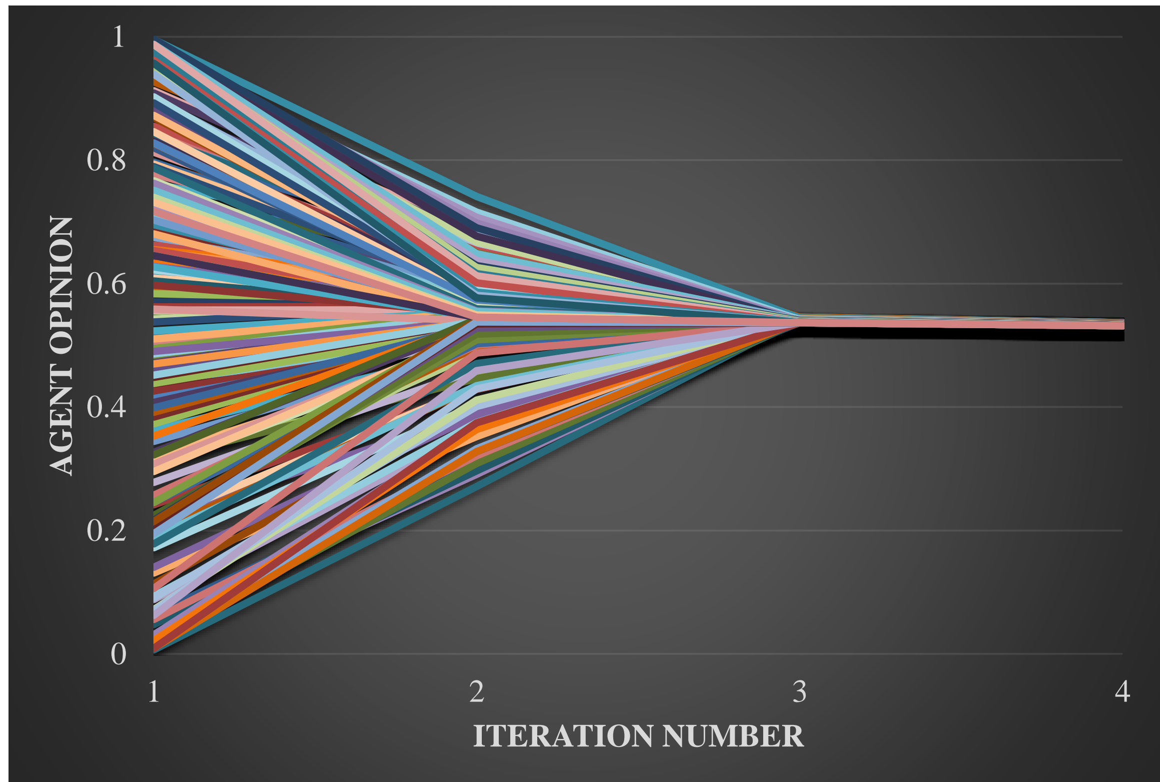

Figure 15: Output of Bounded Confidence Experiment with Uniform(0,1) Agent Initial Opinion.

5.3 FJ MODEL

FJ model has a difference definition of consensus. According to FJ model consensus is the steady-state of the system. There need not be any convergence in opinion. In Fact opinions typically don't converge, and multiple groups are formed. Also, FJ model requires a weighted graph as an input.

We find that the range of opinions at steady state is controlled by the agent susceptibility parameter. With high susceptibility final opinion range is more concise.

Table 13: Input Specification for FJ Experiment with Uniform(0.5, 0.7) Agent Susceptibility.

Graph source	Synthetic graph
Agents	30
Initial Opinion Distribution	Uniform(0,1)
Susceptibility	Uniform(0.5, 0.7)
Iterations	10

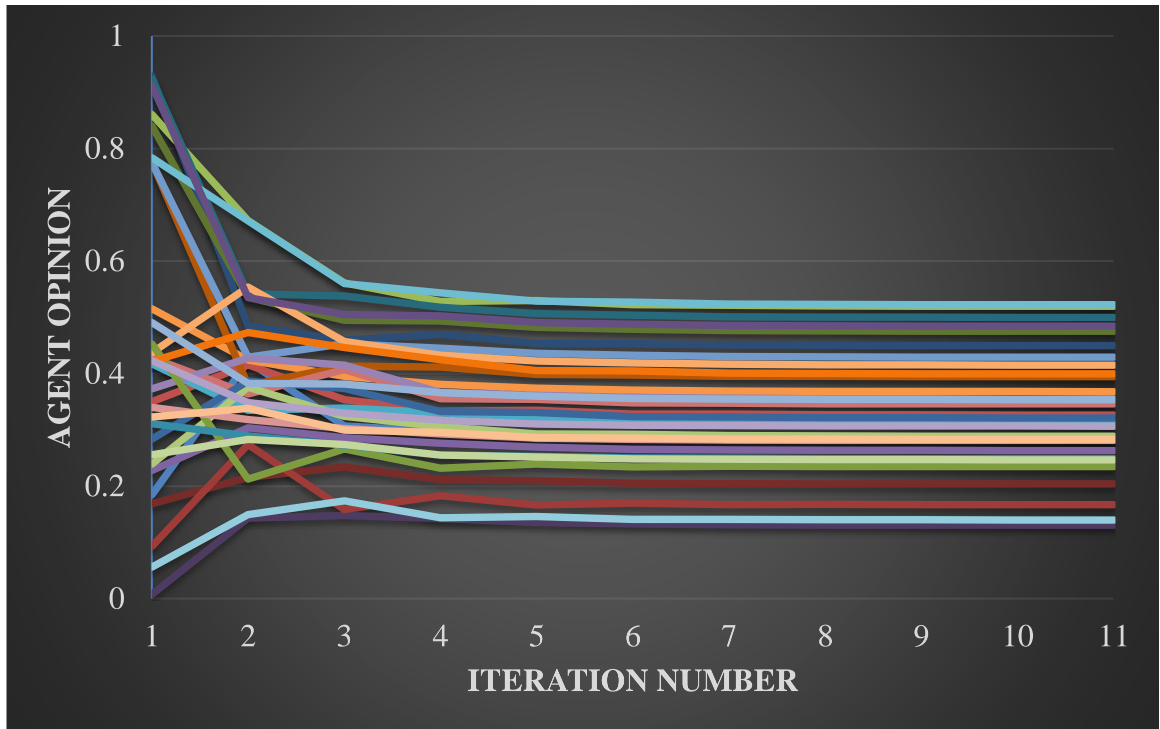

Figure 16: Output of FJ Experiment with Uniform(0.5, 0.7) Agent Susceptibility.

Table 14: Input Specification for FJ Experiment with Uniform(0.1, 0.3) Agent Susceptibility.

Graph source	Synthetic graph
Agents	30
Initial Opinion Distribution	Uniform(0, 1)
Susceptibility	Uniform(0.1, 0.3)
Iterations	10

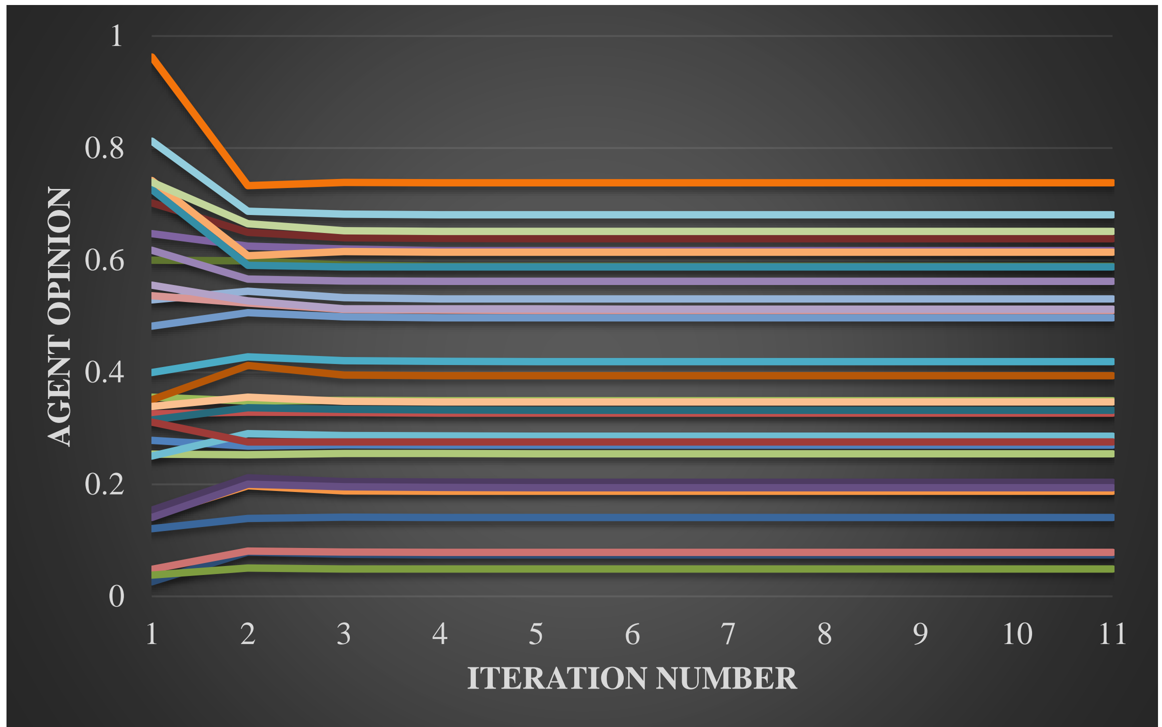

Figure 17: Output of FJ Experiment with Uniform(0.1, 0.3) Agent Susceptibility.

Chapter 6. CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

Using the developed simulation framework we were able to explore multiple social dynamics models. By studying model parameters we were able to understand each model as well as similarities and differences among them. We observe two kinds of similarities; relative weights, and susceptibility to change.

6.1.1 *Relative weights similarity:*

In FJ models we see explicit weights given to connections between any connected agents. These weights are generated by the experiment designer and allow exploring existing weighted graph networks. Also, weights can be calculated from interactions within the network [Sathanur2013]. In Bounded Confidence model weights exist in the form of averaging factor for each agent opinion. In Relative Agreement model weight are calculated as a function of agents' opinion and uncertainty.

6.1.2 *Susceptibility to change similarity:*

In FJ model susceptibility to change opinion is specified as an explicit agent parameter. On the other hand Bounded Confidence model has the notion of susceptibility in the form of increased confidence range. While in Relative Agreement model we see that susceptibility is exhibited by the agent's uncertainty. In all models we see that increased susceptibility means reaching consensus faster and close to average initial opinion.

6.2 FUTURE WORK

- Add support for cloud-based computations to run large-scale simulations (e.g. on twitter and Facebook social graphs).
- Use GPU processing; very suitable for parallelizable operations and offers good performance on a single machine.
- Add support for templates to investigate emergent phenomena.
- Introduce an optimization framework allowing phenomena engineering (find parameters that lead to a certain population level phenomena).

APPENDIX: CODE LISTING FOR CORE SIMULATION FRAMEWORK

ExperimentExecutor.java:

```
package edu.uw.acelab.social.simulation;

import java.io.IOException;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

import edu.uw.acelab.social.simulation.reporters.ExperimentReporter;

/**
 * Executes an experiment by setting up necessary parameters,
simulation
 * factory, and performing simulations in a multi-threaded fashion.
 *
 */
public class ExperimentExecutoer<T extends Simulation> {

	private final SimulationFactory<T> factory;
	private final ExecutorService pool;
	private final ExperimentReporter reporter;

	public ExperimentExecutoer(Class<T> clazz,
Set<SimulationParameter> params,
			ExperimentReporter reporter, int numberOfThreads) {
		factory = new SimulationFactory<T>(clazz, params,
reporter);
		pool = Executors.newFixedThreadPool(numberOfThreads);
		this.reporter = reporter;
	}

	public ExperimentExecutoer(Class<T> clazz,
Set<SimulationParameter> params,
			ExperimentReporter reporter, int numberOfThreads,
String graphFileName) throws IOException {
		factory = new SimulationFactory<T>(clazz, params, reporter,
graphFileName);
		pool = Executors.newFixedThreadPool(numberOfThreads);
		this.reporter = reporter;
	}
```

```
	public void execute() throws InterruptedException {
		Simulation sim;
		while ((sim = factory.create()) != null) {
			pool.execute(sim);
		}
		pool.shutdown();
		// Ridiculously long time to ensure simulation is done.
		pool.awaitTermination(Long.MAX_VALUE, TimeUnit.MINUTES);
		// report output
		reporter.createReport();
	}
}
```

Simulation.java:

```
package edu.uw.acelab.social.simulation;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Set;

import edu.uw.acelab.social.graph.Graph;
import edu.uw.acelab.social.random.RandomDistributionGenerator;
import edu.uw.acelab.social.random.RandomDistributionGeneratorFactory;
import edu.uw.acelab.social.simulation.reporters.ExperimentReporter;

/**
 * A simulation that uses set of parameters to perform a unit in a
social
 * experiment.
 *
 * @author abuomarm
 *
 */
public abstract class Simulation implements Runnable {

	private final Set<SimulationParameter> parameters;
	private final ExperimentReporter reporter;
	protected double output;

	// Number of time the simulation is performed. Agents are
reinitialized
	// prior to performing any given sample. Defined as double to
remove the
	// need for casting while using this in calculations.
	protected final double numberOfSamples;
```

```
	// Number of steps within a simulation sample. ith step uses
agent
	// properties from (i-1)th step.
	protected final double numberOfSteps;

	protected final int numberOfAgents;

	// Graph to be used for simulation in case of graph simulations.
	protected Graph<String> graph;

	// Generate opinions using a local opinion generator (using
factory) to be
	// independent of RandomGeneration library thread-safety support.
	protected RandomDistributionGeneratorFactory
opinionGeneratorFactory;
	protected RandomDistributionGenerator opinionGenerator;

	// Time series data; Stores initial agent properties and after
each
	// simulation step. The logical structure looks like:
	// ti => p11,...p1n, p21, ...p2n,...pm1,...pmn.
	// Where pij represents the value of property i for agent j, and
ti is the
	// step number.
	protected List<List<Double>> timeSeriesData;

	/**
	 * Instantiates Simulation with given parameters.
	 *
	 * @param parameters
	 *            {@link Set} of {@link SimulationParameter} to be
used for
	 *            performing the simulation.
	 */
	public Simulation(final Set<SimulationParameter> parameters,
			ExperimentReporter reporter, Graph<String> graph) {
		// Parameters set should never change.
		this.parameters = Collections.unmodifiableSet(parameters);
		this.reporter = reporter;
		SimulationParameter param =
getParameter(SimulationParameterName.NumberOfSamples);
		this.numberOfSamples = param == null ? 1 :
param.getValue();
		param =
getParameter(SimulationParameterName.NumberOfAgents);
		this.numberOfAgents = param == null ? 0 :
param.getValue().intValue();
		param =
getParameter(SimulationParameterName.NumberOfSteps);
		this.numberOfSteps = param == null ? 1 : param.getValue();
		SimulationParameter opinionGeneratorParam =
getParameter(SimulationParameterName.AgentOpinionGeneratorFactory);
```

```
		if (opinionGeneratorParam != null) {
			this.opinionGeneratorFactory = opinionGeneratorParam
					.getRandomGenerator();
			this.opinionGenerator =
this.opinionGeneratorFactory.getGenerator();
		}
		this.graph = graph;
	}

	/**
	 * @return Simulation parameters.
	 */
	public Set<SimulationParameter> getParameters() {
		return parameters;
	}

	/**
	 * @return Simulation output.
	 */
	public double getOutput() {
		return output;
	}

	protected abstract void initAgents();

	protected abstract double analyzeSample();

	private double performSample() {
		// Initialize agents (once per sample).
		initAgents();
		// interactions.
		performInteractionSteps();
		// analyze results and return
		return analyzeSample();
	}

	/**
	 * Performs the simulation using provided parameters.
	 */
	private void perform() {
		// Perform the simulation numberOfSteps times.
		for (int i = 0; i < numberOfSamples; i++) {
			output += performSample();
		}
		// average final output.
		output /= numberOfSamples;
	}

	public void run() {
		perform();
		reporter.reportOutput(this);
	}
```

```
	/**
	 * Get a parameter using its name.
	 * 
	 * @param name
	 *            Parameter name.
	 * @return {@link SimulationParameter} with provided name, or
null.
	 */
	public final SimulationParameter getParameter(
			final SimulationParameterName name) {
		for (SimulationParameter v : parameters) {
			if (v.getName().equals(name)) {
				return v;
			}
		}
		// If nothing found return null.
		return null;
	}

	/**
	 * Get agent properties for all steps performed.
	 * 
	 * @return <p>
	 *         List of all agent properties for each step. Each
element in the
	 *         result is a List<Double> containing properties of each
agent in
	 *         the simulation. Properties are reported sequential by
property
	 *         for all agents, then next property series for all
agents and so
	 *         on. As an example:
	 *         </p>
	 *         <p>
	 *         p11,...p1n, p21, ...p2n,...pm1,...pmn.
	 *         </p>
	 *         <p>
	 *         Where pij represents the value of property i for agent
j, and ti
	 *         is the step number.
	 *         </p>
	 */
	public List<List<Double>> getTimeSeriesData() {
		return timeSeriesData;
	}

	protected abstract List<Double> getAgentsOpinion();

	protected abstract void performCompleteGraphInteraction();

	protected abstract void performGraphInteraction();
```

```
	/**
	 * Performs steps of interaction between agents without re-
initialization.
	 */
	protected void performInteractionSteps() {
		// Recreate time series data for current sample.
		timeSeriesData = new ArrayList<List<Double>>();
		// Record initial condition
		List<Double> stepData = getAgentsOpinion();
		// Perform interaction steps.
		timeSeriesData.add(stepData);
		for (int i = 0; i < numberOfSteps; i++) {
			if (graph == null) {
				// Allow all agents to interact with one another
				performCompleteGraphInteraction();
			} else {
				performGraphInteraction();
			}
			// Record last step data
			stepData = getAgentsOpinion();
			timeSeriesData.add(stepData);
		}
	}

}
```

SimulationFactory.java:

```
package edu.uw.acelab.social.simulation;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Set;

import org.apache.commons.collections4.set.ListOrderedSet;

import edu.uw.acelab.social.graph.Graph;
import edu.uw.acelab.social.simulation.reporters.ExperimentReporter;
```

```
/**
 * Creates Simulations using a template {@link Set} of
 * {@link SimulationParameter}s to be run through an experiment.
 *
 * @author abuomarm
 *
 */
public class SimulationFactory<T extends Simulation> {

      /**
       * After all simulations are exhausted, this becomes true and
further calls
       * to create will return null.
       */
      private boolean done;

      /**
       * {@link Set} of {@link SimulationParameter}s which don't change
during an
       * experiment.
       */
      // Set is chosen since parameters are expected to be unique.
      private final Set<SimulationParameter> constantParameters;

      /**
       * A {@link Queue} with all possible permutations of
       * {@link SimulationParameter}s that change during an experiment.
The
       * content of generated by Cartesian product of all varying
parameters.
       */
      // Queue is used since we need a permutation exactly once. And it
also
      // removes the reference to permutation from Factory's memory as
soon as a
      // simulation is created.
      // Plus it's more compact that using a List/Array and keeping
track of the
      // current/next index permutation set.
      private final Queue<Set<SimulationParameter>> varyingParameters;

      private final Class<T> clazz;
      private final ExperimentReporter reporter;

      private Graph<String> graph;

      /**
       * Initializes the factory with template parameters.
       *
       * @param parameters
       *            A {@link Set} of {@link SimulationParameter}
representing a
```

```
     *             template for generating simulation instances.
     * @param reporter
     *             responsible for reporting simulation output.
     */
    public SimulationFactory(Class<T> clazz,
            Set<SimulationParameter> parameters,
ExperimentReporter reporter) {
        this.clazz = clazz;
        this.reporter = reporter;
        constantParameters = new HashSet<SimulationParameter>();
        ArrayList<Set<SimulationParameter>> baseVaryingParameters =
new ArrayList<Set<SimulationParameter>>();
        for (SimulationParameter p : parameters) {
            if (!p.getIsVarying()) {
                constantParameters.add(p);
            } else {
                // Generate parameter set (all possible values)
using input
                // range.
                Set<SimulationParameter> parameterSet =
p.expand();
                baseVaryingParameters.add(parameterSet);
            }
        }
        varyingParameters = new
LinkedList<Set<SimulationParameter>>();
        if (baseVaryingParameters.size() == 1) {
            // Expand the single Set to a set of sets each of 1
parameter.
            for (SimulationParameter p :
baseVaryingParameters.get(0)) {
                Set<SimulationParameter> set = new
HashSet<SimulationParameter>();
                set.add(p);
                varyingParameters.add(set);
            }
        } else if (baseVaryingParameters.size() > 1) {

    varyingParameters.addAll(cartesianProduct(baseVaryingParameters))
;
        }
    }

    /**
     * SimulationFactory with a graph.
     *
     * @param clazz
     *             Simulation class.
     * @param params
     *             parameters for the simulation.
     * @param reporter
```

```
	 *            experiment reporter (reports each simulation
output).
	 * @param graphFilename
	 *            File containing graph to be used by each simulation
instance.
	 * @throws IOException
	 */
	public SimulationFactory(Class<T> clazz, Set<SimulationParameter>
params,
			ExperimentReporter reporter, String graphFilename)
			throws IOException {
		// Initialize with regular params
		this(clazz, params, reporter);
		// Make sure graph file exists
		File file = new File(graphFilename);
		if (!file.exists() || file.isDirectory()) {
			// Invalid input
			throw new FileNotFoundException("Couldn't find input
file.");
		}
		// initialize graph. The graph consists of identifiers
linking nodes.
		// Each node is of type Long to hold an Id to an agent in
the graph.
		// When the simulation run, agents are created with the
same Id
		// corresponding to graph nodes. Then the graph is used to
get the
		// linked agents.
		graph = new Graph<String>();
		try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
			for (String line = reader.readLine(); (line =
reader.readLine()) != null;) {
				// File format is assumed to be
				// <SourceNodeId>\t<DestinationNodeId>
				// A line starting by # is considered a comment
line and is
				// ignored.
				if (line.startsWith("#")) {
					continue;
				}
				String[] nodeIds = line.split("[\t ]");
				if (nodeIds.length == 2) {
					// None weighted graph
					graph.addEdge(nodeIds[0], nodeIds[1]);
				} else if (nodeIds.length == 3) {
					// weighted graph
					graph.addEdge(nodeIds[0], nodeIds[1],
							Double.valueOf(nodeIds[2]));
				}
			}
```

```
		}

	}

	private Set<SimulationParameter> prepareParameters() {
		Set<SimulationParameter> instanceParameters = new
HashSet<SimulationParameter>();
		// copy over constant parameters
		instanceParameters.addAll(constantParameters);
		if (varyingParameters.size() < 2) {
			// less than 2; means either 1 => varyingParameters
were originally
			// supplied and will be exhausted in this call, or 0
=>
			// varyingParameters were never supplied and we'll
create just
			// this one parameter set.
			done = true;
		}
		if (varyingParameters.size() > 0) {
			// pick the next set of varying parameters
			for (SimulationParameter v : varyingParameters.poll())
{
				instanceParameters.add(v);
			}
		}
		return instanceParameters;
	}

	private static Set<Set<SimulationParameter>> cartesianProduct(
			ArrayList<Set<SimulationParameter>> sets) {
		if (sets.size() < 2)
			throw new IllegalArgumentException(
					"Can't have a product of fewer than two
sets (got "
							+ sets.size() + ")");

		return _cartesianProduct(0, sets);
	}

	private static Set<Set<SimulationParameter>> _cartesianProduct(
			final int index, final
ArrayList<Set<SimulationParameter>> sets) {
		Set<Set<SimulationParameter>> ret = new
ListOrderedSet<Set<SimulationParameter>>();
		if (index == sets.size()) {
			ret.add(new HashSet<SimulationParameter>());
		} else {
			for (SimulationParameter p : sets.get(index)) {
				for (Set<SimulationParameter> set :
_cartesianProduct(
						index + 1, sets)) {
```

```
                        set.add(p);
                        ret.add(set);
                    }
                }
            }
            return ret;
        }

        /**
         * Creates a simulation T
         *
         */
        public synchronized T create() {
            if (done) {
                return null;
            }
            T instance = null;
            Constructor<T> constructor = null;
            try {
                constructor = clazz.getDeclaredConstructor(Set.class,
                        ExperimentReporter.class, Graph.class);
                instance =
constructor.newInstance(prepareParameters(), reporter,
                        graph);
            } catch (NoSuchMethodException | InstantiationException
                    | IllegalAccessException |
InvocationTargetException e) {
                e.printStackTrace();
            }
            if (instance == null) {
                throw new UnsupportedOperationException(
                        "Something Terrible happened, can't create
Simulation!");
            }
            return instance;
        }
}
```

SimulationParameter.java:

```
package edu.uw.acelab.social.simulation;

import java.util.HashSet;
import java.util.Set;

import edu.uw.acelab.social.random.RandomDistributionGenerator;
import edu.uw.acelab.social.random.RandomDistributionGeneratorFactory;

/**
 * Parameter used for simulation as part of a social experiment.
```

```
 *
 * @author abuomarm
 *
 */
public class SimulationParameter {
     private final SimulationParameterName name;
     private final Boolean isVarying;
     private final Double value;
     private final Double min;
     private final Double max;
     private final Double steps;
     private Integer index;
     private final RandomDistributionGeneratorFactory
randomGeneratorFactory;
     private final Integer numberOfSamples;

     /**
      * @return the name.
      */
     public SimulationParameterName getName() {
          return name;
     }

     /**
      * @return the isVarying.
      */
     public Boolean getIsVarying() {
          return isVarying;
     }

     /**
      * @return the value.
      */
     public Double getValue() {
          return value;
     }

     /**
      * @return the min
      */
     public Double getMin() {
          return min;
     }

     /**
      * @return the max
      */
     public Double getMax() {
          return max;
     }

     /**
```

```
	 * @return the steps
	 */
	public Double getSteps() {
		return steps;
	}

	/**
	 * @return the index of this parameter within the parameter range
during the
	 *         experiment.
	 */
	public Integer getIndex() {
		return index;
	}

	/**
	 * Update varying parameter index. Should only used by the
	 * SimulationFactory.
	 *
	 * @param index
	 */
	public void setIndex(Integer index) {
		this.index = index;
	}

	/**
	 * Constructor - Initializes the parameter instance with supplied
values.
	 *
	 * @param name
	 *            Parameter name.
	 * @param isVarying
	 *            Specifies if the parameter is varying during the
experiment.
	 * @param value
	 *            Specifies parameter value.
	 * @param min
	 *            Min value for the parameter. only applicable if
isVarying is
	 *            true.
	 * @param max
	 *            Max value for the parameter. Only applicable if
isVarying is
	 *            true.
	 * @param steps
	 *            Number of steps to vary between min and max to
define the
	 *            range. Only applicable if isVarying is true.
	 * @param index
	 *            Current index within parameter range during the
experiment
	 */
```

```
	private SimulationParameter(final SimulationParameterName name,
			final Boolean isVarying, final Double value, final
Double min,
			final Double max, final Double steps, final Integer
index,
			final RandomDistributionGeneratorFactory
randomGenerator,
			final Integer numberOfSamples) {
		this.name = name;
		this.isVarying = isVarying;
		this.value = value;
		this.min = min;
		this.max = max;
		this.steps = steps;
		this.index = index;
		this.randomGeneratorFactory = randomGenerator;
		this.numberOfSamples = numberOfSamples;
	}

	/**
	 * @return the randomGenerator
	 */
	public RandomDistributionGeneratorFactory getRandomGenerator() {
		return randomGeneratorFactory;
	}

	/**
	 * Generates a set of all possible values a parameter can take
using
	 * specified Min, Max, and Steps.
	 *
	 * @return {
	 */
	public Set<SimulationParameter> expand() {
		if (!isVarying) {
			throw new UnsupportedOperationException(
					"Only varying parameters can be expanded");
		}
		Set<SimulationParameter> set = new
HashSet<SimulationParameter>();
		if (randomGeneratorFactory == null) {
			for (int ctr = 0; ctr <= steps; ctr++) {
				double value = min + ctr * (max - min) / steps;
				SimulationParameter v = new
SimulationParameter(name, true,
						value, min, max, steps, ctr, null,
null);
				set.add(v);
			}
		} else {
			RandomDistributionGenerator generator =
randomGeneratorFactory
```

```
                    .getGenerator();
            for (int ctr = 0; ctr < numberOfSamples; ctr++) {
                set.add(new SimulationParameter(name, true,
generator
                        .nextDouble(), null, null, null, null,
null, null));
            }
        }
        return set;
    }

    public static SimulationParameter newConstantParameter(
            SimulationParameterName name, Double value) {
        return new SimulationParameter(name, false, value, null,
null, null,
                null, null, null);
    }

    public static SimulationParameter newVaryingParemter(
            SimulationParameterName name, Double min, Double max,
Double steps) {
        return new SimulationParameter(name, true, null, min, max,
steps, null,
                null, null);
    }

    public static SimulationParameter newRandomDistributionParameter(
            SimulationParameterName name,
            RandomDistributionGeneratorFactory generator, Integer
numberOfSamples) {
        return new SimulationParameter(name, true, null, null,
null, null,
                null, generator, numberOfSamples);
    }

    public static SimulationParameter newAgentDistributionParameter(
            SimulationParameterName name,
RandomDistributionGeneratorFactory generator) {
        // Agent distribution is constant throughout the
experiment.
        return new SimulationParameter(name, false, null, null,
null, null,
                null, generator, null);
    }
}
```

REFERENCES

[Gilbert2005] "Simulation for the social Scientist", 2nd Edition, Nigel Gilbert, Klaus Troitzsch, Open University Press, 2005.

[Friedken1990] "Social Influence and Opinions", Noah E Friedkin, Eugene C. Johnsen, Journal of Mathematical Sociology, 1990, Vol. 15 (3-4), pp. 193-205.

[Sathanur2013] "PHYSENSE: Scalable Sociological Interaction Models for Influence Estimation in Online Social Networks", Arun Sathanur, Vikram Jandhyala, Chuangia Xing, IEEE International Conference on, Intelligence and Security Informatics (IS), 2013.

[Gabbay2012] "Majority Rule in Nonlinear Opinion Dynamics", Michael Gabbay, Arindam Das, International Conference on Theory and Applications of Nonlinear Dynamics (ICAND), 2012.

[Hegselmann2001] "Opinion Dynamics and Bounded Confidence", Rainer Hegselmann, Ulrich Krause, Journal of Artificial Societies and Social Simulation (JASSS), Vol 5, No. 3, 2002.

[Furtunato2004] "The Krause-Hegselmann Consensus Model with Discrete Opinions", Santo Furtunato, International Journal of Modern Physics C (Computational Physics and Physical Computation), 2004.

[Deffuant2000] "Mixing Beliefs among Interacting Models", Guillaume Deffuant, David Neau, Frederic Amblard, Gerard Weisbuch, Advances in Complex Systems, Vol. 3, 2000.

[Blondel2009] "On Krause's Multi-Agent Consensus Model with State-Dependent Connectivity", IEEE Transactions on Automatic Control, 2009.

[Dittmer2001] "Consensus Formation under Bounded Confidence", Jan Christian Ditmer, Nonlinear Analysis, Vol. 47, 2001.

[Deffuant2002] "How Can Extremism Prevail? A Study Based on The Relative Agreement Interaction Model", Guillaume Deffuant, Frederic Mablard, Gerard Weisbuch, Thierry Faure, Journal of Artificial Societies and Social Simulation, Vol. 5, No. 4, 2002.

[Meadows2012] "Reexamining the Relative Agreement Model of Opinion Dynamics", Michael Meadows, Dave Cliff, Journal of Artificial Societies and Social Simulation, 2012.

[Meadows2013] "The Relative Agreement Model of Opinion Dynamics in Populations with Complex Social Network Structure", Michael Meadows, Dave Cliff, Proceedings of the 4th Workshop on Complex Networks, CompleNet, 2013.

[Deffuant2005] "An individual-Based Model of Innovation Diffusion Mixing Social Value and Individual Benefit", Guillaume Deffuant, Sylvie Huet, Frederic Amblard, American Journal of Sociology (AJS), 2005.

[Pineda2009] "Noisy Continuous -Opinion Dynamics", Miguel Pineda, Raul Toral, Emilio Hernandez-Garcia, Journal of Statistical Mechanics: Theory and Experiment, 2009.

[Marsell2004] "PsychSim: Agent-Based Modeling of Social Interactions and Influence", Stacy Marsell, David Pynadath, Stephen Read, Proceedings of the International Conference on Cognitive Modeling, 2004.

[SNAP2008] http://snap.stanford.edu/data/wiki-Vote.html