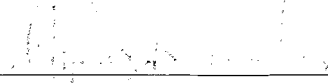


AGENT-BASED MODELS IN PUBLIC CHOICE

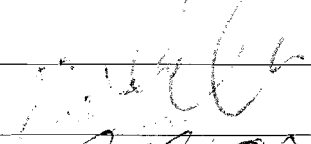
by

Richard R. Wallick
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Economics

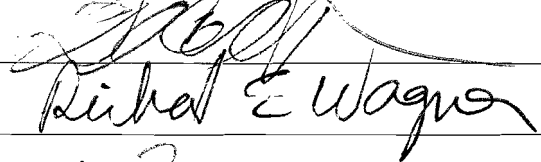
Committee:




Director



Department Chairperson


Richard E. Wagner

Program Director



Dean, College of Humanities
and Social Sciences

Date: June 25, 2012

Summer Semester 2012
George Mason University
Fairfax, VA

Agent-Based Models in Public Choice

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Richard R. Wallick
Master of Arts
George Mason University, 2007
Bachelor of Science
University of Calgary, 1992

Chairman: Charles K. Rowley, Duncan Black Professor Emeritus of Economics
College of Humanities and Social Sciences

Summer Semester 2012
George Mason University
Fairfax, VA

Copyright 2012 Richard R. Wallick
All Rights Reserved

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my advisor, Dr. Charles K. Rowley, without whom this dissertation would not have been possible. The phrase “would not have been possible” is an oft-invoked cliché, but in this case it is undeniably true. Dr. Rowley’s encouragement and guidance have been invaluable at every step in the process. His dedication to his calling should be an example to everyone who aspires to a life as a scholar and educator. There is a certain paradox in the fact that Dr. Rowley has spent his career proclaiming the dominance of self-interest in human affairs, and yet has consistently put the interests of his students ahead of his own. I cannot explain this, but I am grateful for it.

I also wish to thank Dr. Robert Axtell, who inspired me with visions of agent-based modeling in just my third week of graduate school, at Pete Boettke’s weekly Politics, Philosophy, and Economics workshop. I asked for his advice immediately after the workshop and he kindly replied with guidance that has served me well in the intervening years. I also owe a debt of gratitude to Dr. Omar Al-Ubaydli, who has helped me understand how agent-based modeling can best serve the discipline of economics, and who has kept me from tilting at windmills. And no George Mason University economics dissertation is complete without an expression of gratitude to Mary Jackson, without whom the entire edifice would crumble.

No one can complete an undertaking of this magnitude without a support network, and in this respect I have been especially blessed. I want to thank Dianne for tolerating my prolonged absences as I labored to complete this dissertation; if she has ever resented my dedication to her rival, she never let it be known. My grandmother has now suffered through two dissertations: that of my grandfather, and that of her grandson. I have promised her that I would remain humble, and this is an easy promise to keep: the more I learn, the more I understand how very much more there is yet to learn. Finally, I hope mom and pop understand that while I may thank them for the thousands—literally thousands!—of home-cooked meals and underlined newspapers, what I have *really* appreciated, and what I could not have done without, is the steadfast support, unwavering confidence, and unconditional love. Thank you from the bottom of my heart.

TABLE OF CONTENTS

	Page
List of Tables	v
List of Figures	vi
Abstract	viii
Agent-Based Modeling, Public Choice, and Gordon Tullock	1
An Agent-Based Model of Regulatory Capture	26
An Agent-Based Model of the Condorcet Jury Theorem	68
Appendices	145
References.....	188

LIST OF TABLES

Table	Page
3.1 Expected joint distribution of C1 and C2, no correlation	94
3.2 Expected joint distribution of C1 and C2, correlations -1 and 1	94
3.3 Observed joint distribution of C1 and C2, various correlations	95
3.4 GSS background and demographic questions.....	98
3.5 EARTHSUN as a function of GSS agent characteristics.....	103
3.6 GSS scientific questions	107
3.7 Regression results from the GSS	110
3.8 Composite regression coefficients	114
3.9 Behavior of wise agents	120

LIST OF FIGURES

Figure	Page
1.1 Identical buyers, Walrasian auctioneer	9
1.2 Identical buyers, price-setting sellers.....	11
2.1 Tax rate by percent of citizens holding licenses	44
2.2 Tax rate by percent of citizens holding licenses and deadweight costs	45
2.3 Approval rating by transfer per licensee and deadweight cost	46
2.4 Approval rating by transfer per licensee and dispersion of deadweight costs	48
2.5 Approval rating by transfer per licensee and types of heterogeneity.....	50
2.6 Approval rating by transfer per licensee and heterogeneity status	51
2.7 Approval rating by transfer per licensee: detail.....	53
2.8 Approval rating by persuasion per licensee	54
2.9 Approval rating by percent of citizens holding licenses	56
2.10 Approval rating by amount of cross-subsidy	58
2.11 Approval rating by amount of cross-subsidy and degree of heterogeneity	59
2.12 Approval rating by potential income of cross-subsidized agents	60
2.13 Approval rating by amount of transfer and direction: homogeneous groups	62
2.14 Approval rating by amount of transfer and direction: heterogeneous groups	63
2.15 Approval rating by income heterogeneity.....	65
3.1 Likelihood of a correct vote by number of voters.....	80
3.2 Likelihood of a correct vote by number of voters: detail.....	82
3.3 Values returned by the model by number of runs.....	84
3.4 Convergence by number of runs	86
3.5 Convergence and run time by number of runs.....	87
3.6 Convergence by voters and number of runs	89
3.7 Likelihood of a correct vote by variance of voter distribution	91
3.8 Likelihood of a correct vote by number of voters and distribution	92
3.9 Percent of agents with $C1=C2=0$ by $\text{corr}(C1,C2)$	97
3.10 Distribution of WORDSUM values from the GSS.....	99
3.11 Likelihood of a correct vote by complexity: EARTHSUN only	106
3.12 Likelihood of a correct vote by complexity: all questions.....	116
3.13 Likelihood of a correct vote by agent confidence.....	119
3.14 Likelihood of a correct vote by agent wisdom.....	122
3.15 Likelihood of a correct vote by agent wisdom: detail.....	123
3.16 Likelihood of a correct vote by agent wisdom: herding	124
3.17 Likelihood of a correct vote by agent wisdom: wise vs. unwise	126

3.18	Likelihood of a correct vote by social network fragmentation	127
3.19	Likelihood of a correct vote by social network fragmentation: detail	129
3.20	Likelihood of a correct vote by complexity: wise vs. unwise	130
3.21	Likelihood of a correct vote by confidence: sincere vs. pivotal	133
3.22	Likelihood of a correct vote by confidence: three judges	134
3.23	Likelihood of a correct vote by confidence: unanimity	135
3.24	Likelihood of a correct vote by confidence: pivotal vs. predominant	137
3.25	Likelihood of a correct vote by complexity: GSS agents	138

ABSTRACT

AGENT-BASED MODELS IN PUBLIC CHOICE

Richard R. Wallick, Ph.D.

George Mason University, 2012

Dissertation Director: Dr. Charles K. Rowley

Public choice is the study of political science using the tools of economics. The future of modeling in public choice may be glimpsed by examining its evolution in economics. For problems that are influenced by heterogeneity of actors, social networks, or *emergence*—the arising of a complex system from simple phenomena, such as Adam Smith’s “invisible hand”—economists increasingly are turning to agent-based modeling. This dissertation argues that public choice scholars can benefit from following the same path.

Chapter 1 introduces agent-based modeling, describing its advantages and its disadvantages. By definition, the agent-based approach emphasizes methodological individualism. Since methodological individualism is a cornerstone of public choice, many public choice models are amenable to agent-based implementations. This is best exemplified by a review of Gordon Tullock’s approach to modeling. The agent-based approach features prominently in Tullock’s writings. Had he been armed with the

technology of the 21st century instead of the 20th, Tullock would have been an agent-based modeler. Agent-based modeling offers a new vehicle for exploring Tullock's classic works.

Chapter 2 demonstrates the agent-based approach by transforming a familiar public choice model, Samuel Peltzman's 1976 model of regulatory capture, into an agent-based model. The agent-based version can explore situations that Peltzman's formulation cannot, such as the presence of bilateral transfers and the impact of heterogeneity among citizens. Different types of heterogeneity affect the model's predictions in different ways. Taken as a supplement to Peltzman's model, the agent-based version sheds new light on an old problem.

Chapter 3 then develops an extended agent-based model from scratch: a model of Condorcet's famous jury theorem. Condorcet's theorem rests on three assumptions, each of which may be questioned: voter homogeneity, voter independence, and voter sincerity. The agent-based version explores the impact of relaxing each of these assumptions. There is no shortage of rational choice models that relax these assumptions independently, but only the agent-based version relaxes all three simultaneously. The model is calibrated to represent a population like that of the United States, and used to explore policy proposals.

1. AGENT-BASED MODELING, PUBLIC CHOICE, AND GORDON TULLOCK

1.1 Introduction

The field of public choice has been described as “the application of economics to political science” (Mueller 2003, 1), so it is not surprising that public choice theorists have adopted the economist’s reliance on *models*. A model is an abstract, small-scale representation of some real-world phenomenon. Suppose, for example, that one asks an economist what will happen to the economy if the Federal Reserve raises banks’ reserve requirements. The economist will respond by selecting a model, manipulating the reserve requirements variable in that model, and reporting what the model predicts. Like economists, public choice scholars create and manipulate models. The difference between an economist and a public choice scholar is a matter of domain: economists model economic phenomena, while public choice scholars model political phenomena. Given their common heritage and shared approach, it seems likely that public choice modeling will follow the same evolutionary path as economic modeling. This chapter argues that this evolutionary path leads to agent-based modeling. It also argues that Gordon Tullock started down this path long ago, and thus that Tullock, 50 years later, remains at the forefront of the field he helped to found.

Agent-based modeling is a computational technique. Contrasted with other forms of modeling, agent-based modeling has two distinguishing features: support for arbitrarily heterogeneous actors (or “agents”), and support for adaptive behavior. These features may not seem impressive on their own, but when combined with a framework of rules guiding agents’ interactions, they can produce almost startlingly complex results. The ability of a self-interested agent “to promote an end which was no part of his intention” was recognized by Adam Smith (1994 [1776], 485) over two centuries ago. Hayek (1976, 2:33) calls it “spontaneous order”: “the position of each individual is the resultant action of many other individuals, and nobody has the responsibility or the power to ensure that these separate actions of many will produce a particular result.” It is for this reason that Vriend (2002, 811) describes Hayek as “an agent-based computational economist (ACE) *avant-la-lettre*.” By describing economic systems in terms of agents and their interactions, Hayek followed the agent-based modeling approach implicitly. Wagner (2008) observes that the same can be said of Gordon Tullock. If Hayek can be called an ACE, so too can Tullock, whose “scholarly oeuvre ... reflects a deep understanding and appreciation of spontaneous order theorizing” (ibid., 56). Tullock anticipated the agent-based modeling approach. By continuing down that path, modern scholars can explore Tullock’s work in new and deeper ways.

1.2 Economic modeling

Economic models have taken different forms at different times in history. An early form of economic modeling was mythology: in Roman times, for example, the

goddess Ceres protected crops. The Roman model of grain production was essentially this: to obtain more grain, make Ceres happy (Spaeth 1996). This model may seem simplistic to the modern reader, yet it was compelling enough to the Romans to motivate the erection of temples dedicated to Ceres. The model was compelling because it provided both explanatory and predictive power: it explained the dynamics of grain production, and it offered a way of understanding seasonal output variations. But what this model lacked—and what makes the model unacceptable to the modern reader—was *falsifiability*. The absence of an *ex ante* definition of what precisely made Ceres happy meant that no test could be devised that would disprove the model.

A model's lack of falsifiability does not mean that the model is *wrong*; a model may be accurate, yet unprovable in any formal sense. But nonfalsifiable models are no longer regarded as useful in the scientific community (Popper 1959). By contrast, falsifiable models *are* useful. A falsifiable model is one that begins with assumptions, applies logical deductions to those assumptions, and concludes by stating empirically testable predictions. Such a model is disproved when the assumptions are granted but the predictions are not fulfilled. Falsifiable models are powerful because they split the problem being modeled into two distinct parts—the premises and the deductions—which can be analyzed separately. For this reason, among others, mathematical models have come to predominate the economics profession (Stigler, Stigler and Friedland 1995). A mathematical model is the cleanest form of a falsifiable scientific model. Assumptions are expressed as mathematical identities; predictions are the algebraic consequences of those identities. Sociologist Vilfredo Pareto, writing in 1911, summarized the approach

now followed by all orthodox economists: “given the mathematical laws according to which certain individuals usually behave, determine the consequences of these laws” (Pareto 1955 [1911], 58).

Mathematical models in economics have evolved from simple models, such as IS-LM, to considerably more complex models. Many modern mathematical models of the economy are *rational choice* models. Rational choice models embody two key ideas. First, economic activity is generated by individual actors; macro-level activity is produced by aggregating micro-level activity. Second, those individual actors have “model-consistent expectations”: the actors understand the model and its consequences, and use those consequences to guide their own behaviors. The latter characteristic is intended to address the Lucas (1976) critique—the observation that real-life economic actors react to policy changes. A model that does not address the Lucas critique may make false predictions.

An example of a rational choice model is the well-known model of monetary policy of Barro and Gordon (1983). In the Barro-Gordon model, the economy manifests an optimal rate of price inflation, p^* . An inflation rate above p^* is suboptimal because “inflation is assumed to generate increasing marginal disutility” (Walsh 1998, 324), while an inflation rate below p^* is suboptimal because seigniorage opportunities are foregone, and so “distortions due to taxes, monopoly unions, or monopolistic competition may lead [output] to be inefficiently low” (ibid., 324). Bond-buyers in the Barro-Gordon model form expectations about the inflation rate, p^e . These expectations are important, because the Barro-Gordon model includes a short-run Phillips curve relationship between output

and unexpected inflation: output increases when the actual inflation rate, p , exceeds p^e . If the monetary authority optimizes for the long run, it sets $p = p^*$. But if the monetary authority successfully convinces all bond-buyers that p will equal p^* , and thus establishes $p^e = p^*$, it then faces a short-run incentive to renege, and set $p > p^*$ instead.

In the Barro-Gordon model, bond-buyers have rational expectations, so they understand the incentives of the monetary authority. If the monetary authority can renege on its promise of $p = p^*$, bond-buyers will account for this possibility by setting $p^e > p^*$. But they also know that there is an upper bound for p^e : for sufficiently large p , the disutility of p overwhelms the utility of $p - p^e$. (In the treatment in Walsh (1998), for example, p enters the objective function quadratically and negative, while $p - p^e$ enters the function linearly and positive.) Because the bond-buyers understand the model, they set p^e such that the monetary authority's optimal behavior is to produce $p = p^e$ (thus removing the incentive to generate a suboptimal $p > p^e$). One result of the Barro-Gordon model is the demonstration that a monetary authority with the discretion to renege on its commitments will always produce $p > p^*$. This is known as "inflation bias" (Walsh 1998, 336). Barro and Gordon argue that in the presence of rational actors, a rule which prevented the monetary authority from reneging on its commitments, and enforcing $p = p^*$, would be utility-enhancing.

The Barro-Gordon model hinges on rational expectations. Without rational expectations, the monetary authority could fool the bond-buyers repeatedly, and permanently increase output. In order to employ rational expectations, Barro and Gordon assume the following. First, the bond-buyers know the objective function that the

monetary authority will optimize. Second, the bond-buyers and the monetary authority share the same information. Both of these assumptions have been challenged in the subsequent literature. Blinder (1997) attacks the first assumption, suggesting that it is not just the bond-buyers who form expectations about the monetary authority; the monetary authority forms expectations about the bond-buyers, too, and factors those expectations into its own behavior. Cukierman and Meltzer (1986, 1102) attack the second assumption; they build a model in which “the policymaker uses information that the public does not have.” Ireland (1999) offers a model which allows credibility to supercede rational expectations. Gomes (2006) replaces rational expectations with bounded rationality.

In each of these cases, the modeler is augmenting the Barro-Gordon model to support greater agent heterogeneity. The importance of models with heterogeneous agents was identified by Haltiwanger and Waldman (1985, 326), who noted that heterogeneity is “an aspect of the real world,” yet “most models that contain the rationality assumption ignore the idea that agents tend to be heterogeneous in terms of information-processing abilities.” Branch (2004, 592) echoes a similar sentiment, noting that in the context of inflation expectations, “agents lack the requisite sophistication to form expectations rationally.”

The Cukierman and Meltzer (1986), Ireland (1999), and Gomes (2006) models address different areas of heterogeneity (limited information, credibility, and bounded rationality, respectively). In that sense, they share a common goal. Yet each of these variations on Barro-Gordon is independent. None of these models builds upon its

counterparts. There is no philosophical reason why this should be so; a single agent might exhibit heterogeneity in all three areas. The most likely reason for the lack of a unified, “compound” model is tractability. In rational choice models, supporting heterogeneity along a single dimension is not easy; supporting heterogeneity along three dimensions may be simply too difficult. One might say that while it is possible to represent heterogeneity in rational choice models, it does not “scale” well. Adding two dimensions of heterogeneity is considerably more than twice as complicated as adding one.

This is an unfortunate circumstance. In an ideal world, these three models would be combined into a single model, each building on the others. But the rational choice formulations of these models preclude any such collaboration. Nor are these isolated examples. Rubinstein and Wolinsky (1985, 1133), for example, model “the micro-mechanisms of price formation and their role in shaping market outcomes.” Heterogeneity in their model takes the form of employing exactly two types of actors: buyers and sellers. All buyers are identical to all other buyers, and all sellers identical to all other sellers. All actors “employ the same ... strategy ... against any potential opponent” (ibid., 1134), and all actors share a common degree of urgency to trade. The degree of heterogeneity is very low—but the rational choice framework allows no more. To expand the range of heterogeneity, a different technique must be used. Gjerstad and Dickhaut (1998) explore similar territory, but avoid these limitations by using simulations. They do not describe their simulation framework, nor do they explain

exactly how they simulate their model, but one can infer that their approach is similar to what is now known as *agent-based modeling*.

1.3 Agent-based modeling

An agent-based model is a computational problem in which the fundamental unit of analysis is the actual economic agent, rather than some aggregate or average of agents (Epstein and Axtell 1996). The designer of an agent-based model imbues each agent with preferences and behaviors, then allows the agents to interact. Those interactions result in changes in agents' endowments, and possibly in their future preferences and behaviors. Over repeated iterations, equilibria may or may not emerge (Tesfatsion 2006). The key benefit of the agent-based approach is that it imposes no limits on heterogeneity. Additional degrees of heterogeneity may be added to an agent-based model at very low cost. Support for arbitrary degrees of heterogeneity makes agent-based modeling the ideal vehicle for incorporating behavioral and experimental economics into broader economic models (Macal and North 2010).

To get the flavor of agent-based modeling, consider a simple model of supply and demand. Suppose there are S identical sellers with supply schedule $Q_s = a + bP$, and B identical buyers with demand schedule $Q_d = c - dP$. In equilibrium, the quantity supplied equals the quantity demanded, so the equilibrium price is found by setting $SQ_s = BQ_d$ and solving for P . The equilibrium price is $(Bc - Sa)/(Sb + Bd)$, and the equilibrium quantity is $S(a + b(Bc - Sa)/(Sb + Bd))$. When $a=100$, $b=10$, $c=20$, and $d=7$, a market with 1000 buyers and 50 sellers would produce sales of 6000 units at a price of 2 per unit. If the market

then experienced a negative demand shock, and c dropped from 20 to 17, the new equilibrium would be 5800 units at a price of 1.6 per unit.

A simple agent-based model of supply and demand produces the same results. Consider the simplest version of such a model: identical buyers, identical sellers, and a Walrasian auctioneer. Each period the auctioneer determines the total-surplus-maximizing price, and buyers and sellers are allowed to trade at that price (and only at that price). The total-surplus-maximizing price is, of course, the same as the equilibrium price in the Econ 101 model. Appendix 1 presents such a model, implemented in the Python programming language. Figure 1.1 shows that the agent-based model produces the same results as its Econ 101 counterpart. Before the demand shock, each period sees 6000 units sold at a price of 2 per unit. After the demand shock (in period 50), each period sees 5800 units sold at a price of 1.6 per unit.

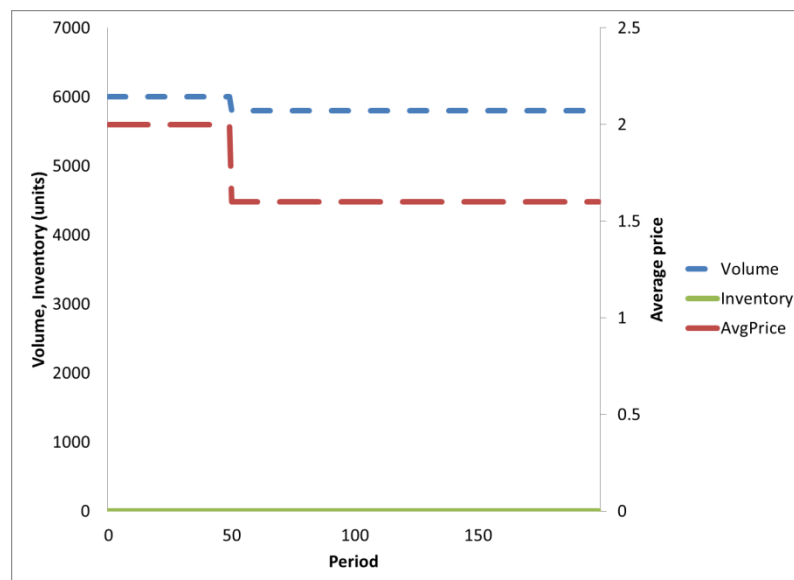


Figure 1.1: Identical buyers, Walrasian auctioneer

Notice in Figure 1.1 that inventories are always zero. The Walrasian auctioneer is able to foresee the demand shock and adjust production accordingly; therefore suppliers are never left with unsold goods. In real life, this would be unlikely. The reason the model produces an unrealistic prediction is that it assumes a surplus-maximizing auctioneer. But this auctioneer does not exist in real life. What happens if we remove the auctioneer from the model, and allow agents to set prices directly?

Consider a simple algorithm for price-setting: at the end of each period, each seller compares its sales with those of the previous period. If sales rose (fell) by $X\%$, the seller raises (lowers) prices by $X\%$ in the next period. In setting prices, the seller uses only local knowledge (its own supply schedule and inventory). As Figure 1.2 shows, the surplus-maximizing equilibrium price still emerges after a shock to demand, even without a Walrasian auctioneer. The difference is that the adjustment is not instantaneous. Sellers are unprepared for the demand shock, and inventories spike in the period of the shock. Sellers respond by slashing prices. Prices fall too far, partially offsetting the demand shock. Sellers then adjust in the other direction. Over time, average price converges to the expected equilibrium. Without the auctioneer, the equilibrium must be discovered; it must *emerge*.

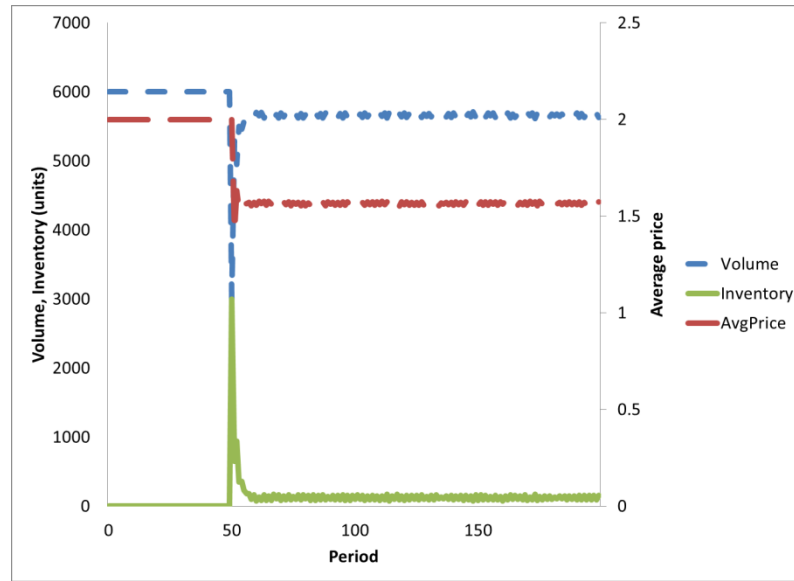


Figure 1.2: Identical buyers, price-setting sellers

This simple example demonstrates the overlap between agent-based models and rational choice models: in both modeling styles, individuals drive the model. The styles differ in the way that they express these individuals. Rational choice models employ rational agents, and produce closed-form results. Agent-based models employ arbitrary agents, and produce emergent results. Each style has its advantages. Rational choice models use rational expectations, so they are immune from the Lucas critique. Agent-based models support any type of expectations, so they allow higher degrees of heterogeneity. Rational choice models produce closed-form results, so their predictions can be tested econometrically. Agent-based models produce emergent results, so they do not require mathematical tractability.

The agent-based modeler must address the susceptibility of agent-based models to the Lucas critique. At first glance, it may seem that abandoning rational expectations is

an advantage, not a disadvantage: after all, real-world agents are not always rational. But the assumption of rationality is safer than the assumption of irrationality; rationality should be the point of departure. From an agent-based perspective, it is all too easy to implement an agent in terms of *rules* rather than *preferences*. Rule-driven agents will obey those rules even when it is not in the agent's interests to do so; preference-driven agents attempt to maximize their own utility, and will alter their behaviors in response to policy changes. To address the Lucas critique, the modeler must model preferences.

Modeling preferences not only addresses the Lucas critique; it “can throw light on ... endogenously arising innovative behaviour of agents as best response strategies” (Markose 2005, F186). But Fagiolo, Moneta, and Windrum (2007, 208) note that the Lucas critique may still be relevant, even when modeling preferences: agent-based models are frequently calibrated according to real-world conditions, and “calibration has a strongly conservative tendency.” If preferences are influenced by calibration, agents are actually behaving in policy-invariant ways—adhering to behaviors that were more appropriate under a different policy regime—even though it *seems* they are reacting to policy changes. The subtle nature of this problem makes agent-based models even more susceptible to the Lucas critique than the econometric models against which the critique was first leveled. So a disadvantage of agent-based modeling, relative to rational choice modeling, is that the former must work to avoid the Lucas critique, while the latter is immune by construction.

Another disadvantage of agent-based models, relative to rational choice models, is that they are more susceptible to outright errors. A rational choice model produces a

closed-form solution, each step of which may be checked by skeptics. An agent-based model produces an emergent solution by employing a computer program which is usually difficult to read and even more difficult to verify. (The simple agent-based model of supply and demand described earlier, for example, runs to four pages.) A reader who disbelieves the predictions of an agent-based model is forced to wade through pages of computer code, looking for something to criticize. Such a model is more likely to be ignored than rejected. It is true that a similar charge could be made against some rational choice models, with appendices containing proofs of theorems and lemmas that span many pages. But people learn mathematics from their earliest days in school; demanding that the reader work through a few pages of proofs may be acceptable. Demanding that they learn computer programming is not.

Agent-based modelers have made many attempts to address this criticism. For example, as Tobias and Hofmann (2004, 1.3) note, “the use of already developed simulation frameworks ... increases the reliability and efficiency” of agent-based models. Such frameworks include AgentSheets, Ascape, Breve, Cormas, ECHO, JADE, Madkit, MAGSY, MASON, MIMOSE, NetLogo, Ps-i, Quicksilver, RePast, SimAgent, SimPack, StarLogo, Sugarscape, Swarm, TeamBots, and VSEit. The sheer number of frameworks, by itself, indicates a lack of consensus on the best approach to agent-based modeling. A key challenge for agent-based modelers is to converge upon a consensus approach that combines brevity with reliability. The emergence of such a consensus will constitute a major breakthrough in the field. Until that breakthrough arrives, modelers are left to their own devices.

1.4 Agent-based modeling and public choice

The motivation for such a breakthrough will continue to mount, however, as agent-based models show greater degrees of descriptive and predictive power. This breakthrough is likely to come in fields in which real-world complexity exceeds the capacity of mathematical modeling. Public choice is one such field: for as Buchanan and Tollison (1984, 14) put it, “[by] any comparison with politics, economic theory is *simple*”. Economic interaction is impossible until “political exchange” has first occurred, and political exchange “necessarily involves *all* members of the relevant community rather than the two trading partners that characterize economic exchange” (ibid., 14). This combination of economic and political exchange occurs, for example, in the development of regulatory regimes: agents’ trading behaviors are shaped by regulations over which they themselves, as voters, may have some degree of influence. The decision to trade is a private choice; the decision to regulate is a public choice. Yet the same agent is involved in both, and each impacts the other. As Tullock (2000, 5) observes, “the *same* people engage in market activities and in politics.” Models of political behavior should recognize this.

That agent-based modeling is an ideal vehicle for exploring public choice is made clear by a closer examination of the entities that public choice scholars seek to understand. At the most elemental level, Buchanan and Tollison (1984, 13) criticize analysis of the political sector which “models the government as some sort of monolith, with a being of its own, somehow separate and apart from the individuals who actually participate in the process.” The “government” is comprised of many different types of

individuals—all of them self-interested, in the sense of *homo economicus*, but potentially with very different utility functions. In the legislative and executive realms, some pursue power for its own sake; some pursue power for monetary gain; some pursue power because they sincerely want to help others. Inevitably, too, some discover that doing good can also mean doing well, and that power carries unexpected charms. Utility functions change over time and with experience. Shifting utility functions are difficult to model mathematically; they are highly suited to an agent-based approach.

Modeling “government,” however, is about more than modeling group formation; it is also about modeling group decision-making and group implementation. In each of these areas, agent-based modeling offers advantages. Wagner (2007, 71) notes, for example, that crafting a government budget is a process “of bottom-up emergence, in which the aggregate entity called a budget is generated” via competition among political agents. Modeling this process of political exchange is like modeling the process of economic exchange: a model that *assumes* equilibrium, rather than allowing equilibrium to emerge, is missing something important. (In real life there is no Walrasian auctioneer.) Even when a budget does emerge, its implementation is left to bureaucrats, whose interests may not coincide with those of the budget’s writers. As Niskanen (1968, 293) first observed, bureaucrats are not always best described as actors “who, for whatever reason, want to be efficient.” Bureaucrats have their own utility functions, and bureaucrats are affected by the institutions within which they operate. “The theory of bureaucracy,” wrote Tullock (1976, 27), “should be based upon the assumption that bureaucrats are as self-seeking as businessmen.”

Agent-based modeling offers another key advantage for public choice scholars: the ability to merge related but distinct models. Consider two separate models, one of the budgeting process and another of the budget-maximizing bureaucrat. The bureaucrat may belong to a labor union that places pressure on the budgeters (Tullock 1987), but this connection between the two models is missing. Merging two mathematical models into one is usually not feasible, while merging two agent-based models is a much simpler task. This merging can be carried further, to bridge the gap between public choice and macroeconomics. In macro models that feature the accounting identity $GDP = C + I + G + NX$, G is usually taken to be either exogenous or a function of C , I , and NX —thereby completely ignoring over 50 years of study into the determination of G . An agent-based treatment of G , when merged with agent-based macro models, makes public choice directly relevant to macroeconomists.

1.5 Gordon Tullock: agent-based modeler

The roadmap for the incorporation of agent-based modeling into public choice was defined by Gordon Tullock himself. While exploring what he called “the General Irrelevance of [Arrow’s] General Impossibility Theorem” (Tullock and Campbell 1970, 98), Tullock created—in 1970!—the first agent-based model in public choice. In “Computer Simulation of a Small Voting System,” Tullock and Campbell describe a computer program to explore cycling when a committee faces multiple motions, each of which has multiple dimensions. “Because most issues in the real world probably have more than one dimension,” they write, a computer model “should give a more realistic

measure of the importance of cycles in small voting bodies than has been made thus far” (ibid., 99). They invoke their model with up to five dimensions, up to six motions, and up to 25 committee members. “The most interesting feature” of their model, they report, “is that the difference made by adding more dimensions is small” (ibid., 101). They also find that in a two-dimensional issue space with three voters, there can be no cycles. “Although this is obvious when analysed,” they write, “it was at first unexpected” (ibid., 103). This production of unexpected insights is one of the promises of agent-based modeling.

Despite its usefulness, this would be the last agent-based model Tullock would create. One can only speculate on the reasons he did not continue his foray into agent-based modeling. Perhaps the technology at the time proved too unwieldy for extensive use; their model was, after all, run on punch cards. Perhaps the computers of the time were just too slow. Perhaps no one asked for the computer code that they offered to provide upon request. Perhaps the prospect of continuing to implement models in Fortran and Algol simply did not appeal to Tullock, or to others.

It is certain, however, that Tullock did not abandon the approach for methodological reasons. A strict adherent of methodological individualism, Tullock’s models are unfailingly expressed in terms of agents rather than aggregates. In “Information and Logrolling,” for example, Tullock (1983a, 33) creates several agent-based models “intended to be small-scale models of real-world situations.” He creates models “with a very small number of people” because “such examples are easier to deal with” (ibid., 33). But while using a small agent population “does not affect the conclusion,” it nonetheless

imparts an “aura of unreality” to the analysis, and this Tullock laments (ibid., 33). It is regrettable that Tullock did not create an agent-based model in this instance. By doing so he would have dispensed with the “aura of unreality” and bolstered his claim that the small population “does not affect the conclusion” (ibid., 33).

Tullock’s recognition of the importance of agent heterogeneity is a recurring theme in his work. In “Hotelling and Downs in Two Dimensions,” Tullock (1967a) extends Hotelling’s and Downs’s spatial model to multiple dimensions. He notes that in Downs’s version of the model, “the distribution of the voters becomes crucial” (ibid., 56). Concerning that aspect, Tullock notes that the “distribution of the nonvoters would be quite different in different models, and this could be investigated quite easily” (ibid., 57). Tullock then analyzes logrolling in a similarly multidimensional space, pausing to note with regret that, due to the limits of his geometric approach, “we will have to represent the preferences of interest groups, not individuals. Interest groups, of course, are built up out of individuals, and normally do not represent a group of people with identical preferences, but people who feel strongly on one issue and less strongly on others” (ibid., 58). In this case Tullock has taken an approach common in economics: he has abstracted away from heterogeneity in favor of “an approximation of the interest group [average] preferences” (ibid., 58). It is clear that Tullock regrets having to do this.

A few years later, in “A Simple Algebraic Logrolling Model,” Tullock (1970) switches to an agent-based approach. Tullock considers “those situations in the real world in which we observe logrolling” and writes that “we observe immediate differences in the structure of the individual preferences... individuals are assumed to have intense

preferences on certain subjects” (ibid., 420). He then illustrates his model with three- and five-dimensional examples. In each version, Tullock demonstrates that in the presence of logrolling, “less than a majority of the voters... may be able to control the outcome” (ibid., 423). The distribution of preferences, coupled with differences in district configurations, determines the degree to which a minority of voters can do this. Tullock concludes by stating that “this model... provides a basis for future research by demonstrating that it is possible to obtain [Buchanan and Tullock’s 1962] conclusions which differs from the widely used spatial models only by a minor change in parameters” (ibid., 424). A decade later he followed his own advice and explored the issue further in “Why So Much Stability?” (Tullock 1981). He extends his five-member legislative model to 25 members and attempts to explain why, in such a system, “not only is there no endless cycling, but acts are passed with reasonable dispatch and then remain unchanged for very long periods of time” (ibid., 189). An agent-based model of logrolling, calibrated with an agent population reflecting a country’s actual population and the form of its legislature, would be the ideal vehicle to verify Tullock’s answer: that “this stability [is not] a true equilibrium,” but rather “a random member of a large set [of outcomes that] will be left unchanged for long periods of time” (ibid., 189).

Tullock will always be associated with the concept of rent seeking. In “Efficient Rent Seeking” (Tullock 1980), Tullock considers rent seeking as a game with a small number of heterogeneous players. He begins with an assumption that “the individuals can figure out the correct strategy” for optimal bidding for rents, and that “they assume that the other people will be able to figure it out” as well (ibid., 101). He builds an

example of a rent-seeking game with eight different functional forms for probabilities of winning and different numbers of players (2, 4, 10, and 15). From a social welfare point of view, this game can have three different outcomes, which Tullock calls “zones.” In the first zone, “expectancy of the players, if they all play, would be positive”: the prize exceeds the cost of obtaining the prize (ibid., 102). In the second zone, “the sum of the payments made by the individual players is greater than the prize; in other words, it is a negative-sum game” (ibid., 104). In the third zone, “the individual players make payments that are higher than the prize. It might seem obvious that no one would play games of this sort, but, unfortunately, this is not true” (ibid., 104). The implications are obvious: “as a good social policy, we should try to avoid having games that are likely to lead” to zones two and three (ibid., 109). We should “attempt to lower the cost of rent seeking, and ... move ... into zone I” (ibid., 112). But how, in the presence of heterogeneous agents, does one structure the game to accomplish this result? Agent-based modeling provides a solution: by extending Tullock’s 15-agent example to larger, more representative populations, one can realistically compare different versions of the game.

The creation of small, illustrative, example-oriented models continued throughout Tullock’s professional life. In “A New and Superior Process for Social Choices” (Tideman and Tullock 1976), Tullock and coauthor T. Nicolaus Tideman turn their attention to demand revelation. They describe how “Vickrey showed that it would be possible to motivate individuals to reveal their true supply and demand schedules for a private good” and thus extend that concept to public goods (ibid., 1146). They begin

with “two alternatives, which may be conceived of as two policies or two candidates” (ibid., 1147). They “then show how the process can be extended to more than two options” (ibid., 1147). The important feature of their model is that individuals may have different preferences. In a reply, Riker (1979) takes issue with the conclusions reached by Tideman and Tullock. Riker offers an agent-based model of his own, with nine classes of voters. Tideman and Tullock (1981, 325) respond to “the last case given by Riker, in which there is in excess of three million voters, with a coalition of one hundred and ten of these” able to hold sway over the majority. Tideman and Tullock conclude that “Riker agrees with us that coalitions are possible in the demand revealing process, but apparently he does not agree with us that they are much less likely than in ordinary voting. It can be said, however, that he has not demonstrated that they would be more likely than they are with ordinary voting, only that they can exist” (ibid., 328). In this exchange, the two sides are arguing about a set of questions that easily could be answered by an agent-based model, the workings of which are not in dispute.

Tullock again turns to small example models in “Income Testing and Politics” (Tullock 1982). He begins with a model that “has been designed ... to be extremely simple and straightforward and hence easy to follow” (ibid., 99). He had noted that “no direct transfer of the conclusions from the model into the real world is possible,” because “the consequences we draw from this political model are very heavily affected by the detailed assumptions about things such as preference curves and number of people” (ibid., 99). In such a limited model, he suggests, it would not be possible to include enough real-world detail to produce real-world conclusions. Tullock nevertheless

combines the model's results with "unfortunately not very sophisticated" empirical evidence to conclude that "in most of the cases in which income-tested programs have been converted into universal programs, the poor have been injured" (ibid., 116). This is a strong statement. In the 30 years that have passed since he made it, more empirical results have surely been obtained. Those results, in conjunction with a stronger, agent-based model, could strengthen or refute Tullock's conclusion.

Tullock explores government redistribution further in "Horizontal Transfers" (Tullock 1983b). In an essay nearly a decade earlier, Tullock (1974, 7) had observed that governments transfers do not necessarily flow to the poor from the rich, but rather to parties with "sufficient political influence to initiate the transfer" from parties whose "political influence proved insufficient to stop it." In "Horizontal Transfers" he constructs a model to explain why. He begins "with a simple model, which the reader may think unrealistic" (Tullock 1983b, 24). He agrees that the model has unrealistic elements, but that "the model is much discussed in the relevant public choice literature, and there are certain aspects of the model that can be observed in the real world" (ibid., 24). His model examines transfers among five agents, each of whom votes on redistribution policy. "The net result," he suggests, is "that everybody has the same amount of money that they entered with" (ibid., 28). In real life, this result "would not be expected with only five voters," but with "a larger number of voters it is not unrealistic" (ibid., 28). An agent-based model would put Tullock's claim to the test.

An important element of agent-based modeling is emergence, another feature of Tullock's research. In "Proportional Representation," Tullock (1967b) discusses how

proportional representation arose in France. The French system “differs from the Anglo-Saxon system simply in that if no candidate gets a majority in the first balloting, a runoff is held in a few weeks in which only a plurality is necessary for election. These simple rules, together with the French talent for intrigue, have led to a functioning proportional representation system” (ibid., 148). In other words, a system “approximating proportional representation” has emerged from the combination of agent behaviors and institutional rules (ibid., 148). Would such a system produce similar results in England? An agent-based model would allow exploration of the extent to which this system is applicable to other (non-French) distributions of agent behavior. In “The Politics of Persuasion,” Tullock (1967c) examines the emergence of consensus. Tullock was famous for not voting in national elections; the benefit was simply not worth the cost. But his extensive writings show that he did engage in persuasion. The ability to persuade others “is more likely to affect the outcome of [an] election than is voting,” Tullock states, with the caveat that “[t]here will be very great variation” in individuals’ abilities to persuade (ibid., 124). To quantify the effects of this, he creates a ten-agent, ten-opinion model, and shows how majority opinion emerges. In this model, Tullock demonstrates two virtues of agent-based modeling. First, different distributions of agents can produce different patterns of outcomes. Second, the persuasion model can be combined with the earlier logrolling models to predict actual election results. Voters might, for example, be persuaded that ethanol subsidies are inefficient; yet these same voters elect representatives who support ethanol subsidies. The creation of a single model that explains both phenomena is a topic worthy of research.

Above all, Tullock never loses sight of the fact that agent behavior is the key to understanding any problem in public choice. When Tullock turns to the question of voter turnout, for example, he asks: “What is the payoff to the individual from voting?” (Tullock 1967d, 108). A voter for whom “the estimated [benefit] is less than the cost of becoming informed ... will not bother” (ibid., 103). The resulting concept of rational ignorance explains how interest groups come to dominate the political process: “the politician, in making up programs to appeal to rationally ignorant voters, would be attracted by fairly complex programs which have a concentrated beneficial effect on a small group of voters and a highly dispersed injurious effect on a large group of voters” (ibid., 103). Rational ignorance also explains why “charitable activity is likely to be badly designed and ineptly carried out” (Tullock 1966, 142). Donors “are apt to be exceptionally ill informed about the effects of their gifts” (ibid., 142) because “incentives for becoming well informed are extremely weak” (ibid., 146). Even the irrational can be rational in the right circumstance: in dangerous situations, “[y]ou might threaten your opponent with irrational behavior on your part and the threat is indeed rational” (Tullock 1972a, 66). Tullock then says that the “existence of this type of loss of temper then automatically produces a bargaining range” (ibid., 66). (The behavior of Iraqi leader Saddam Hussein, prior to the US invasion of 2003, is a relevant case study: Hussein sought to produce, and indeed did produce for a period of time, a bargaining range.) In modeling revolutions, Tullock observes that “if we are attempting to study the dynamics of the revolution... we should turn to examination of the utility calculus of the participants” (Tullock 1971, 94). Revolutions occur so infrequently, Tullock says,

because a single participant's risk is likely to be greater than the potential reward. Yet revolutions do, in fact, occur—though, as Tullock suggests, most forced changes of government come through coups d'état. In each of these instances, Tullock resorts to analysis of the individual. Tullock, it might be said, is a natural-born agent-based modeler.

1.6 Conclusion

The history of modeling in political science has paralleled the history of modeling in economics. In the beginning, economists and political scientists modeled by narrative. Then came the explosion of mathematical economics. During the postwar period, mathematical modeling came to dominate the economics profession; not long thereafter, the economic style of modeling “invaded” (Tullock 1972b, 317) other social sciences. The invasion of political science was led by Tullock, among others, who combined to produce “a sizeable literature by economists and use of recognizable economic methods in the field normally described as political science” (ibid., 317). This approach to political science would come to be known—to Tullock's occasional dismay—as public choice. The world of economic modeling has, in the meantime, continued to evolve, and agent-based modeling offers the potential to address many problems in economics that hinge on heterogeneity, network effects, and emergence. The world of public choice modeling is not far behind, and one of its founders, Gordon Tullock, has already shown the way.

2. AN AGENT-BASED MODEL OF REGULATORY CAPTURE

2.1. Introduction

“People of the same trade seldom meet together, even for merriment and diversion, but the conversation ends in a conspiracy against the public, or in some contrivance to raise prices.” So said Adam Smith (1994 [1776], 148), in a famous quote on the formation of cartels. Taken out of context, Smith’s quote would seem to be a blanket indictment of business owners: left unchecked, they will collude, to the detriment of their customers. Standing alone, this quote is easily read as a justification for government regulation of commerce. If businesses will inevitably collude, the public must be protected from them.

Yet Smith, in this famous quote, is not talking about how government regulation can protect the public from predation. In fact, he is talking about the exact opposite. His quote occurs in the middle of a passage on government regulation, and he is debunking the notion that regulation exists for the benefit of the consumer. Instead, Smith claims, regulation exists for the benefit of business—and the government enforces regulation not to protect the consumer, but in exchange for a cut of the rent. “The pretence,” Smith writes, that regulations “are necessary for the better government of ... trade, is without any foundation” (ibid., 149). The “prerogative of the crown” to establish regulation

“seems to have been reserved rather for extorting money from the subject, than for the defence of the common liberty against ... oppressive monopolies” (ibid., 143). “The government of towns ... was altogether in the hands of traders and artificers; and it was the manifest interest of every class of them, to prevent the market from being overstocked ... with their own particular species of industry” (ibid., 143).

Smith does not use the term “regulatory capture”; and given his views, it is possible that he would have objected to it. To call this phenomenon “regulatory capture” is to imply that regulation has somehow gone wrong—that it started out on the right track, but was then “captured” by the interests it was meant to control. Smith is clearly expressing a more jaundiced viewpoint: the very *purpose* of regulation—at least the sort of regulation that Smith has in mind—has always been the establishment of cartels. Regulation has not been “captured” so much as bred in captivity. “It is to prevent [a] reduction in price, and consequently of wages and profit, by restraining that free competition which would most certainly occasion it, that all corporations [by which he means civic regulatory bodies] ... have been established,” says Smith (ibid., 142). And how does this happen? “The trades which employ but a small number of hands, run most easily into such combinations,” he says (ibid., 145): a small number of merchants can easily organize a cartel. And these cartels are “seldom opposed” by the citizens, who “have commonly neither the inclination nor fitness” to organize (ibid., 147). In fact, not only do they not resist the cartel, they may even *support* it: “the clamor and sophistry of merchants and manufacturers easily persuade them that the private interest of a part, and of a subordinate part of the society, is the general interest of the whole” (ibid., 147).

The idea that governments regularly conspire with businesses to swindle the consumer is controversial. At some level, one simply does not wish to believe it—not in a democracy, at any rate. Or one might believe that this occurs from time to time—“crony capitalism”—but that it is the exception rather than the rule. As a description of crony capitalism, Smith’s story is perhaps believable. But is Smith, in 1776, decrying crony capitalism, or regulation in general? Regulation encompasses more than cartels; in the modern era it includes the Food and Drug Administration, the Clean Air Act, the Securities and Exchange Commission. Smith lived before the Industrial Revolution; he predated the Clean Air Act by two centuries. Did Smith mean to indict all regulation as a tool crafted by rent-seeking businesses? Is it fair to apply Smith’s words to regulation in the modern world?

According to Samuel Peltzman (1989), the consensus answer before the 1960s was no. In fact, until the 1960s even the most narrow interpretation of Smith’s claim—that regulation of monopolies served businesses rather than consumers—was largely ignored. “Until the early 1960s,” Peltzman writes, “the prevailing theory of regulation was ... called the ‘normative analysis as a positive theory’ ... This theory ... regarded market failure as the motivating reason for the entry of regulation” (ibid.). Regulation existed “to lessen or eliminate the inefficiencies engendered by ... market failure” (ibid.). The “most popular culprit was natural monopoly,” he says (ibid.). “The main role of utility regulators was held to be prevention of private exploitation of the market power that would inevitably flow from natural monopoly cost conditions” (Peltzman 1993). Of course, monopoly power was not the only market failure for which regulation was the

cure. Indeed, Peltzman wryly notes, the “ingenuity of economists ensures that the list of potential sources of market failure will never be complete” (Peltzman 1989). And “credulity is strained when the list of market failures grows at roughly the same rate as the number of regulatory agencies” (Peltzman 1993). Nevertheless, for two centuries, Smith’s depiction of regulation was simply not considered a serious theory of regulation.

In those two centuries, regulation expanded far beyond the scale ever witnessed by Smith. Smith recognized that geography was a key factor in the establishment of regulation: the “inhabitants of a town, being collected into one place, can easily combine together,” he wrote, while the “inhabitants of the country, dispersed in distant places, cannot easily combine together” (Smith 1994 [1776], 145). Twenty-five years after his death in 1790, steam engines had begun to traverse Britain; fifty-five years after his death, telegraph lines connected Washington and Baltimore. Transportation and communication were making the world smaller. And with a smaller world came greater opportunities to organize, and to regulate. Between 1870—just eighty years after Smith’s death—and the 1930s, “regulation proliferated over railroads, gas and electric utilities, radio and television broadcasting, truck and air transport,” and other enterprises (McCraw 1975). Yet Smith’s unromantic—some would say cynical—view of regulation gained little traction. The “reasons for regulation varied according to the industry involved,” says McCraw, but “the notion of the ‘public interest’ continued to dominate the rhetoric of reformers, the utterances of presidents, and the decisions of commissioners” (ibid.). Lawyers were regulated by 1890; dentists by 1900; embalmers by 1910; beauticians and barbers by 1930 (Stigler 1971).

Smith had correctly foreseen that a shrinking world would mean more regulation. The progress of technology ensured that this prediction would have come true under any regime. But the arrival of the Great Depression, and the New Deal that it hatched, spawned an unprecedented increase in regulatory zeal. Franklin Delano Roosevelt, one might say, was never one to let a crisis go to waste. Three months after taking office, Roosevelt signed into law the National Industrial Recovery Act (NIRA), describing it as “the most important and far-reaching legislation ever enacted by the American Congress” (Barber 1996, 29). The NIRA “authorized the president ... to create an administrative apparatus to approve ‘codes of fair competition’ submitted by trade and industrial associations (ibid., 29). A “flood of regulatory law” followed—all of it designed, ostensibly, to protect the public from “enterprises ... with a propensity toward corruption” (McCraw 1975). Roosevelt’s administration achieved a “multiplication of regulatory agencies” on a scale never before seen in the United States (ibid.). The NIRA, the Agricultural Adjustment Act, the Securities Act, the Securities and Exchange Act, the National Housing Act, the National Labor Relations Act, the Social Security Act, and the Fair Labor Standards Act were all passed during the regulatory outburst of the New Deal (Funk and Wagnalls 1971, 17:277).

It was in the wake of the New Deal that Smith’s skepticism toward the “public interest” view of regulation began to grow tiny roots. In 1936, Roosevelt established the Committee on Administrative Management, tasked with streamlining the executive branch. Its report the following year described a problem that Roosevelt had not expected: the regulatory agencies had become a “headless fourth branch of government”

(Curtin 2009, 51), accountable to neither the president nor Congress. In response to this perceived attack on regulation, Roosevelt's appointee to the Securities and Exchange Commission, James M. Landis, wrote his 1938 defense of regulation, The Administrative Process. Regulatory agencies, says Landis, act not from self-interest, but rather with "judgment [that] flows from a determination to promote the public interest" (Landis 1938, 64). He rejects the "fourth branch" criticism resoundingly: the "desirability of four, five, or six 'branches' of government would seem to be a problem determinable not in the light of numerology but rather against a background of what we now expect government to do" (ibid., 47). Landis defended regulation boldly: regulation is what government *should* be doing. To regulate is to serve the public.

The outbreak of war put a temporary end to the debate over regulation. World War II saw the most extensive government involvement in the economy in U.S. history, with price controls imposed on goods comprising about 20% of the typical consumer's market basket (Rockoff 1984, 127). The argument that regulation amounted to a conspiracy between government and businesses gained no traction in an atmosphere of war, even one fought on distant continents. By the end of the war, Roosevelt was dead, and so was the Depression. But most of the New Deal agencies survived.

The postwar period saw a nascent return to skepticism about regulation. Commentators began to notice how regulators "seemed to follow doctrines that equated the 'public interest' with whatever the most powerful elements of [industry] wanted" (McCraw 1975). The first formal diagnosis of regulatory capture came in 1952, when Samuel Huntington claimed that the Interstate Commerce Commission, tasked with

regulating the railroads, had ended up serving the railroad industry instead. “When such a commission loses its objectivity and impartiality by becoming dependent upon the support of a single narrow interest group,” wrote Huntington (1952), “obviously the rationale for maintaining its independence has ceased to exist.” He recommended its abolition. Three years later, Marver H. Bernstein’s book Regulating Business by Independent Commission generalized Huntington’s analysis, describing how business interests might “capture and control” the agencies charged with their regulation (Bernstein 1955, 146). And five years after that, in 1960, James M. Landis—the staunch defender of regulation in 1938—performed a “stunning turnaround,” submitting to President-elect Kennedy “a report roundly criticizing [regulatory] commissions and recommending sweeping reforms” (Ritchie 1980).

The stage had now been set for a paper that, in Peltzman’s (1993) words, “profoundly affected the course of intellectual inquiry” into the theory of regulation. George Stigler and Claire Friedland posed an innocent-sounding question: if regulation mitigates monopoly power, shouldn’t we see a price difference between regulated and unregulated markets? Stigler and Friedland (1962) provided an answer that few expected: in a study of regulated and unregulated providers of electricity, regulation seemed to have no effect on prices. There is a certain irony in the fact that this result turned out to be incorrect; as Peltzman (1993) explains, a coding error led Stigler and Friedland to understate the effects of regulation by a factor of 10. Nevertheless, the result prompted several hundred more empirical studies of regulation. Many of these studies suggested that regulation of monopolies had little to no effect on consumer prices, while

regulation of competitive industries tended to *raise*, not lower, consumer prices (ibid.).

Both of these results run counter to the view that regulation benefits the consumer.

There is no evidence that Stigler and Friedland had Adam Smith's hypothesis in mind when they presented their results. Instead they provide two rather innocent explanations for their findings of "the ineffectiveness of regulation" of electricity: the lack of "any large amount of long run monopoly power" for energy production, and the inability of the regulator to "[force] the utility to operate at a specified combination of output, price, and cost" (Stigler and Friedland 1962). But these were not compelling explanations. "As with much else in economics," Peltzman (1993) writes, "evidence preceded theory. In this case, the evidence ... seemed to ask for an explanation of why regulation had come to work in this seemingly perverse way." Nine years after his article with Friedland, Stigler would provide a new explanation for the phenomenon they had observed—an explanation that echoes Adam Smith.

Stigler (1971) offers the first economic theory of regulation. Before this landmark paper, "normative analysis as a positive theory" held sway: regulation exists because it is in the public interest, and government's role is to pursue the public interest. Stigler offers a very different explanation for the existence of regulation: regulation exists because there are entities that demand it, and there are entities that can supply it. Stigler could not have been more clear about this: "as a rule," he says, "regulation is acquired by ... industry and is designed and operated primarily for its benefit" (ibid.). "[E]very industry or occupation that has enough political power to utilize the state" will do so (ibid.). The view that "regulation is instituted primarily for the protection and

benefit of the public at large” is “idealistic” (ibid.). The “problem of regulation” is not why it sometimes goes wrong, but “why an industry ... is able to use the state for its purposes” (ibid.).

Stigler’s explanation is almost identical to that provided by Smith. “The state has ... the power to coerce,” and will use this power if it is paid to do so (ibid.). “The industry which seeks regulation must be prepared to pay” (ibid.). “These costs ... increase with the size of the industry seeking the legislation” (ibid.). Stigler even invokes the same city/country distinction described by Smith: “expenses in the solicitation of support ... are higher for a diffused occupation than a concentrated one” (ibid.). Stigler expresses the same deep capture that Smith described. Stigler does, however, extend Smith in three ways. First he explains how regulatory capture can exist even in a democracy. In an unattributed allusion to Downs (1957), he explains that the “voter’s [efforts] to learn the merits of individual policy proposals ... are determined by expected costs and returns” (Stigler 1971)—it is not rational for most voters to care. In consequence, the regulator ignores them entirely. Second, he expresses the supply-demand equilibrium explicitly. Third, he extends the collective action problem to free riders, as in Olson (1965).

Stigler’s colleague Richard Posner describes Stigler (1971) as “pathbreaking” (Posner 1974). Posner calls the public interest theory of regulation “unacceptable,” “unsatisfactory,” and “contradicted by ... evidence” (ibid.). In contrast, the economic theory described by Stigler is “precise and hard-edged” and “illuminating” (ibid.). But Posner’s assessment of Stigler’s theory is not uniformly positive. The economic theory is

“promising,” but cannot “be said to have, as yet, substantial empirical support” (ibid.). The theory “has not yet been refined to the point where it enables us to predict specific industries in which regulation will be found” (ibid.). The “economic theory is still so spongy that virtually any observations can be reconciled with it”(ibid.). A particularly puzzling weakness, according to Posner, is the fact that “so many regulated industries appear to be either extremely atomistic (like agriculture) or extremely concentrated (like local telephone or electrical service)” (ibid.). How can the economic theory explain this fact pattern?

Peltzman was also impressed by Stigler’s “pioneering” work, describing it as “one of those rare contributions—rare for the rest of us, though not for him—which force a fundamental change” (Peltzman 1976). Stigler was able “to crystallize a revisionism in the economic analysis of regulation that he had helped launch in his and Claire Friedland’s work” (ibid.). It is clear that Peltzman admired Stigler, because Peltzman takes particular care to acknowledge his “great intellectual debt to Stigler” while nevertheless expressing “dissatisfaction with some of Stigler’s conclusions” (ibid.). That dissatisfaction, and apparent encouragement from Posner, led Peltzman to write his “extension and generalization” (ibid.) of Stigler (1971).

Peltzman (1976) formalizes Stigler’s model. Peltzman begins with an elected regulator/legislator who wishes to maintain office. Maintaining office means winning a majority of votes. Votes are gained by conferring benefits and lost by imposing taxes. Benefits accrue to small, concentrated groups, while taxes are imposed on large, unorganized groups. Arranging the transfers is not without cost: the beneficiaries must

be organized, and the voters may need to be placated in some fashion. (Peltzman calls this “education”; Smith calls it “clamor and sophistry.”) Imposing taxes also means deadweight costs—a restriction that Peltzman describes as “less innocent than it appears,” as it “rules out ... transfer[s] ... with no allocative effects” (ibid.).

The voter plays a more significant role in Peltzman’s model than in Stigler’s formulation. In Peltzman’s model, “the political process must pay [heed] to marginal opposition” (ibid.). Peltzman thus answers one of Posner’s key questions. Monopolies are regulated because the regulator can seize a portion of their rents and refund them to voters; perfectly competitive industries are regulated because the regulator can help generate rents at little political cost. The fact that the equilibrium is somewhere in the middle explains why regulated industries are “either extremely atomistic ... or extremely concentrated” (Posner 1974). As Dal Bo (2006) puts it, the “incentives for ‘regulatory entry’ appear highest when industry is fully monopolistic or perfectly competitive to begin with, because the power gains from moving price towards a middle range are highest.” Peltzman’s model also explains why cross-subsidies exist: “minimization of opposition ... from consumers” can be achieved by “exploiting differences among them” (Peltzman 1976). For example, consider the market for long distance telephony. The regulator benefits from a three-part regulatory structure: restrictions on competition for providers, paid for by (implicit) taxes on users, with an offsetting subsidy for rural users. When “one group of consumers has sufficiently large per capita demand ... relative to the other group, the latter may become part of the winning group” (ibid.). In such a situation,

“political entrepreneurship will produce a coalition which admits members of the losing group into the charmed circle” (ibid.).

A sharp distinction had now been drawn between two theories of regulation: the public interest theory and the economic theory. Readers familiar with the names Stigler, Posner, and Peltzman will have recognized that all three were at the University of Chicago. It is perhaps for this reason that the economic theory of regulation is associated with “the Chicago view of public policy,” although it could equally “be seen as complementary to the emerging literature on *public choice*, which was ... associated with the names of Buchanan, Tullock, and the Virginia school” (Dal Bo 2006). Another Chicago professor, Gary Becker, found himself “stimulated by the atmosphere created by Stigler, Peltzman, Posner, and others” (Becker 1983), and crafted in response a somewhat different take on regulation. Becker accepts Peltzman’s model as his starting point. He begins by noting that the “political effectiveness of a group is mainly determined not by its absolute efficiency ... but by its efficiency relative to [that] of other other groups” (ibid.). A group benefits when it becomes *relatively* more efficient. Competition among interest groups will therefore tend to promote efficiency. The implication is a reconciliation between the public interest theory and the economic theory: regulation prompts a pursuit of efficiency; pursuit of efficiency is welfare-enhancing; thus regulation serves the public interest. “Becker argues that the political process will be drawn toward efficient modes of redistribution in general and to efficiency-enhancing regulation in particular” (Peltzman 1989). One might playfully paraphrase Becker by

saying that the regulator intends only his own gain, and he is in this led by an invisible hand to promote an end which was no part of his intention.

The writings of Stigler, Posner, Peltzman, and Becker allowed the economic theory of regulation to gain widespread acceptance in the 1970s and 1980s. “By conventional measures,” writes Peltzman, “the theory has been an academic success” (ibid.). The “literature as a whole has made its mark on academic analyses of regulation” (ibid.). It is interesting to note that this “academic success” occurred at exactly the same time that the United States was witnessing widespread *deregulation*. The country saw a “substantial elimination of regulatory constraints,” writes Peltzman, “unprecedented in modern American history” (ibid.). “[E]ven as the ink was drying” on the economic theory of regulation, “deregulation was sweeping aside many long-standing barriers to competition” (ibid.). The theory was a success—except that “[n]ot one economist in a hundred practicing in the early 1970s predicted the sweeping changes that were soon to happen” (ibid.). Peltzman takes this in stride; this was “hardly the first or last forecasting failure in economics” (ibid.). Nevertheless, “the fact that deregulation was such a surprise partly reflects ... some general problems in the theory of regulatory entry and exit” (ibid.). Peltzman describes deregulation as “one plausible response to forces that called for regulatory change,” but “not ... the only plausible response”; “more or different regulation would have been an equally plausible response” (ibid.).

Peltzman considers “changes in the ... ‘economics’ of the regulated industries” (ibid.) as a possible impetus for deregulation—a theme picked up by several later researchers. Kroszner and Strahan (1999), for example, point to “broad technological,

legal, and financial innovations that altered the costs and benefits of ... regulations” as factors in the deregulation of banking. Goff (1996, 133) says that “support [for deregulation] within the air carrier industry for deregulation came primarily from innovative firms like Federal Express.” In the electricity industry, “[t]echnological change has reduced the economies of scale in power generation to a fraction of what they were in preceding decades” (White 1996). “These changes arrive on the heels of more than two decades of deregulatory activity” (ibid.). But Peltzman would not have been convinced by this piecemeal defense of the economic theory. The economic theory “purports to be a general model of the forces affecting regulation,” and Peltzman cannot escape the conclusion that “the deregulation movement was selective” (Peltzman 1989). Instead he chooses to “eschew a special-purpose absolution” of the economic theory (ibid.). Peltzman concludes that the economic theory is “a modest step” that has proven to be useful, but that “a full analysis of the scope and form of ... the institutional underpinnings of regulation ... remains unwritten” (ibid.).

Of course, it is possible that Peltzman is simply too modest. It is possible that the deregulation movement in the 1970s was akin to the repeal of the Corn Laws in 1846: a political shift induced by clear-headed arguments. Peltzman is willing to claim that had these “deregulation initiatives ... been put to a vote of the American Economic Association membership, all the initiatives would have passed with large majorities” (ibid.). He goes on to say that “not since the rise of free trade in the nineteenth century has so broad a professional consensus been so well reflected in policy” (ibid.). He is

unwilling to take credit for this fortuitous shift in policy, but it cannot be said that such credit belongs anywhere else.

This chapter presents an agent-based rendering of Peltzman's 1976 model of regulation. The technique of transforming a neoclassical model into an agent-based model is called "agentizing" (Guerrero and Axtell 2011). As is often the case with agentized models, the objective is to supplement, rather than supplant, the original formulation. In particular, the agent-based approach permits exploration of the effects of agent heterogeneity on regulatory regimes. This implementation extends the original model by allowing bilateral regimes, and permits evaluation of the conclusions regarding homogeneity and the size of government found in Peltzman (1980).

2.2. The model

The model is written in Python, a general purpose computer language. Python was chosen because it employs a rich and expressive syntax, is available at no charge¹ for many computer platforms, and encourages a concise programming style that is conducive to publication. A model written in Python is likely to be about a fourth as long as the same model written in Java (Lutz 2011, 1548). (For an introduction to programming agent-based models in Python, see Downey (2012), chapter 10.)

The basic unit of the model is the citizen. Each citizen is defined by five attributes. The first attribute is the citizen's potential income: the income the citizen would receive in the absence of regulation. The second attribute is a measure of how the

¹ Python may be downloaded from www.python.org.

citizen reacts to increases in taxes, with 1 indicating no reaction, greater than 1 indicating a decrease in effort, and less than 1 indicating an (improbable but theoretically possible) increase in effort. The third attribute is the citizen's ex-regulation approval of the regulator, with 1 indicating full approval, 0 indicating full disapproval, and 0.5 indicating indifference. The fourth attribute is a measure of how the citizen's approval of the regulator responds to changes in income. The response to a decrease in income is always negative. A value of 1 means an approval response in constant proportion to the decrease; greater than 1 means an increasing marginal response; and less than 1 means a decreasing marginal response. The fifth attribute is the cost required to persuade the agent to increase her support of the regulatory regime by one percentage point. The agent-based nature of the model derives from citizen heterogeneity; no assumptions are imposed concerning the distributions of these attributes across citizens.

Each citizen belongs to exactly one group. A group may receive transfers from any other group. In order to procure those transfers, a group may incur three types of costs: a cost to persuade other groups, a cost to persuade the regulator, and a cost to organize its own members. A tax is levied on each group in order to raise the necessary transfers. By default, each group member pays the same tax rate, although this restriction is later relaxed. The interpretations of "income" and "tax" are the same as in Peltzman (1976): for "application to problems of regulation, [income] can be thought of as a typical consumer's surplus and [tax] a regulated price if producers are beneficiaries, or [income] might be a producer's surplus and [tax] the difference between the surplus-maximizing price and the regulated price where consumers are beneficiaries." In other

words, transfers may flow in either direction in the model. Unlike Peltzman's model, however, the agent-based model allows transfers in both directions at once.

The total set of transfers in the model is called the *regulatory regime*. Regulators are assumed to be elected by majority vote, so there is no distinction between the regulator and the legislator. In Peltzman (1976), the regulator's objective in crafting the regime is vote maximization: a larger vote margin is preferred to a smaller vote margin, even if both exceed 50%. In the agent-based model, the regulator may choose between maximizing behavior and satisficing behavior. (A satisficing regulator would be as content with 51% of the vote as with 100% of the vote.) Votes come from citizens. A citizen bases her vote on five factors: her prior approval of the regulator, her actual income under the regulatory regime, her potential income without the regime, her degree of responsiveness to changes in income, and the degree to which she can be persuaded to support a regime that is not otherwise in her own interests. The core model appears in Appendix 2.A.

Accompanying the model are three detailed implementations. The first implementation, `license.py`, describes a licensing regime. Citizens are divided into license holders and non-license holders, with the former receiving transfers from the latter. The license implementation appears in Appendix 2.B. The second implementation, `subsidy.py`, describes a regime of cross-subsidies. In this implementation, an otherwise unviable regime is rendered viable by splitting the citizens and granting a cross-subsidy to one of the subgroups. Labor unions are treated as a special type of cross-subsidy. The cross-subsidy implementation appears in Appendix

2.C. The third implementation, `bidirectional.py`, supports a regulatory regime characterized by bidirectional subsidies. This implementation extends Peltzman's 1976 model to his 1980 analysis of the growth of government, and appears in Appendix 2.D.

2.3. The licensing model

As a starting point, consider the licensing model. In the default version of this model, there are 10,000 citizens. In the absence of licensing, each citizen would earn \$15,000. Some citizens are licensed. Licensing produces rents of \$1000 for each licensee. Each licensee will pay \$100 in license fees, and it costs \$50 per licensee to organize support. A \$1000 payment in "education" increases support for the regime among nonlicensees by one percentage point. Natural entry and exit barriers to the industry are low. Assume that voters are initially indifferent to the regulatory regime, that there are no deadweight costs to taxes, and that their disapproval with decreases in income is linear. The following specification explores this situation, gradually increasing the percent of citizens with licenses from 1% to 99%:

```
import model, license
model.headings (["pct"])
for p in range (1, 100):
    pct = p / 100
    model.run (license.regime (percent=pct), ids=[pct], runs=1)
```

As the number of license holders increases, the volume of transfers increases, with more transfers coming from fewer non-licensees. As a result, the tax rate imposed on nonlicensees increases at a faster-than-linear rate. Figure 2.1 illustrates the results.

When the percent of citizens holding licenses reaches 80%, the tax rate on nonlicensees required to finance the transfer exceeds 100%, and the regime becomes unattainable.

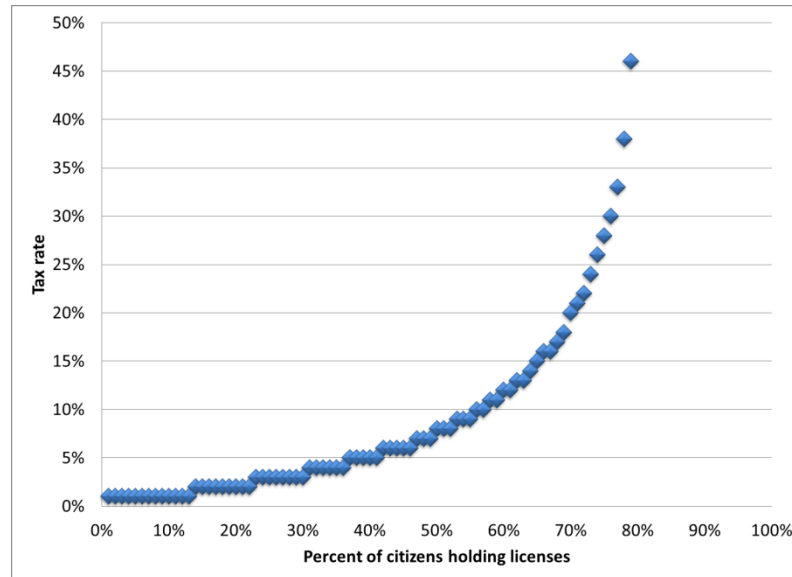


Figure 2.1: Tax rate by percent of citizens holding licenses

The results of both Peltzman (1976) and Becker (1983) rely on the existence of deadweight costs of taxation. When tax rates increase beyond a certain point, workers withdraw from the labor market. After-tax income equals gross income times $(1 - \text{taxrate})$, so deadweight loss enters the model as an exponent on $(1 - \text{taxrate})$, with larger values mean higher deadweight losses. The following specification considers different deadweight loss values:

```
import model, license
model.headings (["d", "pct"])
for d in [1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]:
    for p in range (1, 100):
        model.run (license.regime (percent=p/100, deadweight=d),
                    ids=[d, p/100], runs=1)
```

Figure 2.2 depicts the results. As potential deadweight losses increase, the maximum viable tax rate decreases. However, the effect is insignificant when the percentage of citizens holding licenses is small. Only when this percentage reaches 40% does the effect become noticeable.

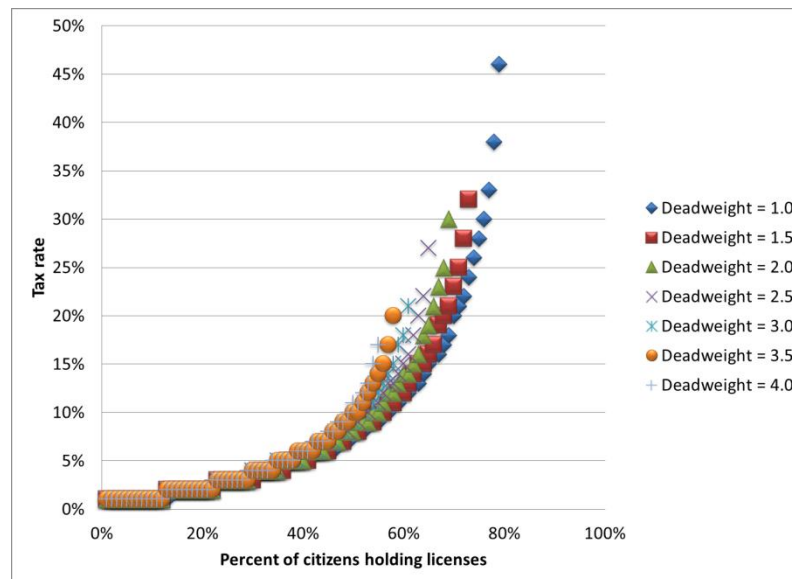


Figure 2.2: Tax rate by percent of citizens holding licenses and deadweight costs

Figure 2.2 hints at a fundamental concept behind successful licensing schemes: if both the number of licensees and the amount of the transfer are small, nonlicensees do not object to the transfer. But as the size of the transfer grows, opposition mounts. Consider a regime in which licensees comprise 1% of the population, with each licensee willing to spend up to 10% of rents to persuade nonlicensees (“education” or “clamor and sophistry”):

```

import model, license
model.headings (["d", "r"])
for d in [1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]:
    for r in range (0, 110000, 10000):
        model.run (
            license.regime (received=r, sent=r*0.1, deadweight=d),
            ids=[d, r], runs=100)

```

Figure 2.3 relates the amount of the transfer to the resulting approval rating of the regulator. Different deadweight costs produce very different results. When deadweight costs are low, the regulator benefits from imposing a regulatory regime. When deadweight costs are high, the regulator suffers. In between, the regulator benefits until the amount of the transfer reaches a critical point. The regulator loses support by imposing taxes but gains support by educating voters; the deadweight cost parameter determines the relative strengths of these opposing forces.

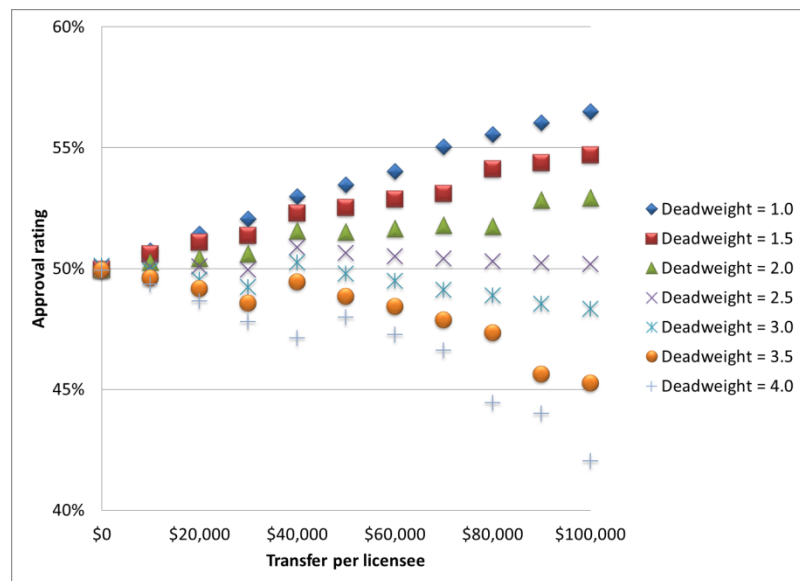


Figure 2.3: Approval rating by transfer per licensee and deadweight cost

The preceding results illustrate some of the comparative statics derived by Peltzman (1976). Peltzman's model assumes that all members of a group are homogeneous. A key benefit of the agent-based approach is the ability to relax the assumption of homogeneity without switching to a different model. A straightforward extension of the previous example is the introduction of a *distribution* of deadweight costs. The following code creates agents whose deadweight costs are distributed lognormally. The lower bound of the distribution is 1.0, meaning no deadweight costs; there is no upper bound. The sigma parameter dictates the degree of dispersion, with zero meaning no dispersion (all agents share the same deadweight costs). Seven values of sigma are explored:

```
import model, license
model.headings (["sigma", "r"])
for sigma in [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]:
    for r in range (0, 110000, 10000):
        model.run (license.regime (received=r, sent=r*0.1,
                                   deadweight=2.5, deadweight_sig=sigma), ids=[sigma, r])
```

Figure 2.4 shows the results. When dispersion is zero, the homogeneous result obtains: the regulator's approval rating hovers just above 50%. As dispersion increases, support for the regulatory regime wanes. The higher the dispersion, the faster the decline in approval. This explains why it "will pay the rational regulator to exploit differences within the group that, taken as a whole, either wins or loses" (Peltzman 1976). If a subset of the nonlicensee group has high deadweight costs, it is in the interest of the regulator to identify and isolate that subset.

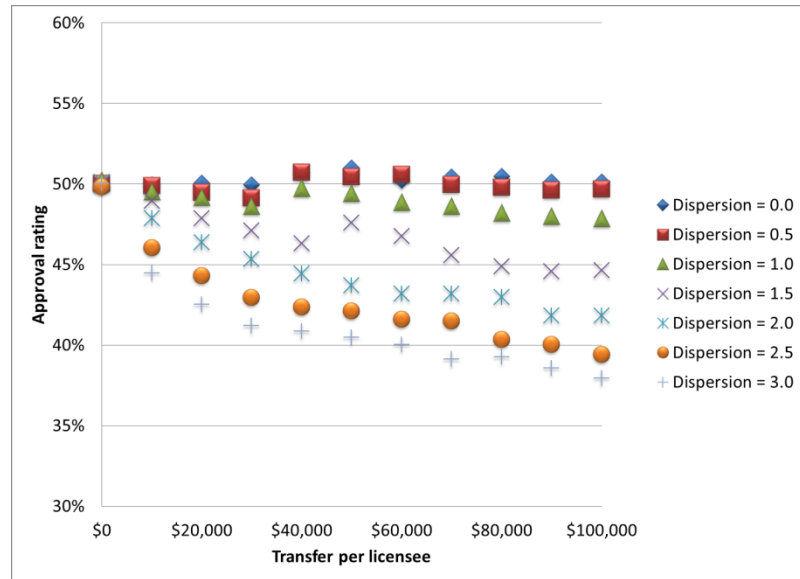


Figure 2.4: Approval rating by transfer per licensee and dispersion of deadweight costs

The homogeneity assumption can be relaxed along any number of dimensions. Different types of heterogeneity will alter the model's results in different ways. The following specification explores how the results from the homogeneous model compare to the results from versions with different types of heterogeneity. The types of heterogeneity are: potential income (lognormal), deadweight cost (lognormal), response in approval rating due to lost income (lognormal), ex-regulation approval rating (normal), and the amount the agent demands in "persuasion" to increase support for the regulatory regime by one percentage point (normal). These five types are defined by:

```
dispersions = [
    {},
    {"potential_sig" : 1.0},
    {"deadweight_sig" : 1.0},
    {"response_sig" : 0.5},
    {"approval_sd" : 0.5},
    {"persuasion_sd" : 500}
]
```

The following specification introduces each type of heterogeneity in isolation of the others:

```
import model, license
model.headings (["n", "r"])
for n,d in enumerate (dispersions):
    for r in range (0, 110000, 10000):
        model.run (license.regime (received=r, sent=r*0.1,
            deadweight=2.5, response=1.5, **d), ids=[n,r])
```

Heterogeneity in the responsiveness of agents to changes in income and in ex-regulation approval ratings have little effect on the model's results. All else equal, more dispersion in income and in the cost of persuasion will increase approval of the regulatory regime.

The former effect is induced by the fact that more high-income agents implies a lower tax rate: income is bounded at the lower end, but not at the upper end. More dispersion in deadweight costs will decrease approval of the regulatory regime. See Figure 2.5.

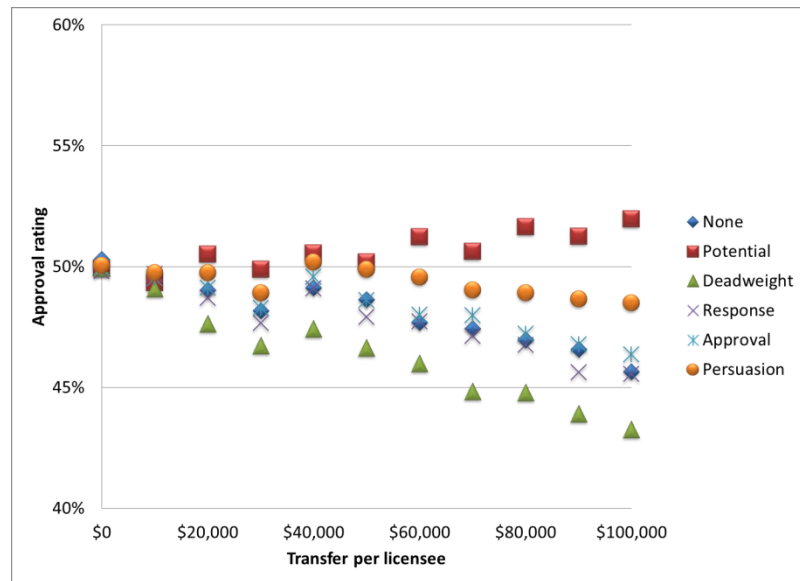


Figure 2.5: Approval rating by transfer per licensee and types of heterogeneity

Figure 2.5 depicts the independent effects of heterogeneity. It is also possible to explore the *cumulative* effects of heterogeneity. First, define the homogeneous characteristics:

```
homogeneous = {
  "deadweight" : 2.5,
  "response"   : 1.5
}
```

Next, augment those characteristics with individual variation:

```
heterogeneous = {
  "deadweight"      : 2.5,
  "response"        : 1.5,
  "potential_sig"   : 1.0,
  "deadweight_sig"  : 1.0,
  "response_sig"    : 0.5,
  "approval_sd"     : 0.5,
  "persuasion_sd"   : 500
}
```

Then exercise the model using both sets of characteristics:

```
import model, license
model.headings (["type", "r"])
for r in range (0, 110000, 10000):
    model.run (license.regime (received=r, sent=r*0.1,
        **homogeneous), ids=["homogeneous", r])
    model.run (license.regime (received=r, sent=r*0.1,
        **heterogeneous), ids=["heterogeneous", r])
```

As Figure 2.6 shows, the presence of agent heterogeneity significantly impacts the predictions of the model. As the amount of the transfer grows, heterogeneity makes the difference between approval and disapproval of the regulatory regime. Heterogeneity in deadweight costs and ex-regulation approval exert downward pressure; heterogeneity in responsiveness to losses in income and in persuasion exert upward pressure. The cumulative effect of agent heterogeneity is to increase support for the regulatory regime.

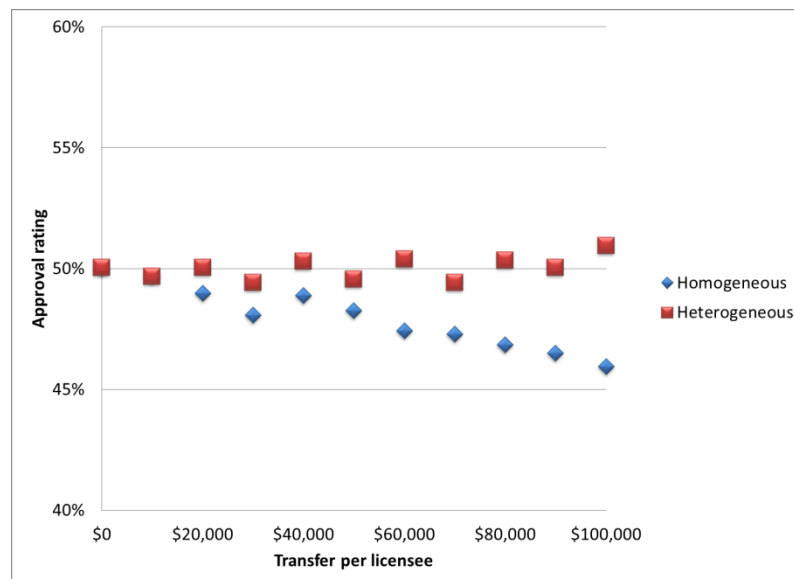


Figure 2.6: Approval rating by transfer per licensee and heterogeneity status

Figure 2.6 also reveals an interesting pattern: the degree to which agents approve of the regulatory regime appears to oscillate as the transfer amount grows. Is this oscillation a prediction of the model or a byproduct of its implementation? This question can be probed by increasing the model's resolution. The following code increases both the horizontal and vertical resolution by a factor of ten. The horizontal resolution is increased by sampling smaller transfer intervals. The vertical resolution is increased by running the model 100 times per transfer interval and calculating an average result.

```
import model, license
model.headings (["r"])
for r in range (0, 110000, 1000):
    model.run (license.regime (received=r, sent=r*0.1,
        **heterogeneous), ids=[r], runs=100)
```

The higher resolution specification reveals that the approval rating oscillates in a sawtooth pattern (see Figure 2.7). This pattern is a result of an assumption made by the model's default implementation: tax rates must be whole numbers. The existence of deadweight costs means that the tax rate applied to each group cannot be calculated directly; it must be determined by numerical methods. This means searching for the tax rate which will produce the desired revenue. Only whole numbers are searched. This illustrates both an advantage and a disadvantage of the agent-based approach. In the real world, tax rates are usually whole numbers, so imposing this restriction actually sheds light on the problem in a way that a continuous solution would not: a self-interested regulator would set the transfer amount at the "top end" of a saw tooth. On the other hand, the agent-based modeler is forced to follow this approach even when it does not

result in such serendipity. This illustrates why agent-based modeling should be *another* tool, rather than the *only* tool, in the modeler's toolkit.

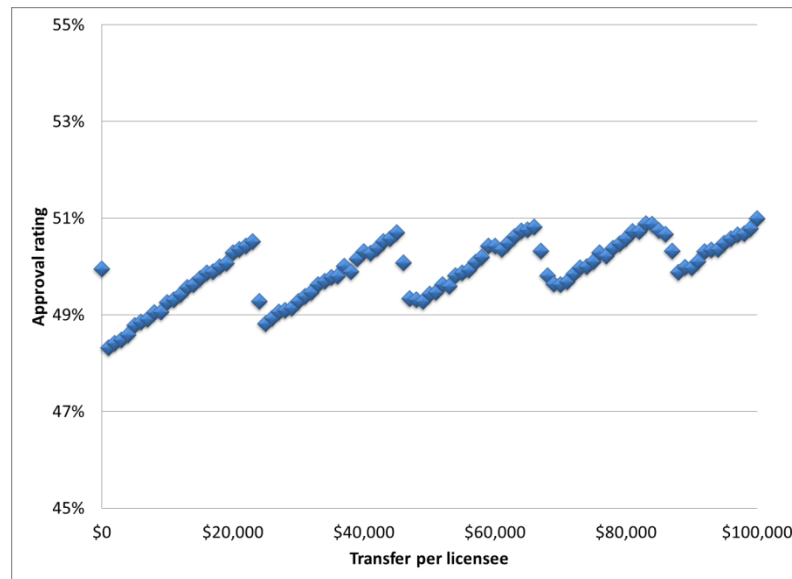


Figure 2.7: Approval rating by transfer per licensee: detail

Peltzman (1976) summarizes the problem facing regulators as follows: they “must pick the size (n) of the group they will benefit, the amount (K) they will ask that group to spend for mitigating opposition, and the amount (T) they will transfer to the beneficiary group.” The effect of introducing heterogeneity on the last of these three problems has now been explored. The effect on the middle problem—the amount spent to mitigate opposition—can be explored in the same way:

```

import model, license
model.headings (["type", "s"])
for s in range (0, 50000, 1000):
    model.run (license.regime (received=50000, sent=s,
        **homogeneous), ids=["homogeneous", s])
    model.run (license.regime (received=50000, sent=s,
        **heterogeneous), ids=["heterogeneous", s])

```

In this specification, the transfer is fixed at \$50,000. The model explores how regulator approval varies with the amount spent to mitigate opposition. Figure 2.8 shows the heterogeneous model’s prediction that approval levels off as persuasion funds approach 100% of the transfer; the homogeneous model predicts no such leveling. When some agents have higher persuasion thresholds than others, the rate at which the regulator benefits from “educating” these agents diminishes.

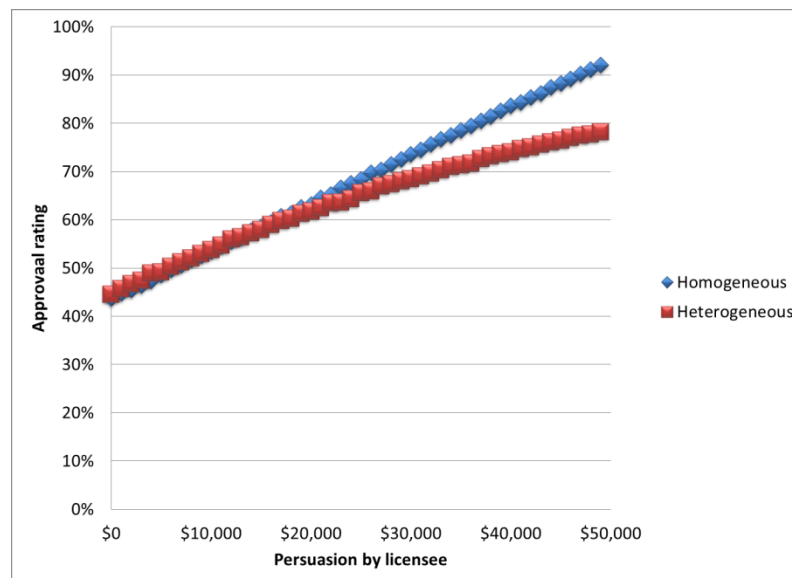


Figure 2.8: Approval rating by persuasion per licensee

Finally, the model can explore the effects of heterogeneity on the question Stigler originally posed. “The Stigler model leads ... to more than the near truism that [the ratio of the size of the subsidized group to the subsidizing group] is less than one; it more nearly asserts that the ratio is close to zero” (Peltzman 1976). Varying the size of the subsidized group is easily accomplished:

```
import model, license
model.headings (["type", "pct"])
for p in range (1, 50):
    pct = p / 1000
    model.run (license.regime (received=50000, sent=5000,
        percent=pct, **homogeneous), ids=["homogeneous", pct])
    model.run (license.regime (received=50000, sent=5000,
        percent=pct, **heterogeneous), ids=["heterogeneous", pct])
```

Figure 2.9 depicts the results. When agents are homogeneous, Stigler’s prediction matches the model: support falls as the number of licensees increases. Agent heterogeneity provides the regulator with greater latitude. One reason for this phenomenon is the application of a uniform tax rate: when some agents earn high incomes, the collective tax rate falls for a given level of revenue. Another is that at least some of these agents will manifest low deadweight costs; they will continue to work even in the presence of diminishing returns.

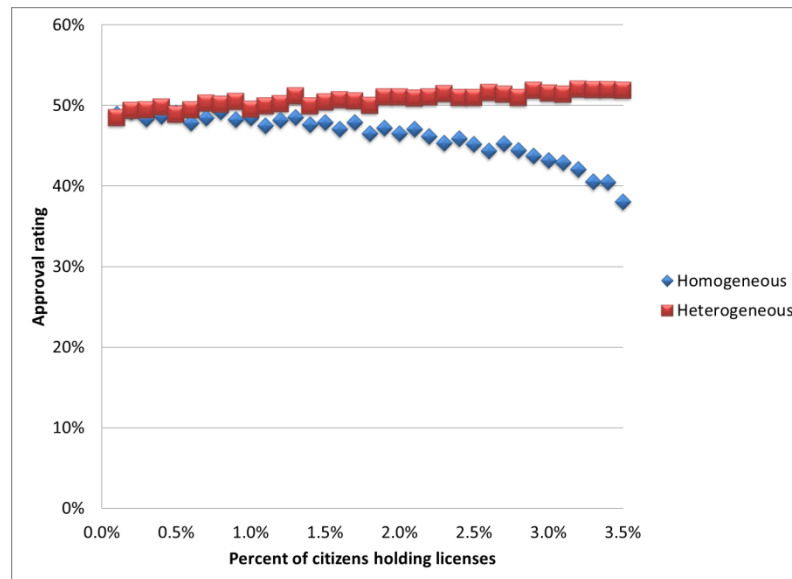


Figure 2.9: Approval rating by percent of citizens holding licenses

2.4. The cross-subsidy model

Figure 2.3 illustrates the effect of deadweight costs on the regulator’s approval rating: for a given transfer, approval is higher when deadweight costs are lower. Figure 2.4 shows how holding the *mean* deadweight cost constant and increasing its dispersal causes a decline in approval. If deadweight costs are distributed in a manner that is uncorrelated with other characteristics, there is little the regulator can do. But if different *types* of agents have different deadweight costs, the regulator has an incentive to partition the agent population—a process analogous to price discrimination. The more “docile” component of the population will subsidize not only the regulated entities, but also the more “hostile” component of the population. Peltzman (1989) enumerates many examples of this sort of cross-subsidization. A classic example is the U.S. telephony market prior to deregulation in the 1980s: AT&T held a monopoly on telephony

services, and used its long distance service to subsidize local service. In effect, long distance customers funded both AT&T's monopoly rents and low-cost local service.

Consider a situation in which there are two groups: a low deadweight cost group and a high deadweight cost group. Each agent is a member of exactly one of these groups. If the regulator can identify the high deadweight cost group and offer it a subsidy, the regulator's overall approval rating should increase. Approval from the low deadweight cost group will decline, but the decline will be outpaced by increases in approval from the high deadweight cost group. The following code tests this hypothesis by using the cross-subsidy implementation of the model. One percent of the population is directly subsidized. Another 10% of the population is cross-subsidized; this group has deadweight costs of 2.5. The remaining 89% of the population pays all subsidies and has deadweight costs of 1.0.

```
import model, subsidy
model.headings (["r"])
for r in range (0, 1600, 100):
    model.run (subsidy.regime (
        received=(50000, r), sent=(5000, r*0.05),
        deadweight=(1.0, 2.5)), ids=[r])
```

As Figure 2.10 shows, the regulator can increase its overall approval rating by partitioning the agents. As the cross-subsidy increases, the rise in approval from the subsidized group outpaces the fall in approval from the subsidizing group.

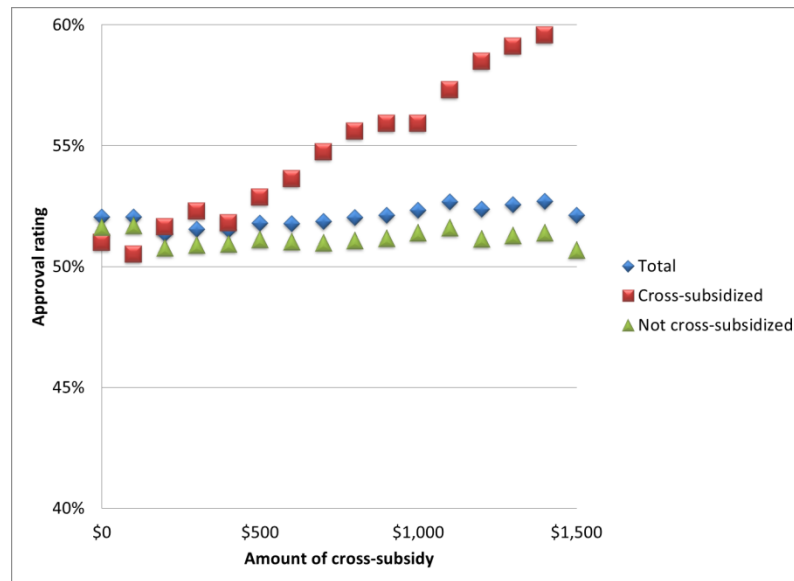


Figure 2.10: Approval rating by amount of cross-subsidy

Figure 2.10 assumes that agents are homogeneous in their responses to taxation: the more an agent's income declines below its potential, the faster the regulator's approval rating falls. The agent-based approach allows heterogeneous populations to be explored as well. The following code modifies its predecessor by varying the degree to which agents blame the regulator for shortfalls in income:

```
import model, subsidy
model.headings (["s", "r"])
for s in [0.0, 0.5, 1.0, 1.5]:
    for r in range (0, 1600, 100):
        model.run (subsidy.regime (
            received=(50000, r), sent=(5000, r*0.05),
            deadweight=(1.0, 2.5), response_sig=(s, s)), ids=[s, r])
```

The higher the degree of heterogeneity, the less likely that a given cross-subsidy will increase the regulator's overall approval rating. For sufficiently high levels of heterogeneity, the regime will cease to be viable, even with cross-subsidies. As Figure

2.11 shows, a cross-subsidy of \$200 is sufficient when agents are homogeneous. But when agents are highly heterogeneous, the minimum cross-subsidy rises to \$1100.

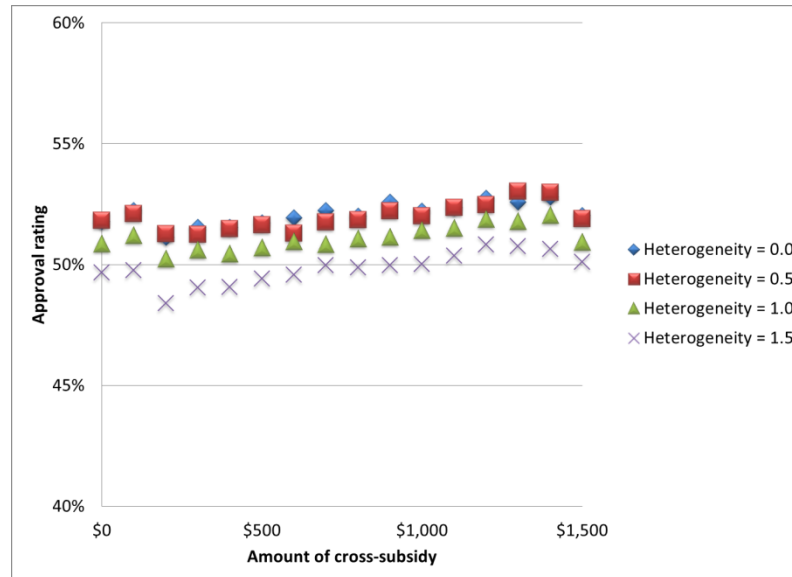


Figure 2.11: Approval rating by amount of cross-subsidy and degree of heterogeneity

Even when agents are homogeneous, the regulator must exercise care in choosing the recipients of cross-subsidies. Cross-subsidies which are poorly targeted may backfire. Consider a regulatory regime that includes a lump-sum transfer to a cross-subsidized group. The following specification traces the behavior of the cross-subsidized group as its potential income rises to match that of the general public:

```
import model, subsidy
model.headings (["p"])
for p in range (1000, 30000, 1000):
    model.run (subsidy.regime (
        received=(50000, 5000), sent=(5000, 500),
        potential=(30000, p), deadweight=(3.0, 1.0)), ids=[p])
```

Figure 2.12 reveals that as the cross-subsidized group's potential income increases, the lump-sum subsidy grows less and less important, and approval of the regulator declines. If the regulator is unable to restrict cross-subsidies to low-income agents only, the cross-subsidy will be counterproductive; the regulator should search for an alternative partitioning. This is another way of expressing Becker's (1983) argument that regulators are incentivized to identify the least inefficient regulatory regimes.

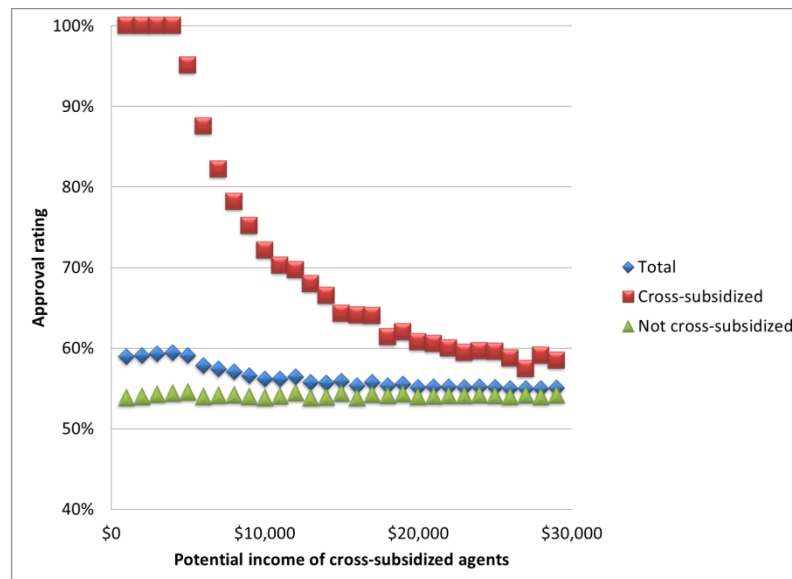


Figure 2.12: Approval rating by potential income of cross-subsidized agents

2.5. The bidirectional model

A consistent theme in the public choice literature is that “most government income redistribution goes not to the poor but to people who for one reason or another have sufficient political influence to get it” (Tullock 2005, 171). In practice, redistribution policy is shaped by “alliances between well-intentioned people who are

attempting to do good by deceiving the average voter, and the people who are not particularly well-intentioned and who are attempting to benefit by the same deception” (ibid., 158). The redistribution regime thus ends up composed mainly of transfers both *from* and *to* the middle class. Mueller (2003, 516) notes that the “most salient feature” of modern government redistribution is its “lack of a uni-directional flow.” Parents send their children to public schools and subsidize retirees via payroll taxes; at the same time, seniors enjoy retirement and fund schools via property taxes.

Peltzman’s 1976 model does not directly support the notion of bidirectional transfers, but its agent-based implementation is easily extended to this case. The following specification divides the population into two groups. The first group has 60% of the population; the second group has 40%. The citizens in each group are identical, differing only in the tax rates that they must pay. As before, tax rates are determined by net outflows from the group.

```
import model, bidirectional
model.headings (["type", "r"])
for r in range (0, 1100, 100):
    model.run (bidirectional.regime (
        received=(r, 0), sent=(r*0.1, 0)), ids=["first", r])
    model.run (bidirectional.regime (
        received=(0, r), sent=(0, r*0.1)), ids=["second", r])
    model.run (bidirectional.regime (
        received=(r, r), sent=(r*0.1, r*0.1)), ids=["both", r])
```

The model produces three series. In the first two series, only one group receives transfers. In the third series, both groups receive transfers. As Figure 2.13 reveals, support for the regulatory regime is higher when transfers are directed to both parties instead of only one. Furthermore, both parties prefer bilateral transfers to no transfers at

all. This is a product of what Peltzman calls “education.” In this model, senior citizens feel an obligation to fund schools, and parents feel an obligation to support senior citizens. Mutual dependence is preferred over autarky.

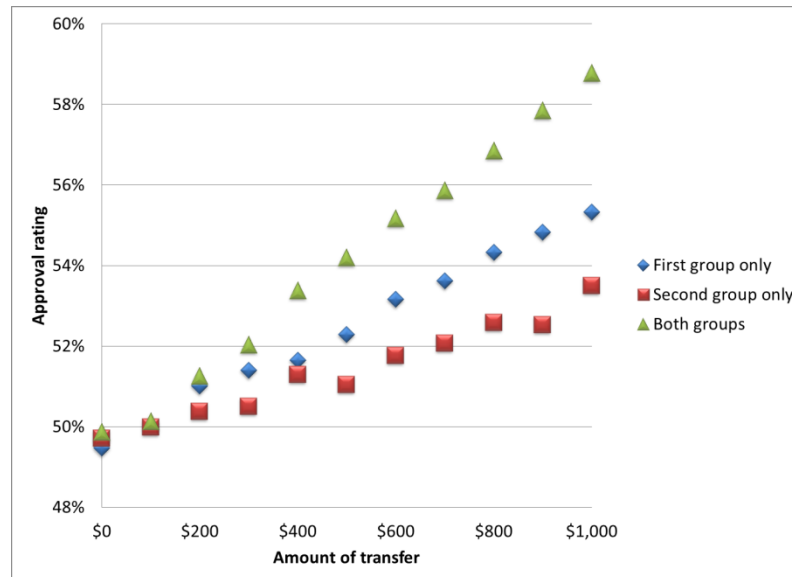


Figure 2.13: Approval rating by amount of transfer and direction: homogeneous groups

These results accord with the findings in Peltzman’s subsequent study of the growth of government, in which he notes the “counterintuitive result that, on balance, more equality breeds a political demand for still more income equalization” (Peltzman 1980). Thus, “homogeneous interests become [an] important source of government growth” (ibid.). The bidirectional model allows this thesis to be tested easily, by introducing heterogeneous agents. The only difference between the following specification and its predecessor is that the second group now resents having to fund transfers to the first group.


```

import model, bidirectional
model.headings (["type", "r"])
for r in range (0, 1100, 100):
    model.run (bidirectional.regime (
        received=(r, 0), sent=(r*0.1, 0),
        deadweight=(1.0, 3.0)), ids=["first", r])
    model.run (bidirectional.regime (
        received=(0, r), sent=(0, r*0.1),
        deadweight=(1.0, 3.0)), ids=["second", r])
    model.run (bidirectional.regime (
        received=(r, r), sent=(r*0.1, r*0.1),
        deadweight=(1.0, 3.0)), ids=["both", r])

```

Figure 2.14 reveals that the regulator will not receive support for a regime which benefits only the first group. A bilateral regime is viable, but even more viable is a regime in which the first group subsidizes the second. This supports Peltzman's thesis that "homogeneous interests" are positively correlated with the size of government.

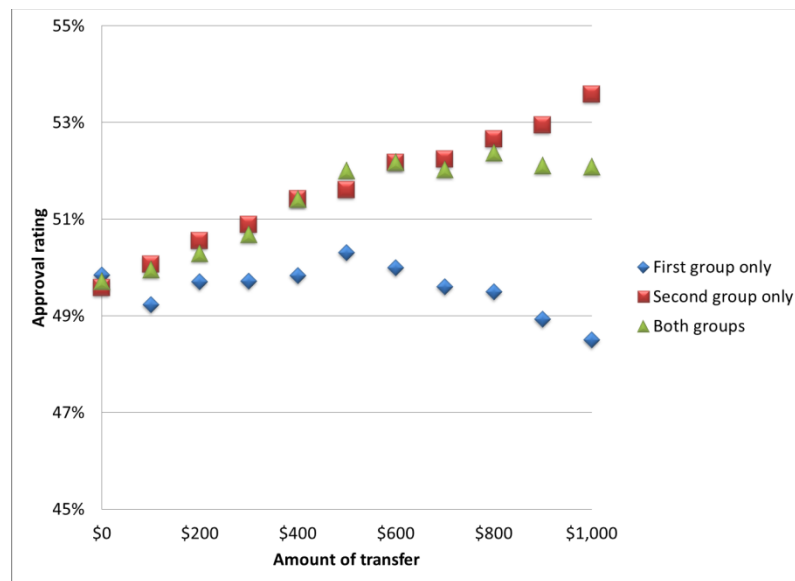


Figure 2.14: Approval rating by amount of transfer and direction: heterogeneous groups

However, Peltzman also argues that the size of government tends to rise as incomes equalize. The agent-based model can test this proposition. The following specification creates two groups. In the first group, individuals have incomes of \$50,000; in the second group, individuals have incomes of \$100,000. The percent of agents in the first group is traced from 1% to 99%.

```
import model, bidirectional
model.headings (["pct"])
for p in range (1, 100):
    pct = p / 100
    model.run (bidirectional.regime (percent=pct,
        received=(1000, 1000), sent=(50, 100),
        potential=(50000, 100000)), ids=[pct])
```

The agent-based version of Peltzman (1976) predicts the opposite of the conclusion presented in Peltzman (1980): as heterogeneity increases, support for the regulatory regime *increases*. Figure 2.15 shows the shift from homogeneous (1%) to heterogeneous (50%) and back to homogeneous (99%). The shape of the transition is unmistakably concave.

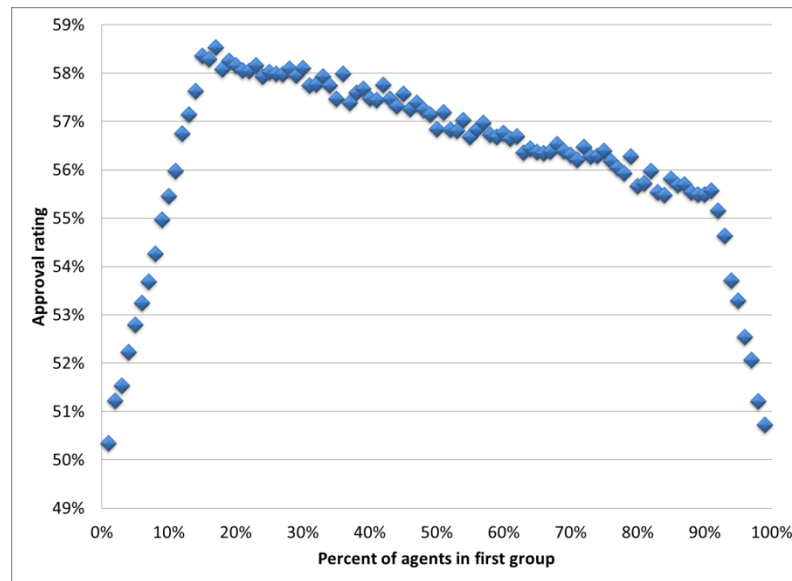


Figure 2.15: Approval rating by income heterogeneity

The verdict on the effects of heterogeneity on regulatory regimes is therefore mixed. Homogeneous interests tend to promote regulation, while homogeneous incomes tend to discourage it.

2.6. Conclusion

The study of regulatory capture dates back at least to the time of Adam Smith. In The Wealth of Nations, Smith develops a view of regulation very similar to that demonstrated by public choice scholars today: regulatory regimes do not always protect those whom they purport to protect. Instead, regulation is a vehicle for self-advancement by commercial interests; and citizens, placated with “clamor and sophistry,” generally do not object. But despite Smith’s clear reasoning, this view was largely ignored for two centuries. Not until Stigler (1971) reformulated Smith’s theory did the concept of

regulatory capture become widely accepted. In the intervening years, the dominant theory of regulation was the normative-as-positive theory: regulation existed to protect the weak from the strong.

Stigler (1971) wholeheartedly rejects the protect-the-weak narrative. Instead, Stigler argues provocatively that regulation exists entirely for the benefit of business. Posner (1974) and Peltzman (1976) express appreciation for Stigler's approach, but object to his characterization of regulation as entirely one-sided. If regulation is entirely in aid of business, how can regulation pass the ballot box test, as it most surely does? Peltzman (1976) refines and formalizes Stigler's vision. In Peltzman's model, regulators serve two masters: business and citizens. The regulator chooses the regime which maximizes total support across both groups. To engineer such a result, the regulator will attempt to divide each group into smaller groups and purchase support from those groups separately. Thus regulatory regimes will often be accompanied by extensive networks of cross-subsidies. Furthermore, regulatory regimes will tend to appear at the extreme ends of the spectrum of competition: both monopolies and perfectly competitive industries are likely to be regulated. Thirty-five years after its formulation, Peltzman's model is still the starting point for any discussion of regulation.

Like any rational choice model, Peltzman (1976) faces a tradeoff between tractability and complexity: as complexity increases, tractability decreases. Thus Peltzman's model handles heterogeneity in only a very limited way. Intergroup heterogeneity is permitted to motivate cross-subsidies, but intragroup heterogeneity is

disallowed. Recasting Peltzman (1976) as an agent-based model allows exploration of a full range of agent heterogeneity.

Adding intragroup heterogeneity to the model reveals that such heterogeneity can either promote or inhibit regulation. When all agents share a common tax rate, income heterogeneity can increase support for regulation: high-income agents subsidize low-income agents. But when agents splinter into factions, support for regulation declines. Variability in the amount of “persuasion” required for an agent to support a transfer tends to promote regulation. Deadweight costs are the biggest threats to regulation: even a modest regulatory regime may prove unviable if agents are highly reactive to tax rates. This outcome is consistent with the prediction of Becker (1983): regimes which minimize deadweight costs will triumph. Peltzman emphasized the necessity of cross-subsidies, and the agent-based model confirms Peltzman’s view. Partitioning citizens according to their “elasticities of resistance” can make the difference between a successful and unsuccessful regime.

3. AN AGENT-BASED MODEL OF THE CONDORCET JURY THEOREM

3.1. Introduction

When Marie-Jean-Antoine-Nicolas Caritat, Marquis de Condorcet, was born into French nobility in 1743, there was no reason to believe that he would be remembered, two and a half centuries later, as a pioneer of the application of science to the social condition. He came from a long line of military men, and might have followed his father into that profession, had the latter not died in the line of duty, a mere month after his son was born (Baker 1975, 2-3). Instead the young Condorcet was raised by Jesuits, and became a man of letters. By the age of 21 he was considered among the top 10 mathematicians in Europe; by 30 he “had established himself as a mathematician, academician, philosophe, and pamphleteer” at the Academy of Sciences (ibid., 7, 55).

Working at the height of the Enlightenment, Condorcet shared the optimism of his era. He “posited a relationship between scientific advance and social welfare,” and noted that “in all countries where the physical sciences have been cultivated, barbarism in the moral sciences has ... dissipated and ... error and prejudice have disappeared.” He “insisted that scientific progress necessarily entailed the rationalization of the whole social order” (ibid., 75). It was during this period—the years 1775 to 1785—that Condorcet attempted to “reconcile the satisfaction of his mathematical abilities with the

imperatives of his passion for the public good” (ibid., 82). Thus was born, as he would come to call it, *social mathematics* (ibid., 332).

Condorcet regarded his 1785 “Essai sur l’application de l’analyse à la probabilité des décisions rendues à la pluralité des voix” as his “most fundamental contribution to social mathematics” (ibid., 183). His interest in the subject of what is now called public choice was sparked by his association with his friend and fellow scientist Ann-Robert-Jacques Turgot. Turgot had the unfortunate task in 1774 of informing the newly crowned Louis XVI that “he had inherited a realm without a constitution... a constant war between the king and his people” (ibid., 206). It was clear that power could no longer reside with the King alone; some sort of collective decision-making process was needed, or chaos would ensue. And so Condorcet wrote his *Essai* to answer a simple question: “Under what conditions will the probability that the majority decision of an assembly or tribunal is true be high enough to justify the obligation of the rest of society to accept that decision?” (ibid., 228). Condorcet believed that his *Essai* presented a “mathematical guarantee” that “assures the validity of a law passed by the smallest possible majority, such that one can believe that it is not unjust to subject others to this law” (ibid., 230). In other words, Condorcet claimed to have proven that majority rule was a morally defensible scheme for making decisions.

Although Condorcet’s jury theorem, as it came to be known, is now recognized as “far superior to anything that had gone before,” its publication in the *Essai* failed to impress. As Duncan Black explains, the jury theorem was described by Condorcet’s contemporaries as “impracticab[le]” and “fantastic,” and it was dismissed as having “too

little value to detain us longer” (Black 1958, 160-162). But none of them, Black concludes, “had really understood it” (ibid.). The genius of Condorcet’s theorem, according to Black, is not the theorem itself, but rather that Condorcet had established “a system of formal reasoning which is quite independent of the theory of probability” (ibid., 163-164).

Condorcet’s theorem can be summarized as follows. Suppose that a population of N voters must cast a single vote on a single issue, and suppose that each vote may be considered either “correct” or “incorrect.” For example, the voters might comprise a jury, with a “correct” vote being to acquit if innocent or convict if guilty, and an “incorrect” vote being to acquit if guilty or convict if innocent. The theorem states that if all voters have a common probability P of voting correctly, and P exceeds 0.5, then the probability of a correct vote approaches 1 as N approaches infinity. Consider a numerical example: if a single voter votes correctly with probability 0.6, that voter’s judgments will be correct 60% of the time. If a three-voter, majority-rule jury is convened, and each voter votes correctly with probability 0.6, the jury’s judgments will be correct 64.8% of the time². As the number of jurors increases, the probability of a correct vote also increases. In the limit, as N approaches infinity, the probability of a correct vote is 1.

Condorcet himself did not actually provide a formal proof of his theorem; he provided a much longer version of the above example (Condorcet 1995 [1785], 33), and then asserted (but did not prove) that the probability of a correct vote approached 1 in the

² There are eight possible voting combinations (C=correct, I=incorrect): CCC, CCI, CIC, CII, ICC, ICI, IIC, III. The probability of CCC is $0.6 * 0.6 * 0.6 = 0.216$; the probability of CCI, CIC, or ICC is $0.6 * 0.6 * 0.4 * 3 = 0.432$. So the probability of a correct vote is $0.216 + 0.432 = 0.648$.

limit. Mueller (2003, 129) provides a simple mathematical exposition of the theorem. Assuming N voters (where N is odd), the probability of a correct vote is the probability that $(N+1)/2$ voters vote correctly, plus the probability that $(N+1)/2+1$ voters vote correctly, and so on, up to all N voters voting correctly:

$$P_N = \sum_{h=\frac{N+1}{2}}^N P(\text{exactly } h \text{ correct votes})$$

The probability of h identical voters voting correctly is $P^h (1-P)^{N-h} C(N,h)$. That is: h voters vote correctly (P^h); $N-h$ voters vote incorrectly ($(1-P)^{N-h}$); and there are $C(N,h)$ different combinations of the h voters³. The probability of a correct vote is thus:

$$P_N = \sum_{h=\frac{N+1}{2}}^N P^h (1-P)^{N-h} C(N, h)$$

As Young (1997, 183) demonstrates, as N approaches infinity, P_N approaches 1 if $P > 0.5$, 0 if $P < 0.5$, and 0.5 if $P = 0.5$.

The result of the jury theorem is reassuring to those who harbor concerns about the fallability of democracy. The theorem means that if the average voter is even the slightest bit more likely, on average, to be right than wrong, then majority rule will yield the correct outcomes. Rousseau, for example, took comfort in this result, which paralleled his own informal reasoning in *Du contrat social*: “Rousseau believed that the pluses and minuses of individual interests would cancel out,” and the general will would

³ In the previous example, if exactly two voters vote correctly, there are $C(3,2) = 3!/(2!1!) = 3$ different combinations which yield a correct vote: CCI, CIC, and ICC.

emerge (Baker 1975, 231). The theme continues to resonate today; James Surowiecki's (2004) book "The Wisdom of Crowds" covered similar ground, and was a bestseller.

Mueller (2003, 130) notes that Condorcet's theorem, when expressed in the above form, implies three assumptions. First, the same probability P applies to all voters: voters are assumed to be homogeneous. Second, voter interaction does not matter: voters are assumed to be independent. Third, no voter votes strategically: voters are assumed to vote sincerely. As Mueller notes, each of these assumptions may be questioned. Some voters are more likely to vote correctly than others; some voters can influence other voters; and some voters might benefit from voting insincerely (ibid., 131).

Condorcet's theorem is, in the words of List and Gooden (2001), the "jewel in the crown" of the argument that "democracy [is] the best ... procedure available" for collective decision-making. The assumptions that Mueller enumerates are unwelcome caveats to that argument: democracy is the best procedure available, if voters are identical (though they are not), independent (though they are not), and vote sincerely (though they may not). Following this strain of thought, the defense of democracy lies in the degree to which Condorcet's theorem holds when these assumptions are relaxed.

Relaxing Condorcet's assumptions is a challenge which has been accepted by numerous researchers. Boland (1989) describes successful attempts to relax the first two assumptions. Condorcet's theorem is easily extended to heterogeneous agents whose *average* probability of correctness exceeds 0.5 (Hoeffding 1956), and even to circumstances in which a minority of sufficiently well-informed voters can overrule a majority of less-informed voters (Miller 1986). The theorem can also be extended to

situations in which the first voter influences subsequent voters: convergence to a correct result occurs more quickly when voters play “follow the leader” (Boland, Proschan and Tong 1989). Koriyama and Szentes (2007) describe successful attempts to relax the third assumption. When voters take into account not only their own “private” signal but also a cumulative “public” signal inferred from the behavior of other voters, convergence to the correct result can occur even in the presence of strategic voting (Feddersen and Pesendorfer 1997). Myerson (1998) extends this result by illustrating that strategic voters may not even need to know how many other voters have contributed to this cumulative public signal.

However, there are also situations in which relaxing these assumptions undermines Condorcet’s theorem. Franz (2008) considers the question of whether homogeneity and independence can be relaxed simultaneously, and concludes that while it is possible to relax one or the other, relaxing both is problematic. Austen-Smith and Banks (1996) develop a model in which strategic voting is demonstrated to be inconsistent with Nash equilibrium, which implies that a “rational choice foundation for the claim that majorities inevitably ‘do better’ than individuals ... has yet to be derived.” Koriyama and Szentes (2007) develop their own model in which strategic voting, in the presence of nonzero information costs, imposes an upper bound on optimal committee size, contradicting the results of Condorcet’s theorem.

Previous attempts to relax the assumptions that underlie Condorcet’s theorem have something quite striking in common: none of them presents a model in which all three assumptions are relaxed simultaneously. Riechmann (2001) suggests an

explanation: as rational choice models grow in complexity, the difficulty of making these models tractable grows even faster. One cannot simply combine three separate models to produce a single model that relaxes all three assumptions. However, computational techniques, such as agent-based modeling, offer an alternative approach. Agent-based modeling techniques can be used to construct a model which allows a fuller exploration of Condorcet's theorem, in which all three assumptions can be relaxed. Does the Condorcet Jury Theorem still produce democracy-validating outcomes in the presence of heterogeneous, adaptive, rational voters? An agent-based model can help answer this question. This chapter presents such a model.

3.2. The model

The model is written in Python. It contains three conceptual entities: the core model itself, six supporting interfaces, and various specifications used to run the model. The core model accepts as arguments a *specification* and a *policy*. The specification instructs the model:

- how to construct agents;
- how to compute each agent's likelihood of correctly evaluating a proposition;
- how to assign agents to social networks;
- the degree to which an agent is influenced by its social network;
- the type of effect an agent's social network has on the agent's behavior; and
- the voting strategy employed by each agent.

The model begins by instantiating the agents using the specification's *agent factory*. Each agent is imbued with certain characteristics which may vary with the specification. If a policy is supplied, this policy is applied to each agent, potentially altering these characteristics. Each agent is assigned to a social network, with the assignment potentially determined by the agent's characteristics. A proposition is presented to the agents. Each agent evaluates the proposition, using both its own assessment and the assessments of other agents in its social network. A vote is then taken. Voters may vote sincerely or they may vote strategically. The model tabulates the votes. If the percent of correct votes exceeds the threshold required for passage, the vote passes; otherwise it fails. The model repeats this process for multiple runs and calculates the percent of runs in which a passing vote obtains. This value is the output of the model and is returned to the caller. The model appears in source file `model.py` (Appendix 3.A.1).

Each of the six values in the specification is defined via its own interface. Each interface is defined by a function which takes a prescribed set of arguments (a *signature*) and returns a value of a prescribed type. The function itself is provided in the specification. If no function is provided, the interface's "null" function is used instead. The null function implements the interface in an intentionally trivial manner. For example, the null agent factory creates agents that have no characteristics at all. (This may sound uninteresting, but recall that in Condorcet's original formulation, agents have no independent characteristics. As will be shown later, the null functions are used to implement most of the Condorcet specification.)

The *factories* interface consists of a function which takes an integer N and returns a list of N agents. Each agent is represented by a Python dictionary, which maps names to values. For example, an agent might be represented by the dictionary {"MALE": 1, EDUC": 0.8}, meaning that the agent is a male with 16 years of education (0.8 means 80% of the maximum of 20 years, which is 16). This dictionary defines the agent's *characteristics*. The null() function creates agents with no characteristics at all. The characteristic() function accepts an argument containing a list of characteristics, their distributions, and their correlations, and instantiates agents with characteristics drawn from those distributions and with those correlations. This mechanism allows the modeler to calibrate the model using real-world distributions. The agent factory interface appears in source file factories.py (Appendix 3.A.2).

The *likelihoods* interface consists of a function which takes a list of N agents and returns a parallel list of N likelihoods of correctness. The null() function assigns each agent a likelihood of correctness of 0.5. The constant() function assigns each agent a specified constant likelihood. The distribution() function assigns each agent a likelihood drawn from the supplied distribution. The characteristic() function assigns each agent the likelihood returned by the supplied function. The likelihood interface appears in source file likelihoods.py (Appendix 3.A.3).

The *confidences* interface is nearly identical to the likelihoods interface: it consists of a function which takes a list of N agents and returns a parallel list of N degrees of confidence, one for each agent. Confidence is the level of confidence that the agent has in its own likelihood of correctness. The null() function assigns each agent a

confidence of 1, meaning the agent is completely confident (and thus unaffected by its social network). The `constant()` function assigns each agent a specified constant confidence. The `distribution()` function assigns each agent a confidence drawn from the supplied distribution. The `characteristic()` function assigns each agent the confidence returned by the supplied function. The confidence interface appears in source file `confidences.py` (Appendix 3.A.4).

The *networks* interface consists of a function which takes a list of N agents and returns a parallel list of N network assignments. The `null()` function assigns each agent to its own unique social network (meaning, effectively, no social network). The `grid()` function assigns agents to networks geographically, so that each agent has “neighbors.” The number of dimensions, the distributions along each dimension, and the total size of each dimension may all be specified. The `uniform()` function divides agents into networks arbitrarily and uniformly. The `partition()` function assigns agents to social networks based on agent characteristics; the agent population is partitioned into an arbitrary number of networks of any combination of characteristics. The network interface appears in source file `networks.py` (Appendix 3.A.5).

The *effects* interface consists of a function which takes a list of N triples of the form (network, likelihood, confidence) and returns a parallel list containing N revised likelihoods. The `null()` function returns each agent’s original likelihood unchanged. The `mean_reversion()` function pulls an agent’s likelihood toward the average of its social network; the `lift_up()` function pushes the likelihood toward the top of the agent’s social network; and the `pull_down()` function pulls the likelihood toward the bottom of the

agent's social network. The `herd()` function moves the likelihood toward the majority view. The effects source code appears in `effects.py` (Appendix 3.A.6).

Finally, the *strategies* interface consists of a function which takes a list of N pairs of the form (likelihood, confidence), one pair for each agent, and returns the percent of agents who vote correctly. The `null()` function implements a sincere vote: each agent votes according to its likelihood. The `predominant()` function implements a peer-effects strategy: agents vote sequentially, and each agent may consider both its own likelihood (“private information”) and the voting pattern of other agents (“public information”). The `pivotal()` function implements the peer-effects strategy only when the last voter is the pivotal voter. The `exhibitional()` function implements exhibitional voting: each agent votes may consider both its own likelihood and the preferences of an audience (e.g., in the case of an elected official, constituents). The strategies source code appears in `strategies.py` (Appendix 3.A.7).

These six interfaces allow the model to relax each of the assumptions implicit in Condorcet's formulation of his theorem. The factories and likelihoods interfaces allow the homogeneity assumption to be relaxed. The networks, confidences, and effects interfaces allow the independence assumption to be relaxed. The strategies interface allows the sincerity assumption to be relaxed. In each case, the interface provides multiple approaches for relaxing the assumptions. If none of these approaches is sufficient for a desired model specification, new approaches can be added. The ease by which new approaches can be created is one of the strengths of the agent-based approach.

3.3. Demonstrating the model

The preceding section provided an overview of the model. Such an overview is essential for understanding the model; yet the overview, on its own, fails to communicate how the model is actually used. This section demonstrates the model by replicating Condorcet's original results, and prepares the way for further exploration of his famous theorem.

As stated earlier, the behavior of the model is dictated by its specification. The simplest possible specification of the model is the null specification. In this intentionally trivial specification, all agents vote correctly with probability 0.5, all agents are independent, and all agents vote sincerely. The null specification can be invoked by calling the model's `percent()` function without providing a specification argument. The following Python code runs the model using the null specification:

```
import model
print ("voters pct_correct")
for voters in range (1, 200, 2):
    print (voters, model.percent (count=voters))
```

The model is invoked 100 times, starting with one voter and ending with 199 voters (with an odd number of voters each time). As Figure 3.1 shows, the percent of model invocations that result in a correct vote hovers around 50, regardless of the number of voters.

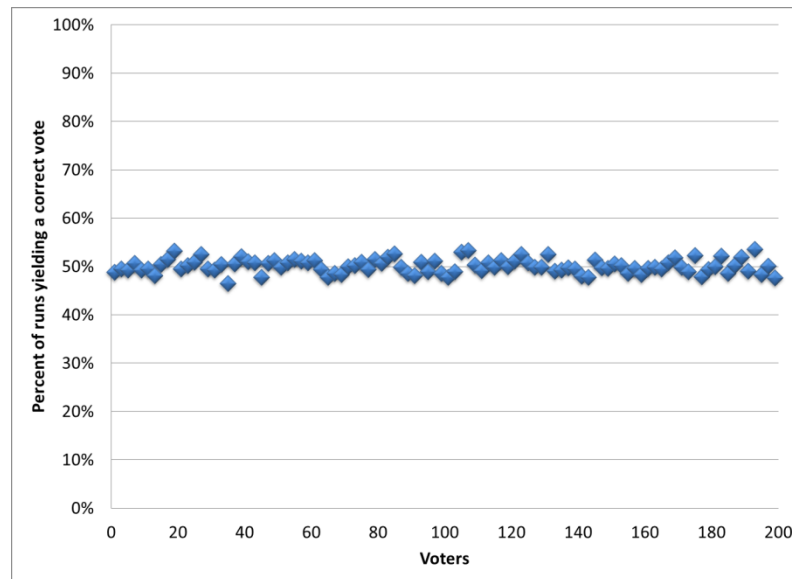


Figure 3.1: Likelihood of a correct vote by number of voters

The results in Figure 3.1 accord with intuition: if every voter flips a coin, a majority vote will be correct roughly half the time. Condorcet’s theorem asserts that if every voter can do a little better than flipping a coin—can vote correctly with $P > 0.5$ —then a large enough electorate will yield a correct majority vote with certainty. The Condorcet specification explores this assertion.

The Condorcet specification expresses the theorem as originally outlined by Condorcet. In Condorcet’s formulation, each agent votes correctly with a fixed probability. In terms of the likelihood interface, this means using the `constant()` function instead of the `null()` function. The Condorcet specification, therefore, consists of a single value: a constant likelihood function. (The Condorcet specification appears in `condorcet.py`; see Appendix 3.B.1.) The following code defines the entire Condorcet specification:

```

import likelihoods
def spec (likelihood):
    return {
        "likelihood": lambda a: likelihoods.constant (a, likelihood)
    }

```

The `spec()` function returns the Condorcet specification, which contains only one element: a likelihood function. The likelihood function returns the same likelihood for each agent; that likelihood is provided when the specification is requested. Thus, a Condorcet specification for a voter that votes correctly with $P = 0.6$ would be created as follows:

```
condorcet.spec (0.6)
```

The following code invokes the model using the Condorcet specification with probabilities between 0 and 1, in increments of 0.1. The electorate ranges from 1 to 199 voters, with the number of voters always odd:

```

import model, condorcet
print ("likelihood count pct_correct")
for likelihood in range (0, 11):
    for voters in range (1, 200, 2):
        print (likelihood / 10, voters, model.percent (
            spec=condorcet.spec (likelihood / 10), count=voters)

```

Figure 3.2 shows the results of running the Condorcet specification. When each voter's likelihood of correctness is less than 0.5, the percent of correct votes approaches zero. The closer the likelihood is to zero, the faster the percent of correct votes approaches zero. Conversely, when each voter's likelihood of correctness is greater than 0.5, the percent of correct votes approaches 100. When the likelihood of correctness is exactly

0.5, the results are identical to the null specification: the percent of correct votes hovers around 50. Figure 3.2 expresses the essence of Condorcet's theorem.

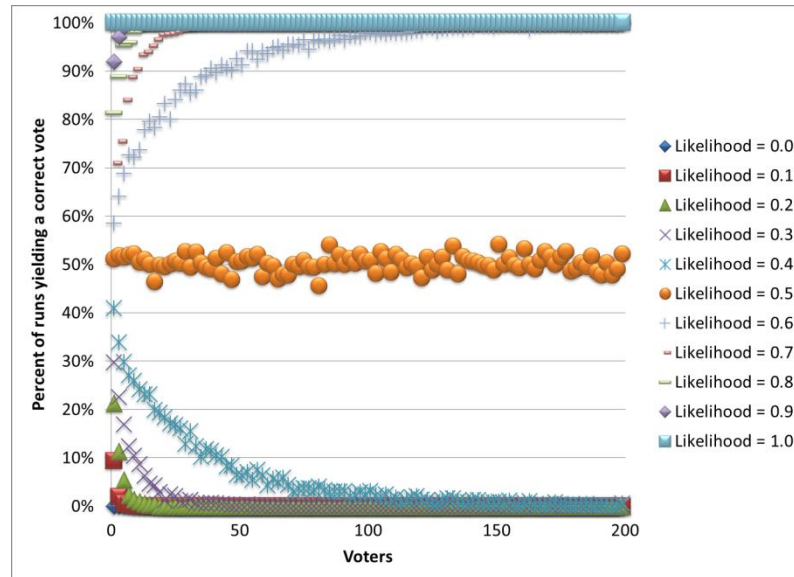


Figure 3.2: Likelihood of a correct vote by number of voters: detail

Of special interest is the case when the number of voters is 1. A single voter with likelihood of correctness P will vote correctly $P * 100\%$ of the time. But consider the fact that each *individual* vote is binary—that is, each vote must be either 0% correct or 100% correct. Figure 3.2 thus implies that the model obtains its results by holding multiple votes and returning the average over all iterations. This is indeed the case, and is a common approach in agent-based modeling. By holding multiple votes, the model is able to discern the overall voting *trend*. A voter who votes correctly with $P = 0.6$ will never cast a (single) vote that is 60% correct; every vote is either 0% correct or 100%

correct. However, when casting a large number of successive votes, such a voter will cast a correct vote about 60% of the time.

Consider the situation in which there is a single voter who votes correctly with likelihood P , and the model holds R votes, returning the average correctness over those R votes. If $R = 1$, the model will return either 0% (0/1) or 100% (1/1). If $R = 2$, the model will return 0% (0/2), 50% (1/2), or 100% (2/2). As R tends to infinity, the model will tend to return $P * 100\%$. The relationship between R and the average value returned by the model can be explored by this invocation:

```
import model, condorcet
print ("runs iteration pct_correct")
for r in range (1, 1000):
    for i in range (100):
        print (r, i, model.percent (
            spec=condorcet.spec (0.6), count=1, runs=r))
```

In this code, a single voter votes correctly with likelihood 0.6. The number of runs varies from 1 to 999. For each of these values, the model is invoked 100 times. Figure 3.3 depicts the results. When $R = 1$, the model always returns either 0% or 100%. When $R = 2$ the model always returns 0%, 50%, or 100%. When $R = 3$ the model always returns 0%, 33%, 67%, or 100%. As R approaches 1000, the values returned by the model converge toward 60%.

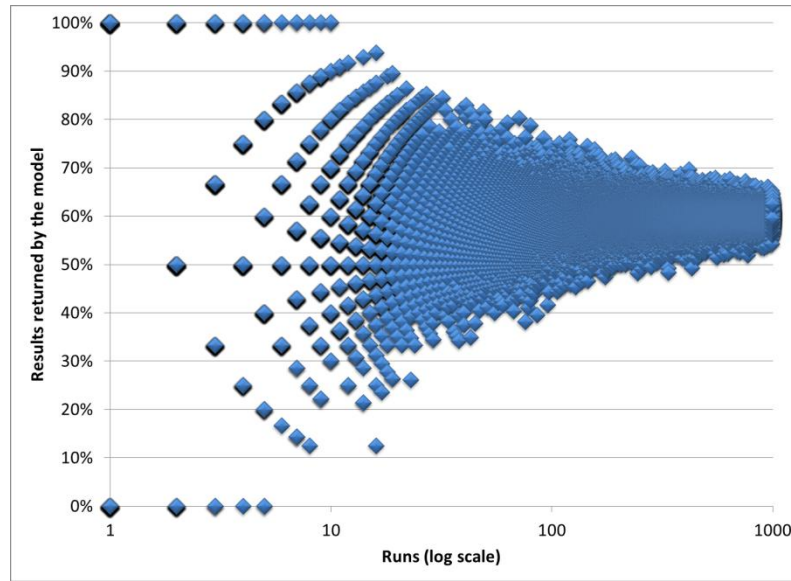


Figure 3.3: Values returned by the model by number of runs

Figure 3.3 demonstrates that in the single-voter case, the model’s meaningfulness rises with R . When the model is run with a value of R that is “too small,” the noise induced by the binary nature of voting overwhelms the voter’s likelihood of being correct, and the results are meaningless. But how small is “too small”? This question can be rephrased more precisely as follows: if the model is invoked X times with R runs per invocation, and thus produces X results, what percent of those X results are within some range ϵ of the average result? For example, if the single-voter model is invoked 100 times with $R = 1$ and $P = 0.6$, it produces 100 results, all of which are either 0% or 100% (each vote is either incorrect or correct). Given $P = 0.6$, we would expect 40 of these invocations to return 0% and 60 to return 100%, for an average return value of 60%. Given $\epsilon = 10\%$, we would say that the model exhibits zero convergence: *none* of the 100 invocations returned a value within 10 percentage points of the average value of

60%. (They all returned either 0% or 100%, and neither value is within 10 percentage points of 60%.)

The model provides a function, `convergence()`, which allows convergence to be explored easily. The following code invokes the single-voter model 100 times for values of R from 1 to 999, with $P = 0.6$, and reports the percent of invocations that are within 10%, 5%, and 1% of the average result:

```
import model, condorcet
print ("epsilon runs convergence")
for e in [0.10, 0.05, 0.01]:
    for r in range (1, 1000):
        print (e, r, model.convergence (
            spec=condorcet.spec (0.6), epsilon=e, count=1, runs=r))
```

Figure 3.4 shows the results. When the number of runs is small, convergence is low: only a small percentage of runs are within epsilon of the average. As the number of runs increases, convergence increases. Not surprisingly, convergence rises more rapidly when epsilon is larger: convergence to 10% of the average occurs more quickly than convergence to 5% of the average. This illustrates a general rule in agent-based modeling: the larger the number of runs, the fewer spurious results.

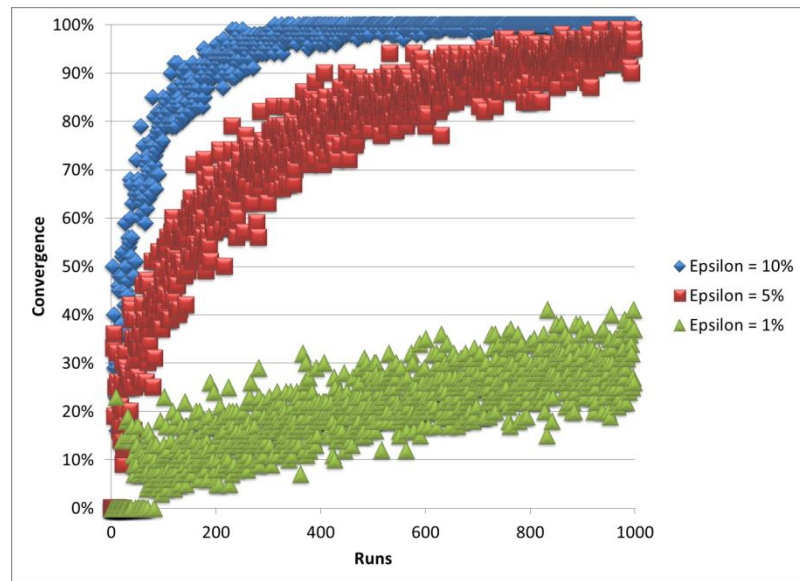


Figure 3.4: Convergence by number of runs

The preceding analysis implies that the modeler should set the number of runs very high. This conclusion would be correct if the marginal cost of each run were zero. However, this is generally not the case in computer models, because each run takes a nonzero amount of time. A tradeoff exists between the number of runs and the amount of time needed to execute the model. This code demonstrates that tradeoff:

```
import time, model, condorcet
print ("runs convergence seconds")
for r in range (1, 1000):
    start = time.time()
    pct = model.convergence (
        spec=condorcet.spec (0.6), epsilon=0.1, count=1, runs=r)
    end = time.time()
    print (r, pct, end-start)
```

Figure 3.5 shows that both convergence and elapsed time increase with the number of runs. Convergence, however, plateaus around 400 runs; elapsed time increases without

bound. The optimum number of runs, then, is the number of runs that it takes for convergence to plateau. Beyond that point, the marginal benefit is zero, but the marginal cost is greater than zero. For the single-voter model with $P = 0.6$, the optimal number of runs is around 400.

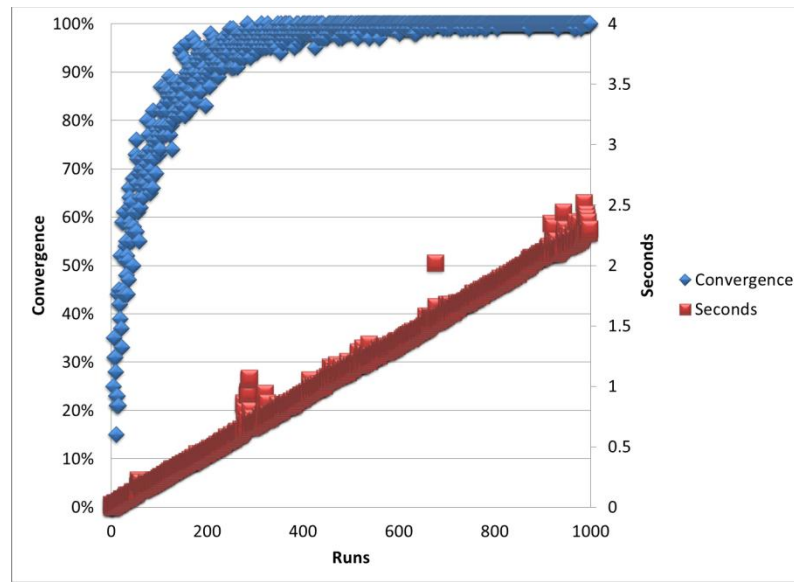


Figure 3.5: Convergence and run time by number of runs

The preceding invocations have explored the single-voter model. The single-voter model requires multiple runs per invocation, because each vote is always unanimous, and thus many runs are necessary to detect the overall trend. But when the number of voters exceeds 1, each vote is no longer unanimous. (If there are five voters, for example, there are six possible outcomes: 0-5, 1-4, 2-3, 3-2, 4-1, and 5-0. Only two of these are unanimous.) Increasing the number of voters reduces the “swing” between adjacent model results: the difference between a 0-1 vote and a 1-0 vote is 100% (all

voters switch votes), while the difference between a 0-5 vote and a 1-4 vote is 20% (only one of the five voters switches votes). The reduced swing implies that convergence should occur more quickly as the number of voters increases. Indeed, Condorcet's theorem can be rephrased in exactly those terms: as the number of voters increases, the number of runs required for convergence should approach 1. The relationship between the number of voters and the rate of convergence is explored by the following code, which invokes the model using 1, 11, 101, and 1001 voters:

```
import model, condorcet
print ("runs count convergence")
for r in range (1, 400):
    for c in [1, 11, 101, 1001]:
        print (r, c, model.convergence (
            spec=condorcet.spec (0.6), epsilon=0.1, count=c, runs=r))
```

Figure 3.6 shows the inverse relationship between number of voters and number of runs required for convergence. As the number of voters increases, the number of runs required for convergence decreases. When the number of voters is 1001, convergence is almost immediate. This is another illustration of Condorcet's theorem.

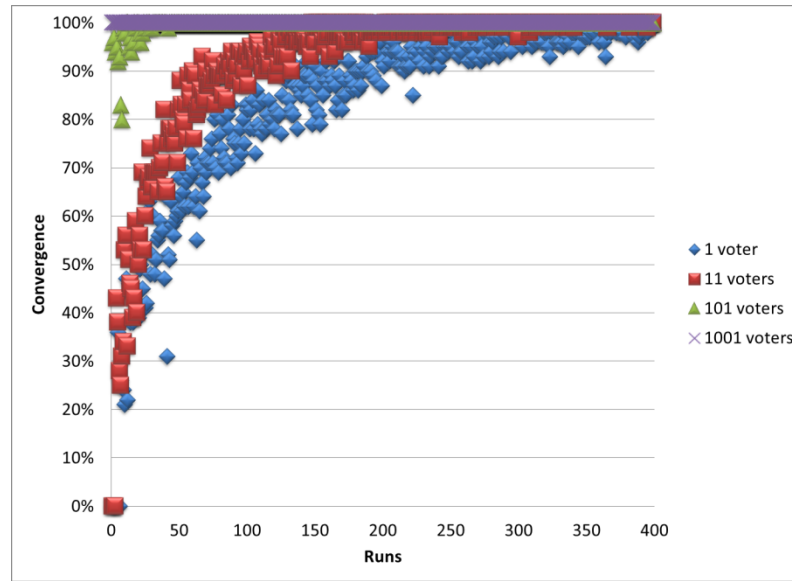


Figure 3.6: Convergence by voters and number of runs

3.4. Relaxing the assumptions

As Mueller (2003, 130) notes, Condorcet's formulation of his theorem is based on three assumptions: homogeneity, independence, and sincerity. Given the fact that none of these assumptions may hold, is Condorcet's theorem relevant to the real world? The agent-based approach allows exploration of this question. The model's plug-in framework allows great flexibility in relaxing Condorcet's assumptions. These assumptions are relaxed by providing alternate specifications of the model. This section analyzes some of these specifications.

3.4.1. Relaxing the homogeneity assumption

In Condorcet's formulation, voters are homogeneous: each voter votes correctly with a fixed likelihood that is common to all voters. A simple way to introduce

heterogeneity is to allow voters to possess different likelihoods of correctness. In the following specification, likelihoods of correctness are distributed normally across the population of voters:

```
import model, likelihoods, distributions
print ("stddev pct_correct")
for s in range (50):
    print (s / 10, model.percent (
        spec={"likelihood": likelihoods.distributions (
            distributions.normal (0.6, s / 10))}))
```

The only difference between this specification and the Condorcet specification is that the likelihood of correctness now varies by agent. Likelihoods are distributed normally with a mean of 60% and standard deviations ranging from 0 to 5, in increments of 0.1. The range of the normal distribution is infinite, so values less than 0% are mapped to 0%, and values greater than 100% are mapped to 100%.

When the variance is 0, the model is identical to Condorcet's specification, and the likelihood that voters reach the correct decision approaches 100%. As the variance increases, the tails become thicker. The mean of the distribution is still 60%, but the distribution is more dispersed. The lower and upper boundaries, 0% and 100%, are encountered with increasing frequency. The polarizing effect reduces the likelihood of a correct vote: given a finite number of voters and an increasing variance, the number of voters who vote correctly 0% of the time approaches the number of voters who vote correctly 100% of the time. Fewer and fewer voters are left to decide the contest. This works directly against the Law of Large Numbers—the principle that underlies Condorcet's theorem. In the presence of heterogeneity, more voters are needed to

produce the correct vote. Figure 3.7 shows how the likelihood of a correct vote decreases as the variance of the distribution increases.

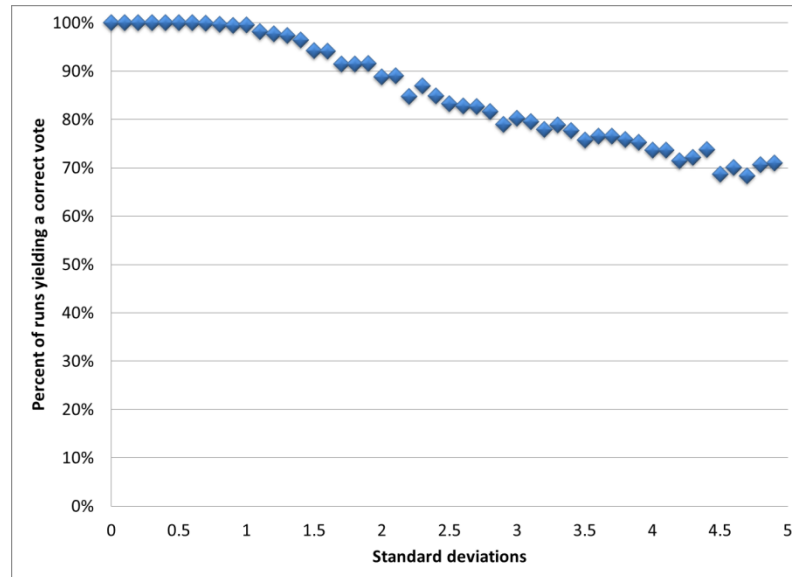


Figure 3.7: Likelihood of a correct vote by variance of voter distribution

The agent-based approach is especially useful in situations that do not lend themselves to closed-form solutions. Consider, for example, the Cauchy distribution. The Cauchy distribution has no expected value; all of its moments are undefined. This implies that the closed-form version of the Law of Large Numbers cannot be used to explore the Cauchy distribution. However, the agent-based approach can be used to investigate its properties. The following code compares two voter distributions: a normal distribution with mean (and median) 60% and standard deviation 1.0, and a Cauchy distribution with median 60%:

```

import model, likelihoods, distributions
normal = distributions.normal (0.6, 1.0)
cauchy = distributions.cauchy (0.6)
normalspec = {"likelihood": likelihoods.distribution (normal)}
cauchyspec = {"likelihood": likelihoods.distribution (cauchy)}
print ("count normal cauchy")
for c in [1, 11, 101, 1001, 10001]:
    print (c,
        model.percent (spec=normalspec, count=c)
        model.percent (spec=cauchyspec, count=c))

```

As Figure 3.8 shows, Condorcet's results emerge under both distributions, but slightly faster under the normal distribution than under the Cauchy.

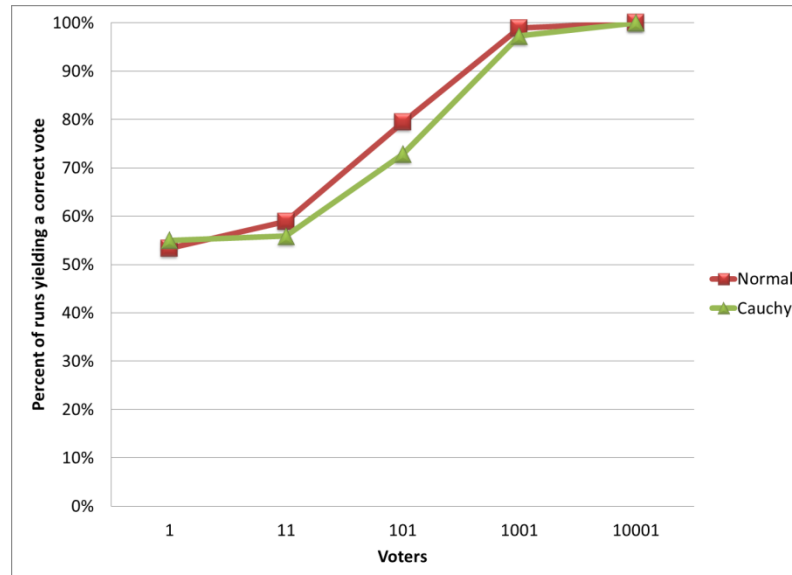


Figure 3.8: Likelihood of a correct vote by number of voters and distribution

Using distributions to define voter likelihoods is a simple way to relax Condorcet's assumption of homogeneity. It demonstrates that heterogeneity works against the Law of Large Numbers. But applying this approach to the real world is difficult: there is no reason to believe that likelihoods of correctness follow any

particular distribution. It seems probable that many different characteristics contribute to the likelihood that an agent votes correctly. For real-world applications, a more robust form of heterogeneity is needed.

So far, all invocations of the model have used the `null()` agent factory. To explore full heterogeneity, the `characteristic()` factory can be used instead. The `characteristic()` factory allows the modeler to define agents by any number of characteristics. These characteristics can then be used to influence the agent's likelihood of voting correctly. For example, suppose agents are believed to have two characteristics, C1 and C2, which shape voting behavior. Suppose that $C1 \in \{0.0, 1.0\}$ and $C2 \in \{0.0, 0.5, 1.0\}$. Then there are six possible agent types: $(C1, C2) \in \{(0.0, 0.0), (0.0, 0.5), (0.0, 1.0), (1.0, 0.0), (1.0, 0.5), (1.0, 1.0)\}$. Each of these agent types may exhibit a different likelihood of correctness. (Characteristics need not be discrete; C1 and C2 are presented as discrete for ease of exposition.)

The model allows C1 and C2 to be distributed differently and to covary. These features allow the model to simulate real-world distributions. To continue the example, let C1 be distributed such that $P(C1=0.0) = 1/3$ and $P(C1=1.0) = 2/3$, and let C2 be distributed such that $P(C2=0.0) = 3/20$, $P(C2=0.5) = 7/20$, and $P(C2=1.0) = 1/2$. Then Table 3.1 shows the expected distribution of agent types across the agent population, assuming that C1 and C2 are uncorrelated.

Table 3.1: Expected joint distribution of C1 and C2, no correlation

	C1 = 0.0	C1 = 1.0	Total
C2 = 0.0	5.00%	10.00%	15.00%
C2 = 0.5	11.67%	23.33%	35.00%
C2 = 1.0	16.67%	33.33%	50.00%
Total	33.33%	66.67%	100.00%

When C1 and C2 are uncorrelated, $P(C2=c|C1)$ equals $P(C2=c)$: C1 and C2 are independent. When C1 and C2 are correlated, $P(C2=c|C1)$ is no longer equal to $P(C2=c)$. Table 3.2 extends Table 3.1 by depicting two additional cases: $\text{corr}(C1,C2) = -1$ and $\text{corr}(C1,C2) = 1$. The independent distributions of C1 and C2 are unchanged, but the joint distributions are different. When $\text{corr}(C1,C2) = -1$, an agent with $C1 = 0.0$ is *least* likely to have $C2 = 0.0$. When $\text{corr}(C1,C2) = 1$, an agent with $C1 = 0.0$ is *most* likely to have $C2 = 0.0$.

Table 3.2: Expected joint distribution of C1 and C2, correlations -1 and 1

<i>corr=-1</i>	C1 = 0.0	C1 = 1.0	Total
C2 = 0.0	0.00%	15.00%	15.00%
C2 = 0.5	0.00%	35.00%	35.00%
C2 = 1.0	33.33%	16.67%	50.00%
Total	33.33%	66.67%	100.00%
<i>corr=1</i>	C1 = 0.0	C1 = 1.0	Total
C2 = 0.0	15.00%	0.00%	15.00%
C2 = 0.5	18.33%	16.67%	35.00%
C2 = 1.0	0.00%	50.00%	50.00%
Total	33.33%	66.67%	100.00%

These distributions may be created in the model by using the `spaced_list()` distribution.

The following code creates 10,000 agents, each with characteristic $C1 \in \{0.0, 1.0\}$ and

$C2 \in \{0.0, 0.5, 1.0\}$, distributed as specified above:

```
import factories, distributions
c1 = distributions.spaced_list ([1, 2])      # 1/3, 2/3
c2 = distributions.spaced_list ([3, 7, 10]) # 3/20, 7/20, 10/20
for corr in [-1, 0, 1]:
    print (factories.profile (factories.characteristic (10000, [
        ("c1", c1, []),
        ("c2", c2, [corr])])))
```

The above code considers three values for $\text{corr}(C1, C2)$: -1, 0, and 1. The results are shown in Table 3.3. Notice that the results are very close to the expected results as shown in Tables 3.1 and 3.2.

Table 3.3: Observed joint distribution of $C1$ and $C2$, various correlations

<i>corr=-1</i>	$C1 = 0.0$	$C1 = 1.0$	Total
$C2 = 0.0$	0.00%	15.00%	15.00%
$C2 = 0.5$	0.00%	35.48%	35.48%
$C2 = 1.0$	33.22%	16.30%	49.52%
Total	33.22%	66.78%	100.00%
<i>corr=0</i>	$C1 = 0.0$	$C1 = 1.0$	Total
$C2 = 0.0$	5.15%	10.03%	15.18%
$C2 = 0.5$	12.14%	22.70%	34.84%
$C2 = 1.0$	16.85%	33.13%	49.98%
Total	34.14%	65.86%	100.00%
<i>corr=1</i>	$C1 = 0.0$	$C1 = 1.0$	Total
$C2 = 0.0$	14.77%	0.00%	14.77%
$C2 = 0.5$	18.27%	17.59%	35.86%
$C2 = 1.0$	0.00%	49.37%	49.37%
Total	33.04%	66.96%	100.00%

Table 3.3 shows that the joint distribution of C1 and C2 shifts with their degree of correlation. The table shows three different degrees of correlation: perfect negative correlation, zero correlation, and perfect positive correlation. Intermediate correlations may be explored using the model as well. Consider the subset of agents with $C1 = C2 = 0.0$, which appears three times in Table 3.3. When $\text{corr}(C1, C2)$ is -1, these agents constitute about 0% of the constructed population. When $\text{corr}(C1, C2) = 0$, these agents constitute about 5% of the constructed population. When $\text{corr}(C1, C2) = 1$, these agents constitute about 15% of the constructed population. So as the correlation shifts from -1 to 1, the proportion of agents with $C1 = C2 = 0$ shifts from 0% to 15%. The following code traces this shift, and explores the intermediate correlations:

```
import factories, distributions
c1 = distributions.spaced_list ([1, 2])
c2 = distributions.spaced_list ([3, 7, 10])
print ("corr percent")
for c in range (-100, 101):
    agents = factories.characteristic (1000, [
        ("c1", c1, []),
        ("c2", c2, [c / 100])])
    print (c / 100, factories.profile (agents)[{"c1": 0, "c2": 0}])
```

Figure 3.9 depicts the results. The discrete nature of characteristics C1 and C2 causes the shifting correlations to induce a step-wise shift in the proportion of the agent population with $C1 = C2 = 0.0$.

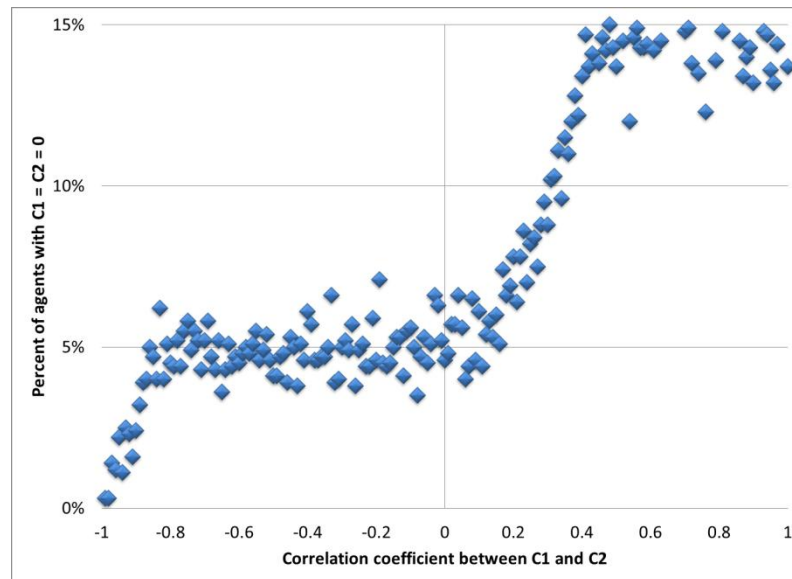


Figure 3.9: Percent of agents with $C1=C2=0$ by $\text{corr}(C1,C2)$

The `characteristic()` agent factory, then, permits an impressive degree of agent heterogeneity. But in order to exploit this capability, one must identify a source for distributions of characteristics in a population of interest. When the population of interest is the United States public at large, a good source of characteristic distributions is the General Social Survey (GSS). The GSS is a biennial face-to-face survey conducted by the University of Chicago’s National Opinion Research Center. Its website claims that “[e]xcept for the U.S. Census, the GSS is the most frequently analyzed source of information in the social sciences⁴.”

The GSS asks respondents several different types of questions. One set of questions is designed to solicit opinions on contemporary issues. Another set is designed to collect background and demographic information. A third set is designed to test

⁴ Retrieved from <http://www3.norc.og/GSS+Website/About+GSS/> on December 10, 2011.

general scientific knowledge. For purposes of Condorcet's theorem, the second and third sets make the GSS a uniquely valuable survey: the second set can be used to define agent distributions, and the third set can be used to infer likelihoods of correctness. Taken together, these two sets of questions allow Condorcet's theorem to be tested against a real-world population.

Table 3.4 shows a selection of the background and demographic questions. Eight questions are shown. Four of the questions are volitive—the answer depends at least in part on choices made by the respondent, and may (arguably) be affected by policy—and four are not.

Table 3.4. GSS background and demographic questions

Name	Question text
AGE	13. Respondent's age
EDUC	15. What is the highest grade in elementary school or high school that you finished and got credit for?
SEX	23. Code respondent's sex
RACE	24. What race do you consider yourself?
PARTYID	56. Generally speaking, do you usually think of yourself as a Republican, Democrat, Independent, or what?
HISPANIC	1601. Are you Spanish, Hispanic, or Latino/Latina? IF YES: Which group are you from?
WORDSUM	1612k. Total number of correct words.
CONRINC	1658. Inflation-adjusted personal income.

The only non-self-explanatory variable in Table 3.4 is WORDSUM. To determine WORDSUM, the respondent is asked to select the correct definition of ten words, one at a time. Each word is accompanied by four definitions, one of which is correct.

WORDSUM is the total number of words defined correctly. The source of the words in

the vocabulary test is the Weschler Adult Intelligence Scale. WORDSUM tends to correlate with other measures of general intelligence, so it is sometimes used as a proxy for intelligence (Zhu and Weiss 2005). Figure 3.10 shows the distribution of WORDSUM for the 25,638 respondents who took the vocabulary test.

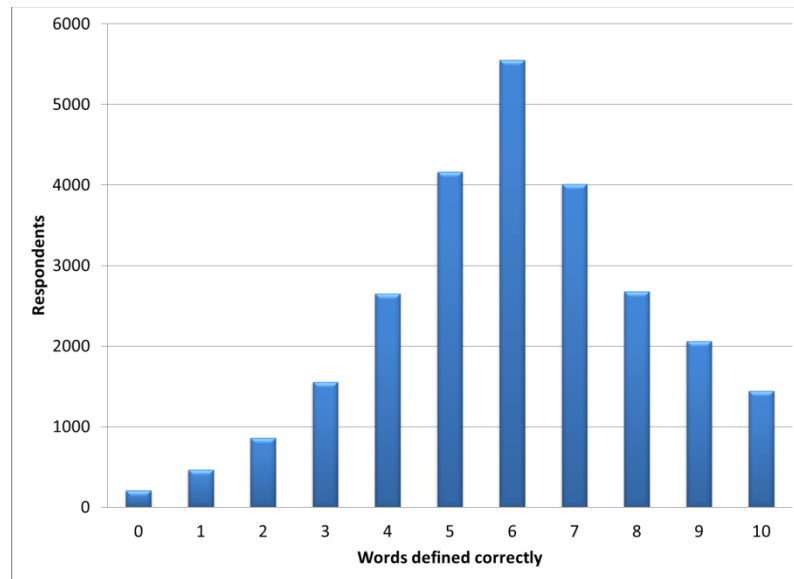


Figure 3.10: Distribution of WORDSUM values from the GSS

These eight questions can be used to define a population that represents the United States. Each question corresponds to an agent characteristic whose distribution can be derived from GSS data. For example, the PARTYID question allows each respondent to describe herself as a strong Democrat (0), a moderate Democrat (1), a Democrat-leaning independent (2), an independent (3), a Republican-leaning independent (4), a moderate Republican (5), or a strong Republican (6). In the GSS model specification, this characteristic is called `repub`, and is expressed as follows:

```

import distributions
repub = distributions.unspaced_list ([
    (0, 8761),
    (1, 11697),
    (2, 6508),
    (3, 8126),
    (4, 4764),
    (5, 8755),
    (6, 5356)
])

```

The numbers in the above definition were taken from the GSS. For the period 1972-2010, a total of 53,967 respondents answered the PARTYID question. Of these, 8,761 described themselves as strong Democrats, 11,697 described themselves as moderate Democrats, and so on. The result of this code is a characteristic called `repub`, with discrete values evenly spaced between 0 (corresponding to GSS coding 0) and 1 (corresponding to GSS coding 6). In the case of the GSS specification, there are eight such characteristics, one for each of the questions in Table 3.4. This means the GSS specification defines eight characteristics: `age`, `male`, `black`, `hisp`, `educ`, `word`, `inc`, and `repub`. (The definitions of these distributions appear in `gss.py`; see Appendix 3.B.2). Additional characteristics can, of course, be added at any time with minimal effort.

When multiple characteristic distributions are provided, their correlations may also be provided. This information is conveyed to the agent factory by building the lower triangle of the correlation matrix, one row at a time. The following code defines the eight agent characteristics as derived from the GSS, including their correlations:

```

import factories
def factory (count):
    return factories.characteristic (count, [
        ("age",    age,    []),
        ("male",   male,   [ 0.01]),
        ("black",  black,  [-0.06,-0.08]),
        ("hisp",   hisp,   [-0.13, 0.03,-0.11]),
        ("educ",   educ,   [ 0.07,-0.04,-0.10,-0.24]),
        ("word",   word,   [ 0.17,-0.05,-0.17,-0.19,0.43]),
        ("inc",    inc,    [ 0.23, 0.25,-0.11,-0.12,0.37,0.22]),
        ("repub",  repub,  [ 0.02, 0.11,-0.29,-0.10,0.03,0.06,0.11])
    ])

```

In this example code, the GSS specification is used to create 10,000 agents. One agent is selected at random and its characteristics printed.

```

import gss, random
agents = gss.factory (10000)
print (agents[int (random.random() * len (agents))])

```

The above code produces output like this:

```

{'male': 0.0, 'word': 0.5, 'age': 0.5070422535211268, 'repub':
0.8333333333333334, 'hisp': 0.0, 'black': 0.0, 'educ': 0.8,
'inc': 0.4501699957501063}

```

The above output means that in this instance, the randomly selected agent was female, scored 5/10 on the vocabulary test, was almost 51 years old, a moderate Republican, not Hispanic or black, had a Bachelor's degree, and earned about \$112,500 per year in 2010 dollars.

What makes the GSS useful in the context of Condorcet's theorem is that it asks more than just demographic questions; it also asks questions concerning general scientific knowledge. A typical example of a scientific question in the GSS is EARTHSUN. The preamble to EARTHSUN (and several related questions) is as follows:

1044. Now, I would like to ask you a few short questions like those you might see on a television game show. For each statement that I read, please tell me if it is true or false. If you don't know or aren't sure, just tell me so, and we will skip to the next question. Remember true, false, or don't know.

EARTHSUN is the tenth question in section 1044, and reads as follows:

j. Now, does the Earth go around the Sun, or does the Sun go around the Earth?

There are two notable aspects to this question. First, it is a positive question, not a normative question. It does not ask whether the respondent thinks the Earth *should* orbit the Sun; it asks the respondent, *does* the Earth orbit the Sun? Second, it offers the respondent the option of saying “don’t know.” This means that when a respondent answers the question with anything other than “don’t know,” she implies that she believes she *does* know. The respondent is evaluating alternatives and rendering a judgment on which alternative is correct. In other words, the respondent is voting.

It is probably not too much of a stretch to say that there is now a consensus in the scientific community that the Earth orbits the Sun. Stating that this is “true,” however, invites an unnecessary detour into epistemology. For purposes of this analysis, let us assume that if the Earth orbits the Sun, EARTHSUN has an objectively correct answer. When interpreted in this way, EARTHSUN allows us to estimate a likelihood of correctness. The results from the GSS are interesting, to say the least. This question was asked of 4,310 respondents. Of the 3,966 respondents who answered, 3,144 (79.3%) said the Earth orbits the Sun, and 822 (20.7%) said the Sun orbits the Earth. For this

particular question, then, and assuming that the Earth orbits the Sun, the average respondent has a likelihood of correctness of 79.3%.

Some readers might find 79.3% to be distressingly low; for many readers, a value closer to 99% might have been expected. But note that 79.3% is significantly larger than 50%. Condorcet's theorem tells us that an electorate composed of these respondents would be very likely to render, in the aggregate, a correct vote on EARTHSUN. That is, if every respondent answered the question correctly with $P=0.793$, a correct vote would be assured. Yet the preceding analysis revealed that heterogeneity works against the Law of Large Numbers. So an average likelihood of correctness of 79.3% may not, in fact, be large enough to produce a correct aggregate vote. It depends on the heterogeneity of the voters.

In the GSS specification, voters are heterogeneous along eight characteristics. The results of a simple linear regression, using EARTHSUN as the dependent variable (with the correct answer coded as 1 and the incorrect answer coded as 0) and the eight characteristics as the independent variables, are shown in Table 3.5.

Table 3.5: EARTHSUN as a function of GSS agent characteristics

Characteristic	Coefficient
(Constant)	0.369
Age (normalized)	-0.158
Sex (male=1)	0.086
Race (black=1)	-0.125
Hispanic (yes=1)	0.013
Years of education (normalized)	0.398
Word score (normalized)	0.346
Real income (normalized)	0.042
Degree Republican (normalized)	-0.013

These coefficients may be combined to create a likelihood function for a GSS agent. The following code creates 10,001 agents whose characteristics are representative of the GSS sample. It uses the regression coefficients to form a likelihood of correctness for EARTHSUN:

```
import gss, likelihoods
def likelihood (agents):
    return likelihoods.characteristic (agents, lambda agent: 0.369+
        agent["age"] * -0.158 +
        agent["male"] * 0.086 +
        agent["black"] * -0.125 +
        agent["hisp"] * 0.013 +
        agent["educ"] * 0.398 +
        agent["word"] * 0.346 +
        agent["inc"] * 0.042 +
        agent["repub"] * -0.013)
agents = gss.factory (10001)
print (sum (likelihood (agents)) / len (agents))
```

The result returned by this code is 79.0%, which is very close to the 79.3% reported by the GSS. This confirms that the agents created by the GSS agent factory are representative of the GSS sample. Now the model can be used to determine whether these voters will, in the aggregate, vote correctly on EARTHSUN:

```
import model
print (model.percent (spec={
    "factory" : gss.factory,
    "likelihood" : likelihood
}, count=10001, runs=1))
```

The model says yes: a population of 10,001 voters, with characteristics distributed according to the GSS sample, will vote correctly on the question of whether the Earth orbits the Sun. Condorcet's theorem proved this result for a homogeneous population;

the model has replicated the result for a heterogeneous population similar to that of the United States.

In this case, the agent's likelihood function has a precise meaning: it is the likelihood that an agent will correctly answer the EARTHSUN question, based on eight agent characteristics. A more general likelihood function would compute the likelihood of correctness for *any* question, not just EARTHSUN. An agent who answers the EARTHSUN question correctly might not enjoy the same success with more complex questions. Consider an array of questions, each somewhat more complex than its predecessor, beginning with EARTHSUN and ending with a question of infinite complexity. The following code gradually discounts the likelihood function to reflect such an array of questions:

```
import gss, model, likelihoods
def likelihood (agents, complexity):
    return likelihoods.characteristic (agents, lambda agent:
        ((1 - complexity) * (0.369 +
            agent["age"] * -0.158 +
            agent["male"] * 0.086 +
            agent["black"] * -0.125 +
            agent["hisp"] * 0.013 +
            agent["educ"] * 0.398 +
            agent["word"] * 0.346 +
            agent["inc"] * 0.042 +
            agent["repub"] * -0.013)))

print ("complexity pct_correct")
for c in range (101):
    print (c / 100, model.percent (spec={
        "factory" : gss.factory,
        "likelihood" : lambda agents: likelihood (agents, c / 100)}))
```

The results are shown in Figure 3.11. They suggest that GSS agents produce correct votes even on more complex subjects, so long as the complexity does not cause each agent's likelihood of correctness to diminish more than about 35%.

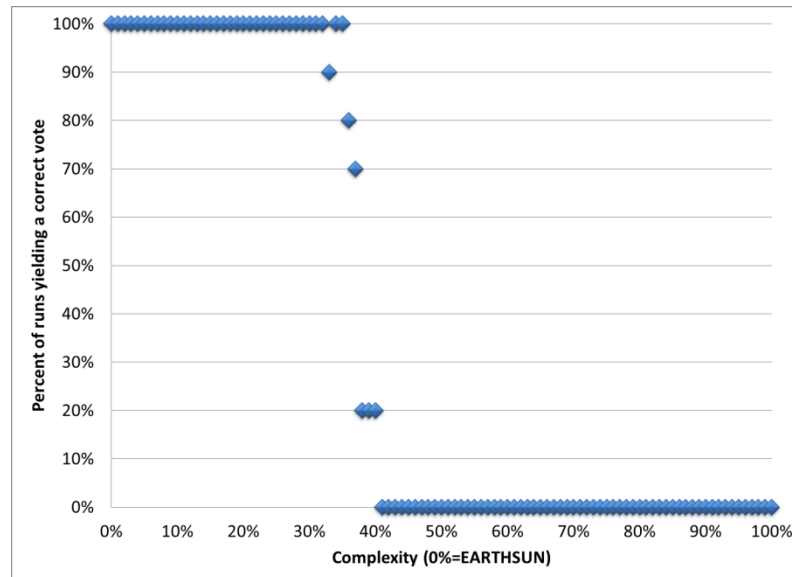


Figure 3.11: Likelihood of a correct vote by complexity: EARTHSUN only

The likelihood function depicted above was derived from the results of one sample question, EARTHSUN. A more robust likelihood function can be obtained from averaging the answers of many sample questions. The GSS contains 33 questions for which, in the context of this model, correct answers may be said to exist. (Notably excluded from this list are questions involving the evolution of the human species and the effect of humans on the global climate: even if these issues enjoy a degree of consensus, their inclusion in this model would invite unnecessary criticism. Interested readers are

encouraged to use the GSS to develop alternative models using the same framework.)

The 33 questions are shown in Table 3.6.

Table 3.6. GSS scientific questions

Name	Question text
ASTROSCI	1037. Would you say that astrology is very scientific, sort of scientific, or not at all scientific?
EXPDESGN	1041. Now, please think about this situation. Two scientists want to know if a certain drug is effective against high blood pressure. The first scientist wants to give the drug to one thousand people with high blood pressure and see how many of them experience lower blood pressure levels. The second scientist wants to give the drug to five hundred people with high blood pressure, and not give the drug to another five hundred people with high blood pressure, and see how many in both groups experience lower blood pressure levels. Which is the better way to test this drug?
EXPTTEXT	1042. Why is it better to test the drug this way?
ODDS1	1043. Now, think about this situation. A doctor tells a couple that their genetic makeup means that they've got one in four chances of having a child with an inherited illness. a. Does this mean that if their first child has the illness, the next three will not have the illness?
ODDS2	b. Does this mean that each of the couple's children will have the same risk of suffering from the illness?
HOTCORE	1044. Now, I would like to ask you a few short questions like those you might see on a television game show. For each statement that I read, please tell me if it is true or false. If you don't know or aren't sure, just tell me so, and we will skip to the next question. Remember true, false, or don't know. a. First, the center of the Earth is very hot. Is that true or false?
RADIOACT	b. All radioactivity is man-made. (Is that true or false?)
BOYORGRL	c. It is the father's gene that decides whether the baby is a boy or a girl. (Is that true or false?)
LASERS	d. Lasers work by focusing sound waves. (Is that true or false?)
ELECTRON	e. Electrons are smaller than atoms. (Is that true or false?)
VIRUSES	f. Antibiotics kill viruses as well as bacteria. (Is that true or false?)
CONDRIFT	h. The continents on which we live have been moving their locations for millions of years and will continue to move in the future. (Is that true or false?)
EARTHSUN	j. Now, does the Earth go around the Sun, or does the Sun go around

	the Earth?
SOLARREV	k. How long does it take for the Earth to go around the Sun: one day, one month, or one year?
SCIIMP3	1072. Now I'm going to read you some statements about science and scientists. Please look at Card B19. How important are each of the following in making something scientific? c. The conclusions are based on solid evidence.
SCIIMP4	d. The researchers carefully examine different interpretations of the results, even ones they disagree with.
SCIIMP7	g. Other scientists repeat the experiment, and find similar results.
NOSUN	1077. The next few questions are about the Arctic and the Antarctic. The Arctic is the region around the North Pole; Antarctic is the region that contains the South Pole. These questions are like ones you might see on a television game show. If you don't know or aren't sure, just tell me so, and we will skip to the next question. Remember true, false, or don't know. b. The sun never shines at the South Pole. (Is that true or false?)
NANOKNW1	1082. Here are a couple of true-false questions about nanotechnology. As before, if you don't know or aren't sure, just tell me so, and we will skip to the next question. Remember true, false, or don't know. a. Nanotechnology involves manipulating extremely small units of matter, such as individual atoms, in order to produce better materials. (Is that true or false?)
NANOKNW2	b. The properties of nanoscale materials often differ fundamentally and unexpectedly from the properties of the same materials at larger scales. (Is that true or false?)
SCITEST2	1415. For each statement below, just check the box that comes closest to your opinion of how true it is. In your opinion, how true is this? b. Antibiotics kill bacteria, but not viruses.
SCITEST3	c. Astrology - the study of the star signs - has some scientific truth.
SCITEST5	e. All man-made chemicals cause cancer if you eat enough of them.
CLONING	The cloning of living things produces genetically identical copies. (Is that true or false?)
MABOYGRL	DOES MOM'S GENE DECIDE BABY'S SEX
ANHEAT	Please look at Card I-1. The two objects shown there have the same mass, but object B loses heat more quickly than object A. Which combination of bodily features would be best suited to a small animal that lives in a cold climate and needs to minimize heat loss?
LFTPLANE	Which of the following is a key factor that enables an airplane to lift?
STORMTXT	DO YOU SEE LIGHTNING BEFORE HEARING THUNDER
LITMSTXT	WHY LITMUS PAPER DOESN'T CHANGE COLOR IN MIXED SOLUTION

EROSION	Which one of the following is not an example of erosion?
GENES	Traits are transferred from generation to generation through the...
UPBREATH	For which reason may people experience shortness of breath more quickly at the top of a mountain than along a seashore?
DAYNIGHT	Please look at Card I-3. Day-night rhythms dramatically affect our bodies. Probably no body system is more influenced than the nervous system. The figure on Card I-3 illustrates the number of errors made by shift workers in different portions of [missing] Based on the data illustrated in the figure, during which of these time periods did the most errors occur?
WEIGHING	As part of a laboratory experiment, five students measured the weight of the same leaf four times. They recorded 20 slightly different weights. All of the work was done carefully and correctly. Their goal was to be as accurate as possible and reduce [missing] Which of the following is the best method to report the weight of the leaf?

Table 3.6 provides the material for 33 ordinary least squares regressions, each with a scientific question as the dependent variable and the eight respondent characteristics as the independent variables. Before running the regressions, each respondent answer in Table 3.6 is recoded, with 0 meaning 0% correct and 1 meaning 100% correct. The recoding may take one of three forms, depending on the question:

Binary questions. VIRUSES asks the respondent whether antibiotics kill viruses as well as bacteria; the GSS coding is 1 for true and 2 for false. The correct answer is assumed to be false; thus 2 is recoded to 1 (correct) and 1 is recoded to 0 (incorrect).

Multiple choice questions. SOLARREV asks the respondent how long it takes for the Earth to orbit the Sun. The respondent is given three choices: one day, one month, and one year. One year is recoded to 1 (correct) and the others are recoded to 0 (incorrect).

Sliding scale questions. SCITEST5 asks the respondent whether all man-made chemicals cause cancer if eaten in sufficient quantities. The respondent is given four choices: definitely true, probably true, probably not true, definitely not true. Definitely not true is recoded to 1 (fully correct), probably not true is recoded to 0.67 (somewhat correct), probably true is recoded to 0.33 (somewhat incorrect), and definitely true is recoded to 0 (fully incorrect).

The results of the 33 regressions are summarized in Table 3.7. The independent variables appear across the top; the dependent variables appear in the far left column. Values shown are coefficient estimates with standard errors in parentheses. (Significance levels are not shown in Table 3.7, but will be explored shortly.)

Table 3.7: Regression results from the GSS

	AGE	SEX	RACE	HISPANIC	EDUC
ANHEAT	0.108 (0.165)	0.035 (0.043)	-0.094 (0.064)	0.023 (0.066)	0.625 (0.18)
BOYORGR	0.015 (0.104)	-0.179 (0.027)	-0.089 (0.043)	-0.109 (0.043)	-0.03 (0.109)
CLONING	0.267 (0.158)	0.071 (0.046)	0.041 (0.074)	0.015 (0.067)	0.061 (0.179)
CONDRIFT	-0.091 (0.066)	0.045 (0.017)	-0.112 (0.028)	-0.063 (0.028)	0.089 (0.069)
DAYNIGHT	0.009 (0.107)	-0.02 (0.029)	-0.024 (0.042)	0.063 (0.044)	0.469 (0.115)
EARTHSUN	-0.158 (0.08)	0.086 (0.022)	-0.125 (0.034)	0.013 (0.034)	0.398 (0.085)
ELECTRON	-0.135 (0.109)	0.029 (0.029)	-0.1 (0.047)	-0.016 (0.047)	0.275 (0.115)
EROSION	0.403 (0.151)	0.217 (0.04)	-0.116 (0.061)	-0.012 (0.061)	0.218 (0.16)
EXPDESGN	-0.324 (0.08)	0.008 (0.022)	0.037 (0.034)	0.035 (0.035)	0.283 (0.087)

EXPTEXT	-0.487 (0.105)	0.001 (0.028)	-0.109 (0.045)	-0.046 (0.045)	0.535 (0.114)
GENES	-0.054 (0.092)	-0.078 (0.025)	-0.223 (0.037)	-0.046 (0.037)	0.224 (0.099)
HOTCORE	-0.017 (0.052)	0.042 (0.014)	-0.049 (0.023)	-0.041 (0.022)	-0.017 (0.055)
LASERS	-0.416 (0.105)	0.229 (0.028)	-0.168 (0.045)	-0.194 (0.044)	0.221 (0.108)
LFTPLANE	0.15 (0.151)	0.207 (0.04)	-0.155 (0.062)	-0.198 (0.06)	0.271 (0.161)
LITMSTXT	-0.28 (0.189)	0.14 (0.047)	-0.12 (0.094)	-0.113 (0.078)	0.617 (0.195)
MABOYGRL	0.02 (0.302)	-0.074 (0.083)	0.011 (0.125)	0.112 (0.151)	0.973 (0.34)
NANOKNW1	0.005 (0.123)	0.018 (0.034)	-0.158 (0.065)	-0.019 (0.058)	0.04 (0.137)
NANOKNW2	0.024 (0.178)	-0.029 (0.049)	0.071 (0.112)	0.053 (0.082)	0.123 (0.194)
NOSUN	-0.133 (0.1)	0.086 (0.027)	-0.09 (0.046)	-0.06 (0.047)	0.264 (0.103)
ODDS1	-0.003 (0.064)	0.032 (0.017)	-0.108 (0.027)	-0.113 (0.028)	0.076 (0.068)
ODDS2	-0.087 (0.085)	0.056 (0.023)	-0.194 (0.036)	-0.132 (0.036)	0.165 (0.091)
RADIOACT	0.157 (0.078)	0.032 (0.021)	-0.11 (0.034)	-0.176 (0.034)	0.154 (0.085)
SCIIMP3	0.017 (0.052)	0.013 (0.014)	-0.046 (0.024)	-0.01 (0.025)	0.105 (0.057)
SCIIMP4	-0.015 (0.054)	-0.014 (0.015)	-0.063 (0.024)	-0.018 (0.026)	0.135 (0.059)
SCIIMP7	0.057 (0.065)	-0.009 (0.018)	-0.07 (0.029)	-0.015 (0.032)	0.011 (0.072)
SCITEST2	0.141 (0.08)	-0.039 (0.021)	0.021 (0.031)	0.087 (0.035)	0.183 (0.084)
SCITEST3	-0.253 (0.087)	0.048 (0.023)	-0.059 (0.034)	-0.021 (0.039)	0.129 (0.092)
SCITEST5	0.212 (0.06)	0.051 (0.016)	-0.05 (0.026)	-0.002 (0.025)	0.198 (0.063)
SOLARREV	-0.226 (0.098)	0.093 (0.027)	-0.153 (0.047)	-0.098 (0.042)	0.353 (0.103)
STORMTXT	-0.275 (0.15)	0.1 (0.04)	-0.307 (0.073)	-0.002 (0.059)	0.716 (0.158)

UPBREATH	-0.187 (0.123)	0.105 (0.032)	-0.3 (0.05)	-0.256 (0.048)	0.478 (0.133)
VIRUSES	0.161 (0.097)	-0.11 (0.026)	-0.24 (0.041)	-0.164 (0.042)	0.359 (0.103)
WEIGHING	-0.174 (0.134)	0.034 (0.034)	-0.128 (0.052)	-0.111 (0.052)	0.563 (0.142)

	WORD-SUM	CONR-INC	PARTY-ID	Constant	R ²	N
ANHEAT	0.44 (0.123)	-0.05 (0.248)	0.081 (0.066)	-0.148 (0.148)	0.106	509
BOYORGL	0.302 (0.08)	0.375 (0.123)	-0.003 (0.041)	0.626 (0.094)	0.079	1099
CLONING	0.012 (0.13)	-0.12 (0.208)	0.029 (0.068)	0.726 (0.145)	0.033	176
CONDRIFT	0.126 (0.05)	0.069 (0.079)	-0.095 (0.026)	0.834 (0.059)	0.044	1210
DAYNIGHT	0.328 (0.082)	0.139 (0.162)	0.055 (0.043)	0.317 (0.096)	0.115	551
EARTHSUN	0.346 (0.062)	0.042 (0.097)	-0.013 (0.033)	0.369 (0.071)	0.096	1274
ELECTRON	0.33 (0.084)	-0.052 (0.128)	0.062 (0.043)	0.335 (0.096)	0.046	1058
EROSION	0.345 (0.115)	0.085 (0.222)	0.055 (0.059)	-0.049 (0.136)	0.122	612
EXPDESGN	0.214 (0.062)	0.039 (0.098)	0.035 (0.033)	0.611 (0.073)	0.038	1259
EXPTEXT	0.563 (0.081)	0.116 (0.127)	0.028 (0.043)	-0.059 (0.096)	0.11	1255
GENES	0.335 (0.071)	0.005 (0.135)	-0.04 (0.037)	0.654 (0.084)	0.173	619
HOTCORE	0.116 (0.04)	0.008 (0.063)	0.015 (0.021)	0.87 (0.046)	0.026	1217
LASERS	0.392 (0.082)	0.253 (0.121)	0.047 (0.043)	0.348 (0.092)	0.173	964
LFTPLANE	0.406 (0.115)	-0.145 (0.219)	-0.018 (0.061)	0.092 (0.137)	0.119	592
LITMSTXT	0.449 (0.146)	0.179 (0.24)	0.031 (0.067)	-0.122 (0.162)	0.143	328
MABOYGL	0.443 (0.267)	-0.309 (0.515)	0.181 (0.13)	-0.193 (0.263)	0.242	101
NANOKNW1	0.107	0.032	-0.052	0.796	0.025	446

	(0.093)	(0.123)	(0.048)	(0.113)		
NANOKNW2	0.06 (0.125)	0.016 (0.192)	0.135 (0.068)	0.596 (0.158)	0.017	351
NOSUN	0.18 (0.078)	0.102 (0.104)	-0.074 (0.042)	0.649 (0.09)	0.085	503
ODDS1	0.189 (0.05)	0.105 (0.077)	0.033 (0.026)	0.714 (0.058)	0.071	1249
ODDS2	0.326 (0.066)	-0.155 (0.104)	-0.047 (0.034)	0.577 (0.077)	0.075	1249
RADIOACT	0.336 (0.062)	0.143 (0.095)	0.058 (0.032)	0.42 (0.072)	0.118	1212
SCIIMP3	0.1 (0.04)	0.05 (0.054)	0.036 (0.022)	0.762 (0.048)	0.066	487
SCIIMP4	0.105 (0.042)	0.016 (0.056)	-0.017 (0.022)	0.788 (0.05)	0.065	480
SCIIMP7	0.073 (0.05)	0.131 (0.067)	-0.038 (0.027)	0.819 (0.06)	0.041	483
SCITEST2	0.134 (0.055)	0.076 (0.104)	0.108 (0.033)	0.399 (0.067)	0.061	693
SCITEST3	0.11 (0.061)	0.049 (0.113)	-0.062 (0.036)	0.487 (0.074)	0.042	694
SCITEST5	0.258 (0.046)	0.057 (0.074)	0.029 (0.024)	0.385 (0.053)	0.092	1274
SOLARREV	0.345 (0.076)	-0.291 (0.112)	0.005 (0.04)	0.427 (0.088)	0.077	976
STORMTXT	0.394 (0.115)	0.429 (0.208)	0.024 (0.059)	-0.034 (0.138)	0.191	465
UPBREATH	0.446 (0.093)	0.279 (0.181)	0.066 (0.048)	0.182 (0.11)	0.263	622
VIRUSES	0.611 (0.075)	0.187 (0.12)	0.048 (0.04)	-0.037 (0.086)	0.191	1270
WEIGHING	0.252 (0.1)	0.376 (0.191)	0.136 (0.051)	0.195 (0.117)	0.141	622

Two interesting trends are readily gleaned from Table 3.7. First, WORDSUM always has a positive effect on correctness: the coefficient is positive in each of the 33 regressions. None of the other demographic characteristics exhibit this effect. (Even

EDUC has a few negative coefficients, though none is statistically significant.) Second, each characteristic's degree of significance varies widely across the scientific questions. The interpretation of this result is that some questions are harder than others. This implies that in order to combine these results into a single measure of likelihood of correctness, we must adjust for the difficulty of the question.

A composite coefficient for each characteristic can be constructed by computing a weighted average of all 33 reported coefficients. (This is why the variables were normalized before running the regressions.) Coefficients are weighted by their t-stats, so a coefficient with a t-stat of 6 receives twice the weight of a coefficient with a t-stat of 3. Statistically insignificant coefficients are included, but exert comparatively little influence (their t-stats are low). This construction yields a single estimate for the likelihood of correctness among the general population when confronted with an objective question of average complexity. Table 3.8 shows the composite coefficients, along with unweighted averages of their standard errors.

Table 3.8: Composite regression coefficients

Characteristic	Composite coefficient
Constant	0.636 (0.10)
Age (normalized)	-0.111 (0.11)
Sex (male=1)	0.062 (0.03)
Race (black=1)	-0.148 (0.05)
Hispanic (yes=1)	-0.111 (0.05)
Years of education (normalized)	0.385 (0.12)
Word score (normalized)	0.334 (0.09)
Real income (normalized)	0.123 (0.14)
Degree Republican (normalized)	0.031 (0.04)

If each coefficient is interpreted as a point estimate, and if we assume that the variation among agents along each characteristic is distributed normally, then an agent in the GSS specification has the following likelihood function:

```
import random, likelihoods
def likelihood (agents):
    return likelihoods.characteristic (agents, lambda agent:
        random.gauss ( 0.636, 0.10) +
        random.gauss (-0.111, 0.11) * agent["age"] +
        random.gauss ( 0.062, 0.03) * agent["male"] +
        random.gauss (-0.148, 0.05) * agent["black"] +
        random.gauss (-0.111, 0.05) * agent["hisp"] +
        random.gauss ( 0.385, 0.12) * agent["educ"] +
        random.gauss ( 0.334, 0.09) * agent["word"] +
        random.gauss ( 0.123, 0.14) * agent["inc"] +
        random.gauss ( 0.031, 0.04) * agent["repub"])
```

The GSS specification can now be used to test Condorcet's theorem against a real-world population. Consider a range of questions with complexity measured from 0 to 1, with 0 meaning as complex as the average scientific question asked by the GSS, and 1 meaning infinitely complex. The following code examines how the likelihood of a correct vote decreases as complexity increases:

```
import gss, model
print ("complexity pct_correct")
for c in range (40, 61):
    print (c / 100, model.percent (spec={
        "factory"      : gss.factory,
        "likelihood"   : gss.complexity (complexity)}))
```

Figure 3.12 shows the results. A population similar to that of the United States could be expected to vote correctly on scientific issues considerably more complex than those asked by the General Social Survey: not until complexity approaches 0.5 does the vote depart from certainty.

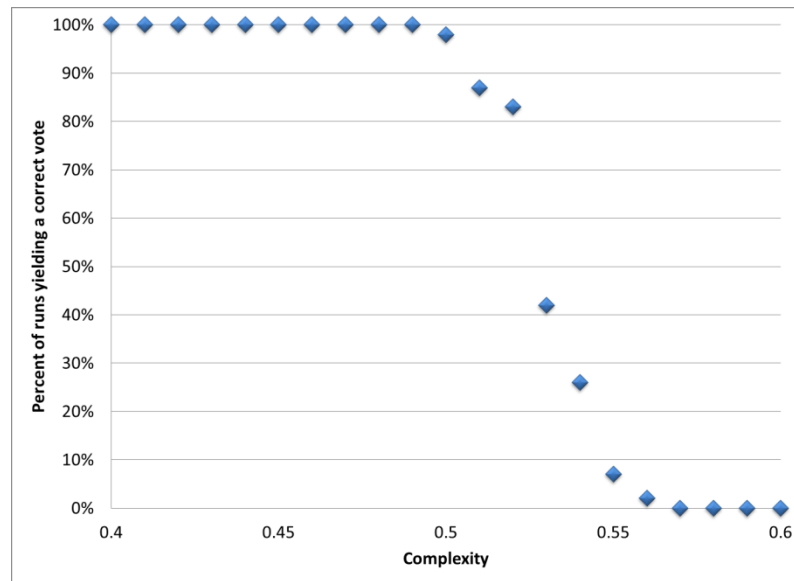


Figure 3.12: Likelihood of a correct vote by complexity: all questions

In subsequent sections, the GSS specification will be used to explore specific policy recommendations. But first, there are two more assumptions to be relaxed: independence and sincerity.

3.4.2. Relaxing the independence assumption

In Condorcet's formulation of his theorem, voters vote independently, without influence from other voters. Voting is a two-stage operation: voters make independent assessments, and those assessments are then aggregated (Austen-Smith and Banks 1996). Omitted from this process is any consideration of the influence one voter may have on another. This is an unfortunate omission, because models which include agent interaction may produce strikingly different results than models which exclude it. Glaeser, Sacerdote and Scheinkman (1996), for example, apply agent interaction to a model of

crime. In their model, agents exhibit differing degrees of susceptibility to influence from peers. When some agents are highly susceptible to the influence of their peers, small swings among the low-susceptibility agents can produce large effects: the followers act as amplifiers for the leaders. Glaeser et al. suggest that this phenomenon may explain the high variance of crime rates in time and space—a phenomenon that is difficult to explain without peer effects.

Up to this point, the agent-based exploration of Condorcet's theorem has assumed voter independence. This assumption can be relaxed by adding three new functions to the model specification: confidence, network, and effect. The confidence function expresses the degree of confidence that the agent has in its own likelihood of correctness, ranging from 0 (meaning zero confidence, and so highly influenced by its peers) to 1 (meaning perfect confidence, so never influenced by its peers). The network function expresses the construction of the social networks. The effect function expresses the effect of an agent's social network: mean reversion, lifting up, pulling down, or voting as a bloc ("herding").

To explore network effects, consider first a simple invocation of Condorcet's theorem. The following code creates 10,001 identical voters, each with a likelihood of correctness of 0.51. When these voters vote independently, the model predicts that the vote will be correct over 95% of the time:

```
import model, condorcet
print (model.percent (spec=condorcet.spec (0.51), count=10001))
```

Likewise, when the likelihood of correctness is 0.49, and voters vote independently, the vote is correct less than 5% of the time. But when network effects are introduced, these results may no longer hold. In the following specification, voters do not vote independently. Instead, each voter has a confidence level, and the agent's vote is a confidence-weighted average of its own assessment and the collective assessment of other agents in its network. In other words, each agent takes into account not only its own "private signal" but also a cumulative "public signal" (Austen-Smith and Banks 1996). When there is no collective assessment—because all agents lack confidence in their own judgments—the collective assessment degenerates to a coin flip. This specification creates a single social network and uses the mean reversion effect: the voter's likelihood of correctness is adjusted toward the group mean.

```
import model, likelihoods, confidences, networks, effects
print ("likelihood, confidence pct_correct")
for l in [0.49, 0.51]:
    for c in range (101):
        print (l, c / 100, model.percent (spec={
            "likelihood" : lambda a: likelihoods.constant (a, l),
            "confidence" : lambda a: confidences.constant (a, c / 100),
            "network"     : lambda a: networks.uniform (a, 1),
            "effect"      : lambda t: effects.mean_reversion (t)}))
```

Figure 3.13 shows how voter confidence affects the likelihood of a correct vote. When voters have zero confidence, the result is essentially a coin flip, and the likelihood of a correct vote hovers around 50%. As voters increase in confidence, more and more contribute to a cumulative public signal. As confidence tends toward 1, the likelihood of

a correct vote tends toward either 0% or 100%, depending on each agent’s independent likelihood. This specification demonstrates how Condorcet’s theorem is undermined when voters do not trust their own judgments.

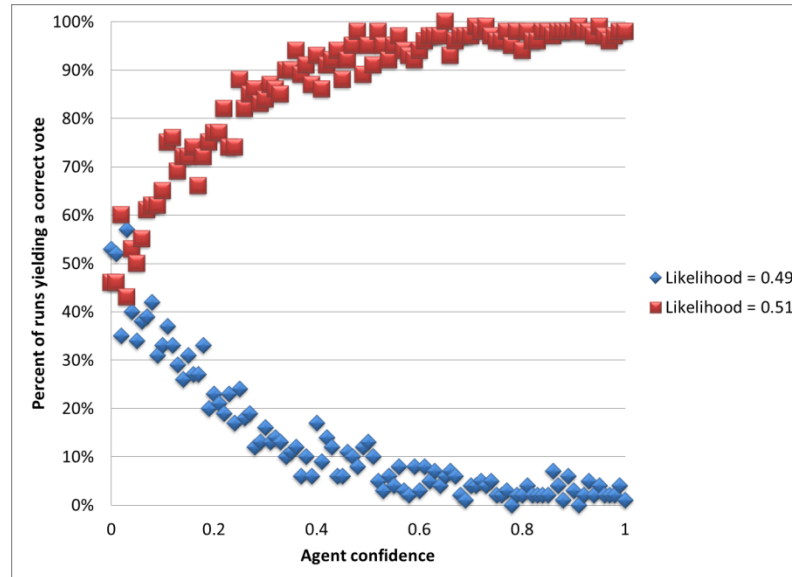


Figure 3.13: Likelihood of a correct vote by agent confidence

The above example demonstrates that relaxing independence can transform a certain vote into an uncertain vote. Relaxing independence may also yield an even more interesting result: it can transform an incorrect vote into a correct vote. Consider a specification in which voters are “wise.” In a voting context, wisdom might be defined as the degree to which an agent defers to the consensus, relative to its own likelihood of correctness. In other words, wisdom means knowing when one is likely to be wrong. A “wise” agent defers to others when its likelihood of correctness is low; an “unwise” agent does not. Table 3.9 expresses the behavior of wise agents.

Table 3.9: Behavior of wise agents

	High likelihood of correctness	Low likelihood of correctness
High confidence	Wise agents: do not defer (rightly)	Overconfident agents: do not defer (wrongly)
Low confidence	Underconfident agents: defer (wrongly)	Wise agents: defer (rightly)

The series shown in Figure 3.13 express the two columns of Table 3.9. The first column represents the $P=0.51$ situation: when agents are underconfident, they become overly deferential, and a correct vote degrades to an uncertain vote. The second column represents the $P=0.49$ situation: when agents are underconfident, they defer to better informed agents, and an incorrect vote improves to an uncertain vote. Note that in both cases, it is the *certainty* of the result, not the result itself, which is affected. In Figure 3.13, an incorrect vote can only become uncertain; it cannot become correct. To transform an incorrect vote into a correct vote, agents must be imbued with wisdom.

Wisdom is represented by the top-left to bottom-right diagonal in Table 3.9. A wise agent defers when its likelihood of correctness is low, and does not defer when its likelihood of correctness is high. Such an agent can be expressed in the model by assigning to the agent a wisdom characteristic between 0 to 1, and defining the agent's confidence as:

$$\text{confidence} = 1 + \text{wisdom} * (\text{likelihood} - 1).$$

Perfectly wise agents (wisdom=1) will defer to the consensus in proportion to their own likelihood of correctness, while perfectly unwise agents (wisdom=0) will ignore the consensus in all cases. (The wise voter specification appears in `wise.py`; see Appendix 3.B.3.)

Consider a specification in which voters are independent, and vote correctly with a likelihood that is normally distributed with mean 0.49 and standard deviation 0.1:

```
import model, likelihoods, distributions
print (model.percent (spec={
    "likelihood": lambda agents: likelihoods.distribution
        (agents, distributions.normal (0.49, 0.1))}))
```

When voters are independent, this specification yields a correct vote less than 10% of the time. Now consider the same specification, but with agents imbued with differing degrees of wisdom. The following code traces the likelihood of a correct vote as a function of agent wisdom:

```
import model, wise
print ("wisdom pct_correct")
for w in range (101):
    print (w / 100, model.percent (
        spec=wise.spec (0.49, 0.1, w / 100)))
```

As Figure 3.14 shows, the results are dramatic. When wisdom is zero, voters are independent, and the likelihood of a correct vote remains less than 10%. As wisdom increases, those voters with lower likelihoods of correctness begin listening to those with higher likelihoods of correctness. “Wisdom” leads lower likelihood voters to discount—

rationally—the values of their own likelihoods. As wisdom approaches 1, the likelihood of a correct vote exceeds 90%. It is important to realize that this occurs despite the fact that the average agent is *less than 50% likely* to vote correctly when voting independently. Given suitably wise voters, relaxing independence has transformed an incorrect vote into a correct vote.

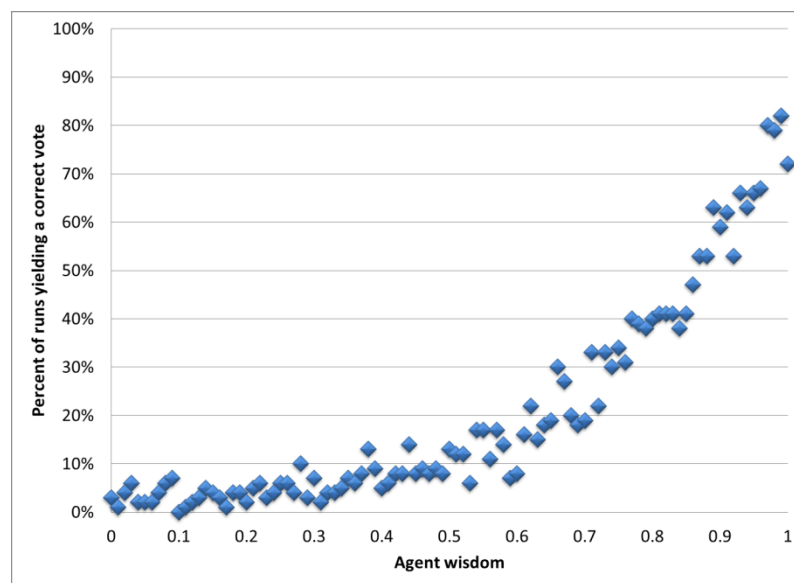


Figure 3.14: Likelihood of a correct vote by agent wisdom

The preceding result accrues from two separate mechanisms. First, wise voters recognize their own ignorance. Second, in deciding how to vote, each voter is moved toward the mean of the group. These two mechanisms may be explored separately. The following code extends the previous example to two additional network effects. The “lift up” effect moves each voter toward the most informed voter of the group; the “pull down” effect moves each voter toward the least informed voter of the group.

```

import model, wise, effects
print ("effect wisdom pct_correct")
for e, effect in enumerate (effects.all_effects):
    for w in range (101):
        print (e, w / 100, model.percent (
            spec=wise.spec (0.49, 0.1, w / 100, effect)))

```

Not surprisingly, the mean reversion effect produces results somewhere between the lift up effect and the pull down effect. As Figure 3.15 shows, the pull down effect neutralizes much of the influence of wisdom, while the lift up effect enhances it. These two extremes represent the upper and lower bounds, respectively, of the efficacy of wisdom.

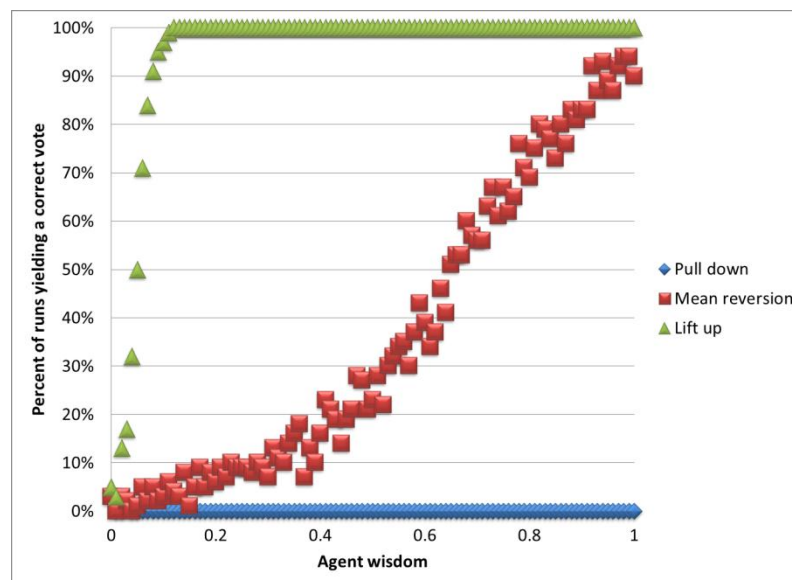


Figure 3.15: Likelihood of a correct vote by agent wisdom: detail

A fourth network effect may also be explored: the herd effect. In the herd effect, voters always vote with the majority. Note that this is not the same as the mean reversion effect. In mean reversion, the majority opinion *influences* the voter's likelihood: the

voter is taking collective wisdom into account, and the vote becomes a function of public and private information. In the herd effect, the majority opinion *defines* the voter's likelihood: the voter wishes to vote with the majority, even if this means disregarding private information. Figure 3.16 shows that the herd effect manifests a tipping point, and is more severe than the mean reversion effect. This is essentially the effect that Glaeser et al. (1996) describe: the more “followers” in society, the less likely that a single equilibrium will emerge.

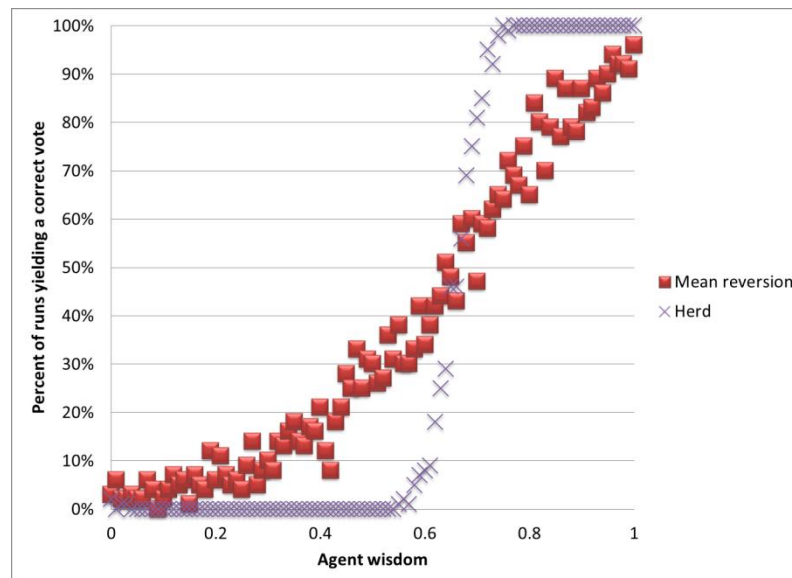


Figure 3.16: Likelihood of a correct vote by agent wisdom: herding

Figures 3.15 and 3.16 demonstrate that wisdom can improve outcomes. However, embedded in the underlying specification is the assumption that wisdom and likelihood of correctness are uncorrelated. But what if these attributes go together? Consider an agent specification in which wisdom is positively correlated with likelihood of correctness. In

such a specification, less informed voters would continue to vote with high degrees of (false) confidence. Confidence would be parabolic, with high values at the extremes of the likelihood spectrum, and low values in the middle:

$$\text{confidence} = 4 * \text{wisdom} * \text{likelihood} * (\text{likelihood} - 1) + 1$$

The “unwise” specification implements such an agent (see Appendix 3.B.4). The following code explores the difference in outcomes between true wisdom (i.e., linear confidence) and selective wisdom (i.e., parabolic confidence), once again using an agent with a normally distributed likelihood of correctness of mean 0.49 and standard deviation 0.1:

```
import model, wise, unwise
print ("spec wisdom pct_correct")
for s, spec in enumerate ([wise.spec, unwise.spec]):
    for w in range (101):
        print (s, w / 100, model.percent (
            spec=spec (0.49, 0.1, w / 100)))
```

As Figure 3.17 shows, when voters with lower likelihoods of correctness are immune from acquiring wisdom, incorrect votes are no longer transformed into correct votes. Increases in wisdom continue to have an effect, but the effect is no longer sufficient to alter the outcome of the vote. Johnson and Fowler (2011) describe how such a situation might arise.

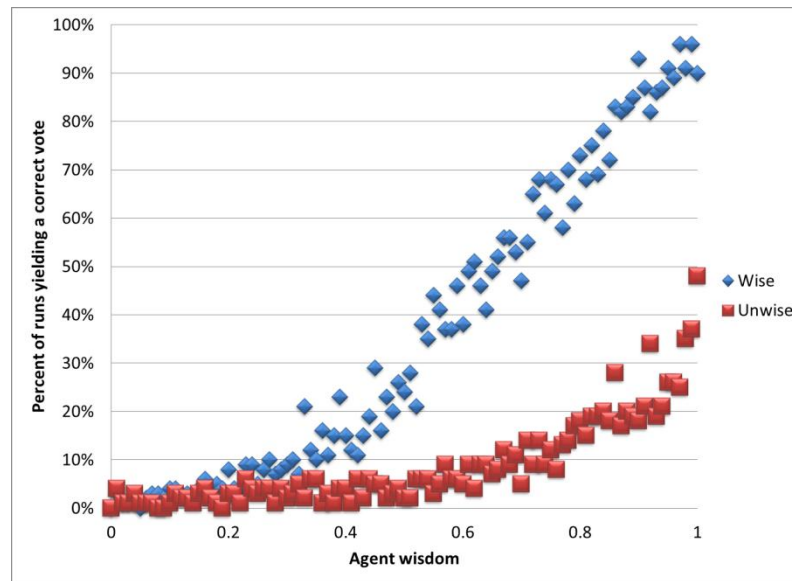


Figure 3.17: Likelihood of a correct vote by agent wisdom: wise vs. unwise

The preceding explorations of agent interaction have examined a single social network—a network in which all voters interact with all other voters. This describes the situation that arises in juries and, to some extent, in legislative bodies. In large-scale elections, however, voter influence is typically confined to a small subset of voters. This subset is called the voter’s social network. The following code explores how fragmentation of social networks can influence outcomes. (This specification is called the “fragmented” specification and appears in `fragmented.py`; see Appendix 3.B.5.) Voters are distributed normally, with a mean likelihood of correctness of 0.5 and a standard deviation of 0.1. Without network effects, these voters would vote correctly 50% of the time. Three network effects are analyzed: pull down, mean reversion, and lift up. Social fragmentation is gradually increased.


```

import model, effects, fragmented
print ("effect fragmentation pct_correct")
for e, effect in enumerate (effects.standard_effects):
    for f in range (20):
        print (e, f, model.percent (
            spec=fragmented.spec (0.5, 0.1, f * 250 + 1, 0, effect)))

```

Figure 3.18 shows the results. When there is a single social network, the lift up effect ensures a correct vote, and the pull down effect ensures an incorrect vote. As the number of social networks increases, the influence of voters in the distribution's tails wanes. If exceptionally well informed voters exert a positive effect on their peers, social fragmentation diminishes electoral outcomes (fewer peers benefit from the positive influence). If exceptionally poorly informed voters exert a negative effect on their peers, social fragmentation improves electoral outcomes (fewer peers suffer from the negative influence). If neither of these effects predominate, the result is mean reversion, and social fragmentation does not change electoral outcomes.

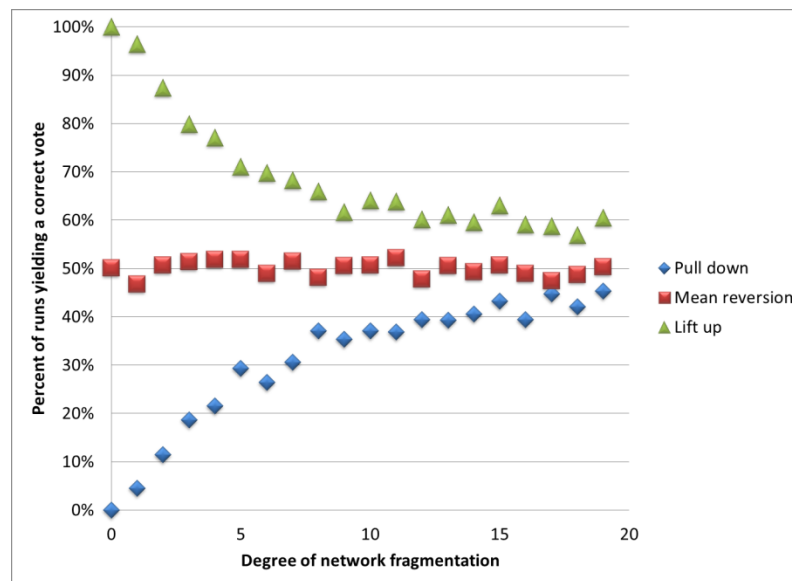


Figure 3.18: Likelihood of a correct vote by social network fragmentation

Underlying Figure 3.18 is the assumption that social networks are constructed arbitrarily. This assumption gives high-likelihood agents the opportunity to influence low-likelihood agents, and vice versa. But in the real world, social networks are not constructed arbitrarily. It is possible that some of the same factors that drive a voter's likelihood of correctness—education and intelligence, for example—may also drive a voter's social networks. When social networks are correlated with likelihood of correctness, the advantages of the lift up effect vanish quickly. The following code traces the lift up effect as social networks grow in size. Six different degrees of correlation are examined:

```
import model, effects, fragmented
print ("correlation fragmentation pct_correct")
for c in [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]:
    for f in range (20):
        print (c, f, model.percent (spec=fragmented.spec (
            0.5, 0.1, f * 250 + 1, c, effects.lift_up)))
```

As Figure 3.19 demonstrates, the lift up effect becomes meaningless when voters arrange their social networks in relation to their likelihoods of correctness. As the correlation between social networks and likelihoods of correctness increases, the opportunity for high-likelihood voters to influence low-likelihood voters evaporates.

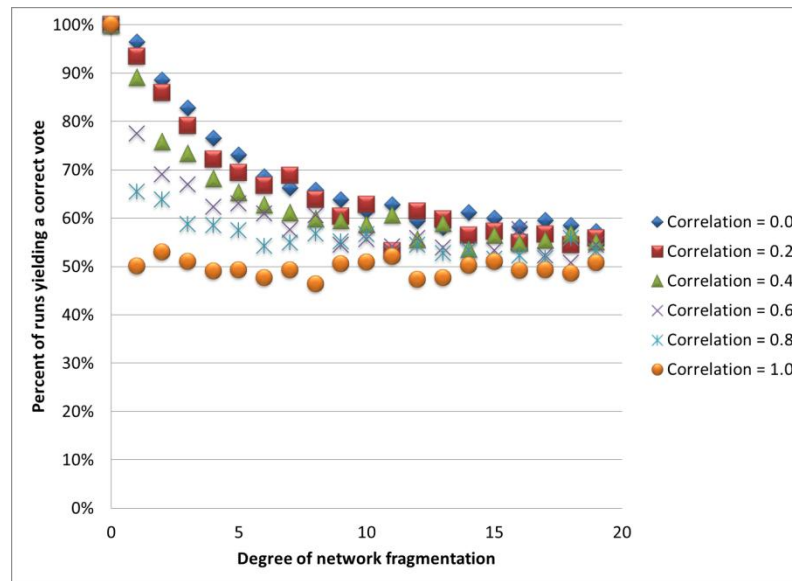


Figure 3.19: Likelihood of a correct vote by social network fragmentation: detail

The potential correlation between a voter's likelihood of correctness and his or her social network presents an identification problem for empiricists. Consider once again the GSS specification. In the GSS specification's likelihood function, the coefficient on EDUC is 0.385. To what degree does this coefficient represent learning, and to what degree does it represent social conformity, organized along educational lines? Without an instrument to distinguish between them, it is impossible to say. Nevertheless, it is possible to explore how social networks might affect the voting performance of real-world agents.

The GSS specification implements two of the confidence functions presented earlier: `wise()` and `unwise()`. In both functions, education is used as a proxy for wisdom. The `wise()` function employs a linear confidence relationship, and the `unwise()` function employs a convex confidence relationship. Each agent's social network is a function of

its characteristics. The following code explores the relationship between the two confidence functions:

```
import gss, model
print ("confidence complexity pct_correct")
for c, confidence in enumerate ([gss.wise, gss.unwise]):
    for complexity in range (40, 61):
        print (c, complexity / 100, model.percent (spec={
            "factory"      : gss.factory,
            "likelihood"    : gss.complexity (complexity / 100),
            "confidence"    : confidence,
            "network"       : gss.network (gss.profile),
            "effect"        : effects.mean_reversion}))
```

As Figure 3.20 shows, there is a margin in which wise agents outperform unwise agents. Education is used as a proxy for wisdom, so Figure 3.20 implies that an education which stresses “knowing thyself” improves voting outcomes, relative to an education which does not (holding total years of education constant).

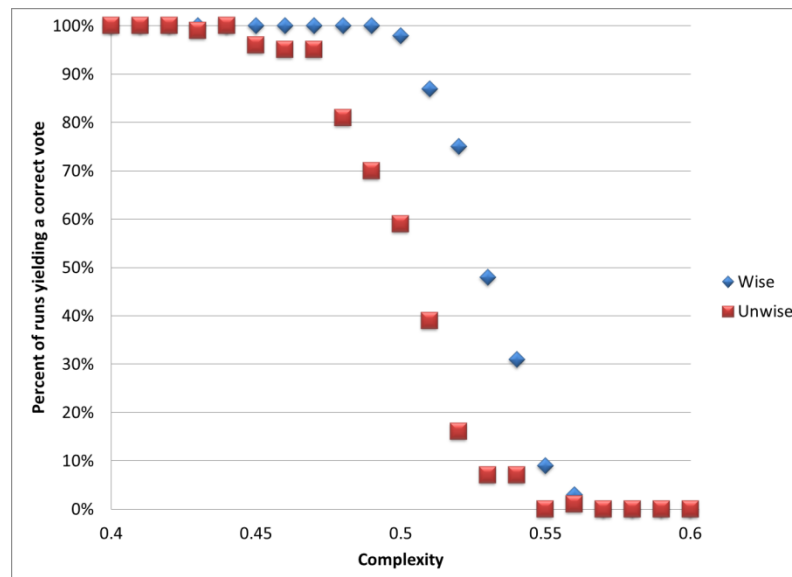


Figure 3.20: Likelihood of a correct vote by complexity: wise vs. unwise

For a real-world population, then, network effects may matter, over and above their direct influence on a voter's likelihood of correctness.

3.4.3. Relaxing the sincerity assumption

The third assumption underlying Condorcet's theorem is sincerity: the voter votes his or her conscience. The public choice literature, however, is full of examples in which voters might depart from conscience (Mueller 2003, 133). Legislators, for example, might vote in favor of a bill in order to impress constituents or other legislators, rather than because the legislator believes the bill to be sensible. Pivotal jurors may believe that the accused is innocent, yet rationally vote to convict (Feddersen and Pesendorfer 1996). Some mechanisms of insincere voting—logrolling, pandering to constituents—are beyond the scope of Condorcet's theorem. Others, however, can be explored by the agent-based model.

All instances of the model have, so far, used the null() voting strategy. The null() strategy implements sincere voting. An alternative to the null() strategy is the pivotal() strategy. In the pivotal() strategy, the agent votes its conscience, *unless it is about to cast the deciding vote*. In that case, the agent weighs its own likelihood of correctness (private information) against the implied likelihood of correctness expressed by the other voters (public information).

For example, suppose that an agent is pivotal, and suppose it votes correctly with a likelihood of 0.75. Suppose there are nine voters. The ninth voter, given his or her priors, would have expected six voters to vote correctly and two to vote incorrectly.

Instead, four have voted correctly, and four incorrectly. What are the odds of this?

Using the formula provided in section 3.1, the odds are $0.75^4 (0.25)^4 C(8,4) = 0.087$. This provides the pivotal agent with additional information: it is quite unlikely that other agents hold private likelihoods of correctness of 0.75. (The probability of this is only 0.087.) It is reasonable—even if incorrect—to conclude that the observed voting pattern is representative, and thus that other agents hold likelihoods of correctness lower than 0.75. A rational agent would react to this additional information by revising its opinion, thus decreasing its own likelihood of correctness (Austen-Smith and Banks 1996).

The following code explores such a situation. Each agent votes correctly with a likelihood of correctness of 0.75. The model instantiates nine voters and holds a majority vote. Two strategies are explored: sincere voting and pivotal voting.

```
import model, likelihoods, confidences, strategies
print ("strategy confidence pct_correct")
for s, strategy in enumerate ([strategies.null,
strategies.pivotal]):
    for c in range (101):
        print (s, c / 100, model.percent (spec={
            "likelihood" : lambda a: likelihoods.constant (a, 0.75),
            "confidence" : lambda a: confidences.constant (a, c / 100),
            "strategy"    : strategy
        }, count=9, runs=1000))
```

Figure 3.21 displays the results. Under sincere voting, the majority vote is correct about 95% of the time. Under pivotal voting, lower-confidence pivotal agents conclude that their likelihoods must be too high, and revise them downwards. The likelihood of a correct vote decreases when low-confidence agents are pivotal. (High-confidence agents

ignore the possibility that they might be wrong, and resist downward revisions to their likelihoods of correctness.)

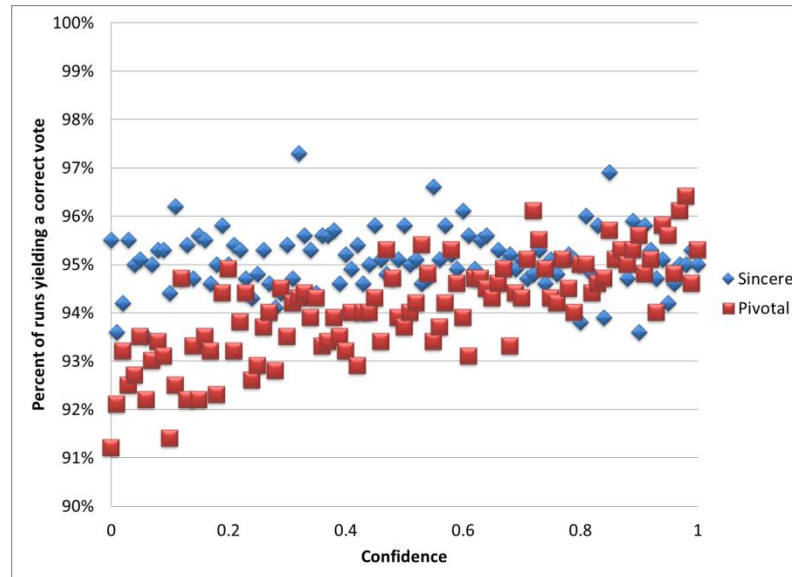


Figure 3.21: Likelihood of a correct vote by confidence: sincere vs. pivotal

As might be expected, the results become more pronounced as the number of voters shrinks. Consider a three-judge panel, each of whose members votes correctly with a likelihood of 0.75. Such a panel should vote correctly about 85% of the time. But when the third judge is pivotal, and considers the (split) opinions of the other two judges, the likelihood of a correct vote falls to around 75%. Figure 3.22 depicts this situation.

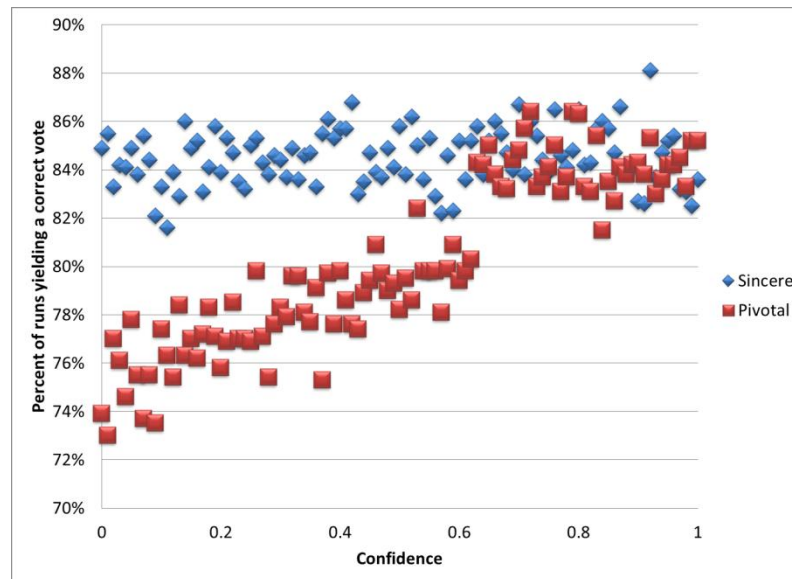


Figure 3.22: Likelihood of a correct vote by confidence: three judges

A pivotal voter, then, can be misled by the consensus. By paying attention to the consensus, a pivotal voter can reduce the likelihood of a correct vote. This situation is especially dangerous in the context of a jury trial requiring a unanimous verdict, as Feddersen and Pesendorfer (1998) observe. Consider the previous example of a three-judge panel, and suppose that a unanimous “guilty” verdict is required to convict. Suppose each judge votes correctly with likelihood 0.75; suppose also that the defendant is not guilty. The likelihood of the first two judges voting guilty is 0.0625. If this occurs, and if the third judge votes as a pivotal voter rather than a sincere voter, she will adjust her likelihood of correctness downward. The following code models this situation:


```

import model, likelihoods, confidences, strategies
print ("strategy confidence pct_correct")
for s, strategy in enumerate ([strategies.null,
strategies.pivotal]):
    for c in range (101):
        print (s, c / 100, model.percent (spec={
            "likelihood" : lambda a: likelihoods.constant (a, 0.75),
            "confidence" : lambda a: confidences.constant (a, c/100),
            "strategy" : strategy
        }, count=3, runs=1000, majority=0))

```

Figure 3.23 illustrates the results. When the pivotal voter defers to the consensus, the last chance for a (correct) “not guilty” verdict is lost. For low-confidence pivotal judges, the likelihood of a correct verdict falls from over 98% to around 95%. Feddersen and Pesendorfer (1998) use this fact to argue against the unanimity principle: a system which places the final voter in this pivotal position encourages strategic voting, which can lead to undesirable outcomes.

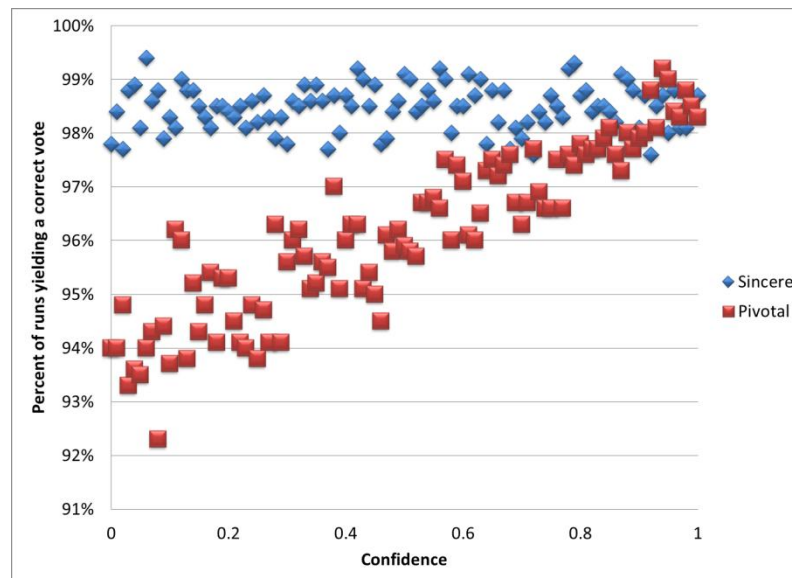


Figure 3.23: Likelihood of a correct vote by confidence: unanimity

Another variation on pivotal voting is described by Mueller (2003, 130). Mueller notes that if voting proceeded sequentially, and every voter took into account all of the preceding votes, the probability of a correct vote “is no greater than that of any single juror’s being correct.” The following code uses the `predominant()` strategy to test this claim, using a jury of nine voters:

```
import model, likelihoods, confidences, strategies
print ("strategy confidence pct_correct")
for s, strategy in enumerate (strategies.all_strategies):
    for c in range (101):
        print (s, c / 100, model.percent (spec={
            "likelihood" : lambda a: likelihoods.constant (a, 0.75),
            "confidence" : lambda a: confidences.constant (a, c / 100),
            "strategy"    : strategy
        }, count=9, runs=1000))
```

Each agent’s individual likelihood of correctness is 0.75. As Figure 3.24 shows, when voters vote sincerely, the cumulative likelihood of a correct vote is around 95%. But when voters vote sequentially, with each voter influenced by all previous voters, the result stays near 75%. Only as confidence increases, and voters stop listening to previous voters, does the result begin to rise. As Mueller observes, the excessive deference that arises in predominant voting completely counteracts Condorcet’s theorem.

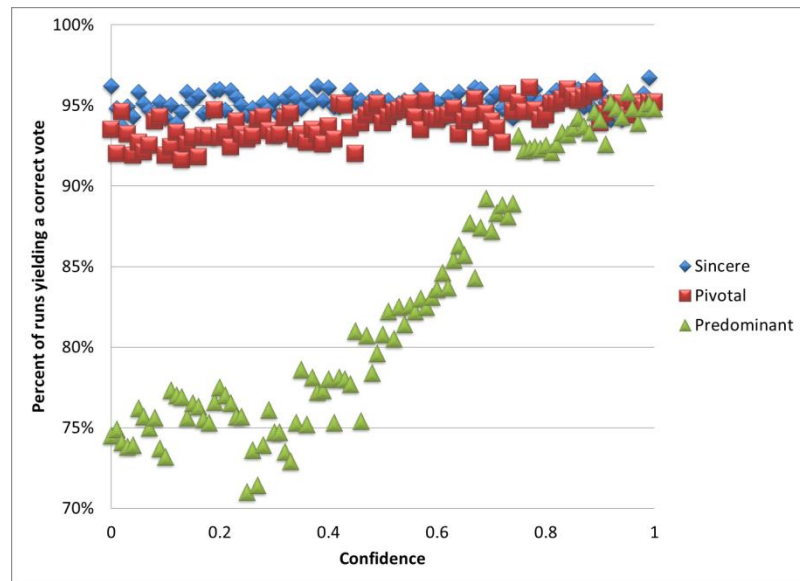


Figure 3.24: Likelihood of a correct vote by confidence: pivotal vs. predominant

The pivotal and predominant strategies have, so far, been explored with constant-likelihood voters. How do these voting strategies fare with real-world agents? The following code examines this question, applying the sincere, pivotal, and predominant strategies to a jury of nine agents defined by the GSS specification:

```
import gss, model, effects, strategies
print ("strategy complexity pct_correct")
for s, strategy in enumerate (strategies.all_strategies):
    for complexity in range (0, 101):
        print (s, complexity / 100, model.percent (spec={
            "factory"      : gss.factory,
            "likelihood"    : gss.complexity (complexity / 100),
            "confidence"    : gss.wise,
            "network"       : gss.network(),
            "effect"        : effects.mean_reversion,
            "strategy"      : strategy
        }, count=9, runs=1000))
```

As Figure 3.25 shows, real-world agents are sufficiently distinctive that the voting strategy is immaterial. As complexity increases, all three strategies transition at about the

same rate. Relaxing the sincerity assumption may work against Condorcet's theorem, but relaxing the homogeneity assumption works in favor of it.

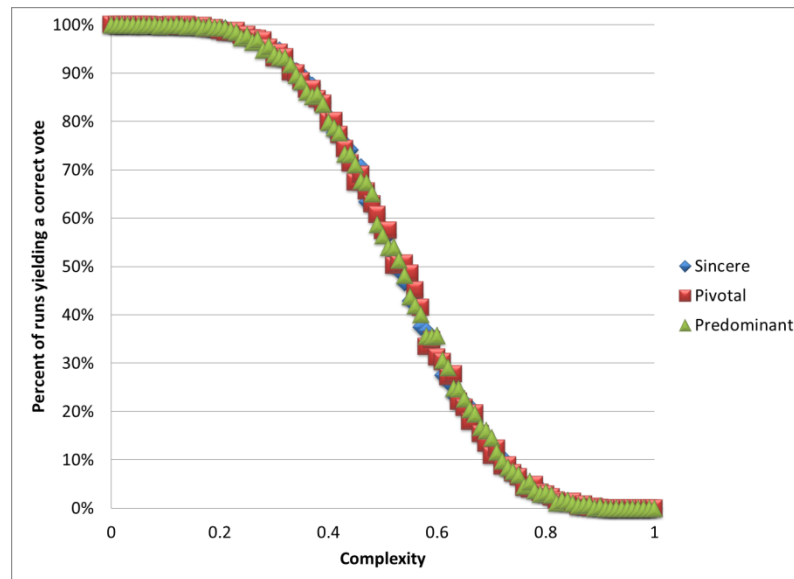


Figure 3.25: Likelihood of a correct vote by complexity: GSS agents

In a jury context, then, insincere voting does not undermine the likelihood of a correct vote, when agents are constructed from a population similar to that of the United States.

3.5. Policy

The preceding sections have assembled a voting model and calibrated it to the United States. This section shows how the model might be used to consider the impact of specific policies on voting outcomes.

It should be noted at the outset that any calibrated agent-based model is subject to the Lucas Critique. Calibrating a model in a policy-invariant way is difficult, if not

impossible. Nevertheless, there is insight to be gained from exploring such a model. (One of the great benefits of the agent-based approach is that if a more structurally sound specification is discovered, it can easily replace a prior specification.) The GSS specification includes all of the features described in earlier sections: agents are heterogeneous, exhibit convex confidences, belong to social networks derived from their characteristics, are susceptible to influence from their social network, and vote sincerely. Each of these features interact: for example, an increase in education affects not only an agent's likelihood of correctness, but also its confidence and social network. Votes are assumed to involve issues with a complexity rating of 0.5, where 0.0 means "as complex as the GSS science questions" and 1.0 means infinitely complex. Shifting this margin may, of course, shift the policy results.

3.5.1. Partisanship

Would less partisanship yield better outcomes? The following code compares three scenarios: the status quo, an electorate 50% less partisan, and an electorate 50% more partisan.

```
import gss, model
print (gss.percent ())
print (gss.percent (model.compress ("repub", 0.5)))
print (gss.percent (model.expand ("repub", 0.5)))
```

The status quo yields a correctness rate of about 50%. When partisanship decreases by half, pushing voters from the extremes toward the middle, the correctness rate stays about the same. When partisanship increases by half, the correctness rate increases slightly, but

not significantly. According to the model, there is no evidence that less partisanship would yield better outcomes at the specified margin.

3.5.2. Education: general

Would more education yield better outcomes? The following code compares three scenarios: the status quo, an electorate 25% less educated, and an electorate 25% more educated.

```
import gss, model
print (gss.percent ())
print (gss.percent (model.increase ("educ", 0.25)))
print (gss.percent (model.decrease ("educ", 0.25)))
```

The status quo yields a correctness rate of about 50%. At the specified margin, neither an increase nor a decrease in education alters the correctness rate. However, when the effect is raised from 0.25 to 0.75, both increases and decreases in education *lower* the correctness rate. An increase lowers the correctness rate to 47%, while a decrease lowers the correctness rate to 43%. This curious result stems from the use of education as a proxy for wisdom, coupled with the convex confidence function: in some cases, the increase in likelihood of correctness is more than offset by a decrease in confidence. Upon reflection, however, the reader may not consider this result all that counterintuitive: education often serves to teach the student how little he knows.

3.5.3. Education: minorities

Would more education yield better outcomes if it were targeted toward minority groups?

The following code extends the previous example by restricting it to minority voters:

```
import gss, model
def minority (agent):
    return agent["black"] or agent["hisp"]
print (gss.percent (model.increase ("educ", 0.25, minority)))
print (gss.percent (model.decrease ("educ", 0.25, minority)))
```

Unlike the previous case, educated targeted toward minority groups does have a slight effect, all else equal. Increases in education increase the correctness rate by about 1 percentage point, while decreases in education lower the correctness rate by the same amount. The same result is observed when the effect is changed from 0.25 to 0.75.

3.5.4. Income inequality: general

Would less income inequality yield better outcomes? The following specification examines four options: an overall increase in income, an overall decrease in income, a compression of incomes toward the median, and an expansion of incomes away from the median.

```
import gss, model
print (gss.percent (model.increase ("inc", 0.25)))
print (gss.percent (model.decrease ("inc", 0.25)))
print (gss.percent (model.compress ("inc", 0.25)))
print (gss.percent (model.expand ("inc", 0.25)))
```

Neither an increase nor a decrease in income affects the correctness rate at the specified margin. A compression of incomes leads to a slight decrease in the correctness rate,

perhaps because income is highly correlated with likelihood of correctness (and uncorrelated with anything else). An expansion of incomes does not affect the correctness rate. Although the Lucas Critique is surely relevant here, it must be concluded that the model presents no evidence that a decrease in income inequality would improve voting outcomes, relative to the status quo.

3.5.5. Income inequality: women

Would a change in income become relevant if it accrued mainly to women?

```
import gss, model
def female (agent):
    return not agent["male"]
print (gss.percent (model.increase ("inc", 0.25, female)))
print (gss.percent (model.decrease ("inc", 0.25, female)))
```

According to the model, the answer is no: changes in income do not translate to changes in the correctness rate.

3.5.6. Income inequality: minorities

Would a change in income become relevant if it accrued mainly to minorities?

```
import gss, model
def minority (agent):
    return agent["black"] or agent["hisp"]
print (gss.percent ())
print (gss.percent (model.increase ("inc", 0.25, minority)))
print (gss.percent (model.decrease ("inc", 0.25, minority)))
```

Once again, the answer is no.

3.6. Conclusion

At its heart, the Condorcet Jury Theorem is a defense of majority rule.

Foreshadowing Buchanan and Tullock by nearly two centuries, Condorcet recognized that the unanimity principle was simply not efficient enough for the modern world. Yet he recognized that imposing decisions by force, on unwilling subjects, was not something to be done lightly. Condorcet found solace in his famous theorem. So long as the average voter was at least slightly more than 50% likely to be correct, the majority could, with a clear conscience, compel the minority.

This chapter has presented an agent-based model of Condorcet's theorem. It demonstrated that Condorcet's basic results are easily reproduced. It then relaxed successively the three assumptions underlying Condorcet's theorem: homogeneity, independence, and sincerity. In each case, the model showed that it is possible to violate the assumption in such a way as to maintain Condorcet's result; but it is also possible to violate the assumption such that the result is undermined. At least in part, then, the assumptions underlying Condorcet's theorem remain relevant. The model can be calibrated to a real-world population. The General Social Survey was used to calibrate the model to a population similar to that of the United States. The result is a model which captures key features of voting in the United States, and can be used to explore the effects of policies which shift those features.

Political scientists may wish to use the model to assess the effects of proposed policy changes on voting outcomes. A key advantage of the agent-based approach is that it can be selectively upgraded; a researcher who wishes to make different assumptions

regarding voter behavior, or who wishes to use a different source to calibrate the model, may do so with minimal effort. This chapter has demonstrated the usefulness of the agent-based approach in public choice.

APPENDICES

1. supply-demand.py

```
#
# define number of buyers, sellers, periods, and iterations
#

B = 1000
S = 50
P = 200
I = 10

#
# consider  $Q_s = a + bP$ ,  $Q_d = c - dP$ ... give some
# concrete values for a, b, c, and d
#

a = 100
b = 10
c = 20
d = 7

#
# define a demand shock (a change to the value of c)
#

demand_shock = -3

#
# define the Walrasian auctioneer
#

class auctioneer:

    # calculate the Walrasian equilibrium price
    def calculate (self, buyers, sellers):

        # begin with price zero and change it incrementally
        price = 0.0
        increment = 100.0

        # see whether the equilibrium price is positive or negative
        supply = sum (map (lambda s: s.quantity (price), sellers))
        demand = sum (map (lambda b: b.quantity (price), buyers))
        positive = (supply < demand)
```

```

# discover the equilibrium price
while abs (supply - demand) > 0.0000001:
    if supply < demand:
        price += increment
        if not positive:
            increment /= 2
    else:
        price -= increment
        if positive:
            increment /= 2
    supply = sum (map (lambda s: s.quantity (price), sellers))
    demand = sum (map (lambda b: b.quantity (price), buyers))

# save the equilibrium price for later
self.equilibrium = price

# return the Walrasian equilibrium price
def price (self):
    return self.equilibrium

#
# create a Walrasian auctioneer
#

walras = auctioneer()

#
# run a model
#

def run (name, model):

    # execute the model multiple times
    results = [model() for i in range (I)]

    # initialize totals
    units = [0 for p in range (P)]
    sales = [0 for p in range (P)]
    inventory = [0 for p in range (P)]

    # calculate totals
    for r in results:
        for (p,u,s,i) in r:
            units[p] += u
            sales[p] += s
            inventory[p] += i

    # calculate averages
    for p in range (P):
        units[p] /= I
        sales[p] /= I
        inventory[p] /= I

    # create the results file
    file = open (name + ".csv", "w")

    # write the header
    file.write ("Period,Volume,AvgPrice,Inventory\n")

    # write results
    for p in range (P):
        file.write ("%d,%f,%f,%f\n" %
            (p, units[p], sales[p] / units[p], inventory[p]))

```

```

# close the output file
file.close()

#
# define the basic model
#

def model (buyer_factory, seller_factory):

    # we haven't seen any results yet
    results = []

    # create the buyers and sellers
    buyers = [buyer_factory() for b in range (B)]
    sellers = [seller_factory() for s in range (S)]

    # for each period...
    for period in range (P):

        # introduce a demand shock
        if period == int (P / 4):
            for b in buyers:
                b.shock (demand_shock)

        # allow the auctioneer to determine the equilibrium price
        walras.calculate (buyers, sellers)

        # engage in production
        for s in sellers:
            s.produce()

        # initialize running totals
        units = 0
        sales = 0

        # for each buyer...
        for b in buyers:

            # identify the seller offering the best deal
            quantity = 0
            for s in sellers:
                if min (b.quantity (s.price()), s.inventory) > quantity:
                    quantity = min (b.quantity (s.price()), s.inventory)
                    seller = s

            # buy from this seller
            if quantity:
                seller.sell (quantity)
                units += quantity
                sales += quantity * seller.price()

        # determine end-of-period inventory
        inventory = sum (map (lambda s: s.inventory, sellers))

        # save this period's results
        results.append ((period, units, sales, inventory))

    # return the results to the caller
    return results

```

```

#
# homogeneous buyers
#

class homogeneous_buyer:

    # initialize the buyer
    def __init__ (self):
        self.shift = 0

    # define demand quantity as a function of price
    def quantity (self, price):
        return max (c + self.shift - d * price, 0)

    # introduce a demand shock
    def shock (self, shock):
        self.shift = shock

#
# price-taking seller
#

class walrasian_seller:

    # initialize inventory to zero
    def __init__ (self):
        self.inventory = 0

    # define supply quantity as a function of price
    def quantity (self, price):
        return max (a + b * price, 0)

    # this seller offers the Walrasian price
    def price (self):
        return walras.price()

    # produce goods
    def produce (self):
        quantity = self.quantity (self.price())
        if self.inventory < quantity:
            self.inventory += (quantity - self.inventory)

    # sell goods
    def sell (self, quantity):
        self.inventory -= quantity

#
# learning seller
#

class learning_seller:

    # initialize inventory to zero and clear sales history
    def __init__ (self):
        self.inventory = 0
        self.prevsales = 0
        self.sales = 0

    # define supply quantity as a function of price
    def quantity (self, price):
        return max (a + b * price, 0)

```

```

# offer the Walrasian price initially and then use history
def price (self):
    return self.nextprice

# produce goods
def produce (self):
    if self.sales > 0 and self.prevsales > 0:
        self.nextprice *= (1.0 + (self.sales-self.prevsales) / self.prevsales)
    elif self.sales == 0 and self.prevsales > 0:
        self.nextprice *= 0.5
    else:
        self.nextprice = walras.price()
    quantity = self.quantity (self.nextprice)
    if self.inventory < quantity:
        self.inventory += (quantity - self.inventory)
    self.prevsales = self.sales
    self.sales = 0

# sell goods
def sell (self, quantity):
    self.inventory -= quantity
    self.sales += quantity

#
# invoke the models
#

run ("classical",
    lambda: model (
        lambda: homogeneous_buyer(),
        lambda: walrasian_seller()))

run ("learning",
    lambda: model (
        lambda: homogeneous_buyer(),
        lambda: learning_seller()))

# end of supply-demand.py

```

2.A. model.py

```

#
# model.py:  an agent-based model of regulation
#

import random

#
# List the statistics to be calculated by the model.
#

stats = [
    "received",
    "sent",
    "organize",
    "regulator",
    "transfers",
    "taxrates",
    "actual",
    "potential",

```

```

    "pctvotes",
    "deadweight",
    "return"
]

#
# The transfer class stores information about an intergroup
# transfer. Each transfer is characterized by an amount
# received from another group, the amount spent to convince
# that group to undertake the transfer (e.g., "education"),
# the amount spent to organize the transfer, and the amount
# spent to convince the regulator to undertake the transfer.
#

class transfer:

    #
    # Define a transfer.
    #

    def __init__ (
        self,
        group,
        received,
        sent,
        organize,
        regulator):

        # remember the other group's identity
        self.group = group

        # remember the amount received from this group
        self.received = received

        # remember the amount sent to this group
        self.sent = sent

        # remember the amount spent to organize the transfer
        self.organize = organize

        # remember the amount sent to the regulator
        self.regulator = regulator

#
# The group class defines a party to a transfer. When a group
# receives a transfer, its members benefit; when a group sends
# a transfer, its members suffer.
#

class group:

    #
    # Define a group.
    #

    def __init__ (
        self,
        name,
        factory,
        citizens,
        transfers):

```



```

# remember the group name
self.name = name

# create the group's citizens
self.citizens = [factory (self) for c in range (citizens)]

# remember the number of citizens in the group
self.members = len (self.citizens)

# we haven't processed any transfers yet
self.received_from = {}
self.sent_to = {}
self.organize = 0
self.regulator = 0

# for each transfer...
for t in transfers:

    # remember the amount received from this group
    self.received_from[t.group] = t.received

    # remember the amount sent to this group
    self.sent_to[t.group] = t.sent

    # remember the amount spent to organize the transfer
    self.organize += t.organize

    # remember the amount sent to the regulator
    self.regulator += t.regulator

#
# Initialize a group. (This cannot be done in the
# constructor, because we need access to *all* groups
# to calculate some group characteristics.) The
# decplaces argument defines the number of
# decimal places in the tax rate.
#

def initialize (self, groups, decplaces=2):

    # initialize group variables
    self.received = 0
    self.sent = 0
    self.persuasion = 0
    self.taxburden = 0

    # remember the transfers received by this group
    for g in self.received_from.keys():
        self.received += self.received_from[g]

    # remember the amount spent to persuade other groups
    for g in self.sent_to.keys():
        self.sent += self.sent_to[g]

    # calculate this group's net transfers
    self.transfers \
    = self.received \
    - self.sent \
    - self.organize \
    - self.regulator

    # remember the amount received to be persuaded
    for g in groups:
        if self.name in g.sent_to:
            self.persuasion += g.sent_to[self.name]

```

```

# determine this group's total tax burden
for g in groups:
    if self.name in g.received_from:
        self.taxburden += g.received_from[self.name]

# if we have no tax burden...
if self.taxburden == 0:

    # we have no taxes
    self.taxrate = 0

# if we have a tax burden...
else:

    # determine the tax rate resolution
    resolution = 10 ** decplaces

    # for each possible tax rate...
    for tax in range (resolution + 1):

        # we haven't figured out the revenues yet
        raised = 0

        # calculate revenues raised at this tax rate
        for c in self.citizens:
            raised += (tax / resolution) * c.income (tax / resolution)

        # if we've raised enough revenue, stop looping
        if raised > 0.99 * self.taxburden:
            break

    # remember this group's tax rate
    self.taxrate = tax / resolution

#
# The citizen class defines each agent.  Each
# citizen belongs to exactly one group.
#

class citizen:

    #
    # Define a citizen.
    #

    def __init__ (
        self,
        group,
        potential,
        deadweight,
        response,
        approval,
        persuasion):

        # remember this citizen's group
        self.group = group

        # remember this citizen's potential income
        self.potential = potential

        # remember the degree to which taxes discourage work
        self.deadweight = deadweight

```

```

    # remember the response accompanying a change in income
    self.response = response

    # remember this citizen's default approval of the regime
    self.approval = approval

    # remember the influence of persuasion
    self.persuasion = persuasion

#
# Determine the citizen's income for a given tax rate.
#

def income (self, taxrate):
    return self.potential * (1 - taxrate) ** self.deadweight

#
# Determine the citizen's actual income.
#

def actual (self):
    return \
        self.income (self.group.taxrate) + \
        self.group.transfers / self.group.members

#
# Determine the citizen's likelihood of voting for
# the current regulatory regime.
#

def vote (self):
    return \
        self.approval * \
        (self.actual() / self.potential) ** self.response + \
        self.group.persuasion / self.group.members / self.persuasion

#
# Execute the model.
#

def run (regime, ids=[], runs=10):

    # we haven't seen any totals yet
    totals = {}

    # for each group...
    for (n,f,a,t) in regime:

        # we haven't created any totals yet
        totals[n] = {}

        # initialize totals to zero
        for s in stats:
            totals[n][s] = 0

    # for each run...
    for r in range (runs):

        # create groups
        groups = [group (n,f,a,t) for (n,f,a,t) in regime]

```

```

# initialize groups
for g in groups:
    g.initialize (groups)

# for each group...
for g in groups:

    # find this group's totals
    total = totals[g.name]

    # update group totals
    total["received"] += g.received
    total["sent"] += g.sent
    total["organize"] += g.organize
    total["regulator"] += g.regulator
    total["transfers"] += g.transfers
    total["taxrates"] += g.taxrate

    # clear citizen totals
    actual = 0
    potential = 0
    deadweight = 0
    votes = 0

    # for each citizen in this group...
    for c in g.citizens:

        # update actual and potential income
        actual += c.actual()
        potential += c.potential

        # calculate deadweight loss
        deadweight += c.potential * (1 - c.group.taxrate) \
            - c.income (c.group.taxrate)

        # ask this citizen to vote
        votes += (c.vote() > random.random())

    # update the citizen totals
    total["actual"] += actual
    total["potential"] += potential
    total["deadweight"] += deadweight
    total["pctvotes"] += votes / g.members

    # calculate total group outflows
    outflows = g.sent + g.organize + g.regulator

    # calculate group's rate of return
    total["return"] += g.received / outflows if outflows else 0

# calculate per-run averages
for (n,f,a,t) in regime:
    for s in stats:
        totals[n][s] /= runs

# clear the group totals
agents = 0
groups = {}
weighted = {}

# initialize group totals to zero
for s in stats:
    groups[s] = 0
    weighted[s] = 0

```

```

# calculate group totals
for (n,f,a,t) in regime:
    for s in stats:
        groups[s] += totals[n][s]
        weighted[s] += totals[n][s] * a

# determine the total number of agents
for (n,f,a,t) in regime:
    agents += a

# calculate the weighted averages
for s in ["taxrates", "pctvotes", "return"]:
    groups[s] = weighted[s] / agents

# allow intermediate output to be suppressed easily
if False:

    # for each group...
    for (n,f,a,t) in regime:

        # print the group name
        print (n, end="\t")

        # print the ID columns
        for i in ids:
            print (i, end="\t")

        # print the statistics
        for s in stats:
            print (totals[n][s], end="\t")

        # print a newline
        print()

    # print the total name
    print ("total", end="\t")

    # print the ID columns
    for i in ids:
        print (i, end="\t")

    # print the statistics
    for s in stats:
        print (groups[s], end="\t")

    # print a newline
    print()

#
# Print the column headings.
#

def headings (ids=[]):

    # print the group column heading
    print ("group", end="\t")

    # print the ID column names
    for i in ids:
        print (i, end="\t")

    # print the statistic column names
    for s in stats:
        print (s, end="\t")

```

```

    # print a newline
    print()

# end of model.py

```

2.B. license.py

```

#
# license.py: model a licensing regime
#

import random
import model

#
# Define a licensing regime.
#

def regime (
    citizens      = 10000,
    percent       = 0.01,
    received      = 1000,
    sent          = 100,
    organize      = 50,
    regulator     = 0,
    potential     = 15000,
    potential_sig  = 0.0,
    approval      = 0.5,
    approval_sd   = 0.0,
    deadweight    = 1.0,
    deadweight_sig = 0.0,
    response      = 1.0,
    response_sig  = 0.0,
    persuasion    = 1000,
    persuasion_sd = 0.0):

    # determine the number of holders and nonholders
    holders = int (citizens * percent)
    nonholders = citizens - holders

    # define a transfer from nonholders to holders
    license = model.transfer (
        "nonholders",
        received * holders,
        sent * holders,
        organize * holders,
        regulator * holders
    )

    # define a factory function for citizens
    citizen = lambda group: model.citizen (
        group,
        random.lognormvariate (0, potential_sig) * potential,
        random.lognormvariate (0, deadweight_sig) * (deadweight - 1) + 1,
        random.lognormvariate (0, response_sig) * (response - 1) + 1,
        random.gauss (approval, approval_sd),
        random.gauss (persuasion, persuasion_sd)
    )

```

```

# return the regulatory regime
return [
    ("holders", citizen, holders, [license]),
    ("nonholders", citizen, nonholders, [])
]

# end of license.py

```

2.C. subsidy.py

```

#
# subsidy.py: model a cross-subsidy regime
#

import random
import model

#
# Define a cross-subsidy regime.
#

def regime (
    citizens      = 10000,
    percent       = (0.01, 0.1),
    received      = (1000, 200),
    sent          = (100, 0),
    organize      = (50, 0),
    regulator     = (0, 0),
    potential     = (15000, 15000),
    potential_sig  = (0.0, 0.0),
    approval      = (0.5, 0.5),
    approval_sd   = (0.0, 0.0),
    deadweight    = (1.0, 1.0),
    deadweight_sig = (0.0, 0.0),
    response      = (2.0, 2.0),
    response_sig  = (0.0, 0.0),
    persuasion    = (1000, 1000),
    persuasion_sd  = (0.0, 0.0)):

    # determine the number of direct subsidy recipients
    direct_count = int (citizens * percent[0])

    # determine the number of cross-subsidy recipients
    cross_count = int (citizens * percent[1])

    # determine the number of subsidizers
    public_count = citizens - direct_count - cross_count

    # define the direct subsidy
    direct = model.transfer (
        "public",
        received[0] * direct_count,
        sent[0] * direct_count,
        organize[0] * direct_count,
        regulator[0] * direct_count
    )

```

```

# define the cross-subsidy
cross = model.transfer (
    "public",
    received[1] * cross_count,
    sent[1] * cross_count,
    organize[1] * cross_count,
    regulator[1] * cross_count
)

# define a factory function for non-cross-subsidized citizens
citizen = lambda group: model.citizen (
    group,
    random.lognormvariate (0, potential_sig[0]) * potential[0],
    random.lognormvariate (0, deadweight_sig[0]) * (deadweight[0] - 1) + 1,
    random.lognormvariate (0, response_sig[0]) * (response[0] - 1) + 1,
    random.gauss (approval[0], approval_sd[0]),
    random.gauss (persuasion[0], persuasion_sd[0])
)

# define a factory function for cross-subsidized citizens
special = lambda group: model.citizen (
    group,
    random.lognormvariate (0, potential_sig[1]) * potential[1],
    random.lognormvariate (0, deadweight_sig[1]) * (deadweight[1] - 1) + 1,
    random.lognormvariate (0, response_sig[1]) * (response[1] - 1) + 1,
    random.gauss (approval[1], approval_sd[1]),
    random.gauss (persuasion[1], persuasion_sd[1])
)

# return the regulatory regime
return [
    ("direct", citizen, direct_count, [direct]),
    ("cross", special, cross_count, [cross]),
    ("public", citizen, public_count, [])
]

# end of subsidy.py

```

2.D. bidirectional.py

```

#
# bidirectional.py: model a regime of bidirectional transfers
#

import random
import model

```



```

#
# Define a bidirectional transfer regime.
#

def regime (
    citizens      = 10000,
    percent       = 0.60,
    received      = (1000, 750),
    sent          = (100, 50),
    organize      = (100, 100),
    regulator     = (0, 0),
    potential     = (15000, 15000),
    potential_sig  = (0.0, 0.0),
    approval      = (0.5, 0.5),
    approval_sd   = (0.0, 0.0),
    deadweight    = (1.0, 1.0),
    deadweight_sig = (0.0, 0.0),
    response      = (1.0, 1.0),
    response_sig  = (0.0, 0.0),
    persuasion    = (1000, 1000),
    persuasion_sd = (0.0, 0.0)):

    # determine the size of the first group
    first_count = int (citizens * percent)

    # determine the size of the second group
    second_count = citizens - first_count

    # define the subsidy from second to first
    first = model.transfer (
        "second",
        received[0] * first_count,
        sent[0] * first_count,
        organize[0] * first_count,
        regulator[0] * first_count
    )

    # define the subsidy from first to second
    second = model.transfer (
        "first",
        received[1] * second_count,
        sent[1] * second_count,
        organize[1] * second_count,
        regulator[1] * second_count
    )

    # define a factory function for the first group
    first_citizen = lambda group: model.citizen (
        group,
        random.lognormvariate (0, potential_sig[0]) * potential[0],
        random.lognormvariate (0, deadweight_sig[0]) * (deadweight[0] - 1) + 1,
        random.lognormvariate (0, response_sig[0]) * (response[0] - 1) + 1,
        random.gauss (approval[0], approval_sd[0]),
        random.gauss (persuasion[0], persuasion_sd[0])
    )

    # define a factory function for the second group
    second_citizen = lambda group: model.citizen (
        group,
        random.lognormvariate (0, potential_sig[1]) * potential[1],
        random.lognormvariate (0, deadweight_sig[1]) * (deadweight[1] - 1) + 1,
        random.lognormvariate (0, response_sig[1]) * (response[1] - 1) + 1,
        random.gauss (approval[1], approval_sd[1]),
        random.gauss (persuasion[1], persuasion_sd[1])
    )

```

```

# return the regulatory regime
return [
    ("first", first_citizen, first_count, [first]),
    ("second", second_citizen, second_count, [second])
]

# end of bidirectional.py

```

3.A.1. model.py

```

#
# model.py
#

import factories
import likelihoods
import confidences
import networks
import effects
import strategies

#
# Determine whether a vote has passed.
#

def passed (percent, majority):
    return (
        majority == 1 and percent == 1 or
        majority < 1 and percent > majority)

#
# This is the model driver. Its first argument is the model
# specification, which may contain the following elements:
#
#   factory      : a function that creates agents
#   likelihood   : a function that computes likelihood of correctness
#   confidence   : a function that computes degree of confidence
#   network      : a function that assigns agents to networks
#   effect       : a function that applies network effects
#   strategy     : a function that implements a voting strategy
#
# The model instantiates agents with characteristics defined
# by the supplied agent factory. It assigns each agent to
# a network. If a policy was supplied, it uses the policy
# to modify agent characteristics. The model then determines
# the likelihood that an agent would vote correctly if voting
# independently. It adjusts these likelihoods based on each
# agent's social network, then holds an actual vote. If a
# majority of votes are correct, the vote passes; otherwise it
# fails. The process is repeated a specified number of times.
# The model returns the percent of runs that passed.
#

def percent (spec={}, policy=None, count=1001, runs=100, majority=0.5, avg=False):

```

```

# we haven't examined the model specification yet
factory      = factories.null
likelihood    = likelihoods.null
confidence    = confidences.null
network       = networks.null
effect        = effects.null
strategy      = strategies.null

# if an agent factory was specified, remember it
if "factory" in spec:
    factory = spec["factory"]

# if a likelihood function was specified, remember it
if "likelihood" in spec:
    likelihood = spec["likelihood"]

# if a confidence function was specified, remember it
if "confidence" in spec:
    confidence = spec["confidence"]

# if a network topology was specified, remember it
if "network" in spec:
    network = spec["network"]

# if a network effect was specified, remember it
if "effect" in spec:
    effect = spec["effect"]

# if a voting strategy was specified, remember it
if "strategy" in spec:
    strategy = spec["strategy"]

# ensure the number of agents is odd
if count % 2 == 0:
    count += 1

# we haven't seen any correct runs yet
correct = 0

# for each run...
for r in range (runs):

    # instantiate the agents
    agents = factory (count)

    # if a policy was provided...
    if policy:

        # for each agent...
        for agent in agents:

            # apply the policy
            policy (agent)

    # determine each agent's independent likelihood of correctness
    agent_likelihoods = likelihood (agents)

    # confirm that each agent has been assigned a likelihood
    assert len (agent_likelihoods) == len (agents)

    # determine each agent's confidence
    agent_confidences = confidence (agents)

    # confirm that each agent has been assigned a confidence
    assert len (agent_confidences) == len (agents)

```

```

# assign each agent to a network
agent_networks = network (agents)

# confirm that each agent has been assigned to a network
assert len (agent_networks) == len (agents)

# revise each agent's likelihood based on network effects
agent_revised = effect (list (zip (
    agent_networks, agent_likelihoods, agent_confidences)))

# confirm that each agent's likelihood has been revised
assert len (agent_revised) == len (agents)

# ask each agent to vote and record whether the vote was correct
result = strategy (list (zip (
    agent_revised, agent_confidences)), majority)

# if we should compute the average correctness...
if avg:

    # remember the percent of correct votes
    correct += result

# if we should compute the percent of passing runs...
else:

    # remember whether this run passed
    correct += passed (result, majority)

# return the percent of runs that were correct
return correct / runs

#
# Determine the degree of convergence exhibited
# by a model specification.
#
def convergence (spec={}, policy=None, epsilon=0.01, iterations=100, **etc):

    # run the model the specified number of times
    pcts = [percent (spec, policy, **etc) for i in range (iterations)]

    # determine the average result
    avg = sum (pcts) / iterations

    # count the number of values within epsilon of the average
    converged = sum (map (lambda p: abs ((p - avg) / avg) < epsilon, pcts))

    # return the average value and the degree of convergence
    return (avg, converged / iterations)

#
# Implement the increase policy:  increase agent attributes.
#
def increase (characteristic, degree, predicate=lambda a: True):

    # nested function:  compress the attribute
    def nested (agent):

        # if we should adjust this agent's value...
        if predicate (agent):

```

```

        # move it up toward the maximum
        agent[characteristic] += degree * (1 - agent[characteristic])

    # return the compression policy
    return lambda agent: nested (agent)

#
# Implement the decrease policy:  decrease agent attributes.
#

def decrease (characteristic, degree, predicate=lambda a: True):

    # nested function:  compress the attribute
    def nested (agent):

        # if we should adjust this agent's value...
        if predicate (agent):

            # move it up toward the maximum
            agent[characteristic] -= degree * agent[characteristic]

        # return the compression policy
        return lambda agent: nested (agent)

#
# Implement the compress policy:  compress agent attributes.
#

def compress (characteristic, degree, predicate=lambda a: True):

    # nested function:  compress the attribute
    def nested (agent):

        # if we should adjust this agent's value...
        if predicate (agent):

            # retrieve the characteristic
            c = agent[characteristic]

            # if the characteristic is below the median...
            if c < 0.5:

                # move it up toward the median
                agent[characteristic] += degree * (0.5 - c)

            # if the characteristic is at or above the median...
            else:

                # move it down toward the median
                agent[characteristic] -= degree * (c - 0.5)

        # return the compression policy
        return lambda agent: nested (agent)

#
# Implement the expand policy:  expand agent attributes.
#

def expand (characteristic, degree, predicate=lambda a: True):

    # nested function:  expand the attribute
    def nested (agent):

```

```

# if we should adjust this agent's value...
if predicate (agent):

    # retrieve the characteristic
    c = agent[characteristic]

    # if the characteristic is below the median...
    if c < 0.5:

        # move it down away from the median
        agent[characteristic] -= max (c, degree * (0.5 - c))

    # if the characteristic is at or above the median...
    else:

        # move it up away from the median
        agent[characteristic] += min (1-c, degree * (c - 0.5))

# return the expansion policy
return lambda agent: nested (agent)

# end of model.py

```

3.A.2. factories.py

```

#
# factories.py
#
# The model uses factories to create agents. The
# model extracts the factory function from the
# specification and calls it with a single argument:
# the number of agents to be created. The function
# returns a list of agents.
#

import random

#
# Define the null factory. Each agent is
# created with no characteristics.
#

def null (count):
    return [{ } for c in range (count)]

#
# Define a characteristic-based agent factory. Each
# characteristic is described by three values: its
# name, its distribution, and its correlation vector.
# The distribution is expressed as a function which
# returns a value between 0 and 1. The correlation
# vector is a list of values between -1 and 1 which
# express the correlation between the characteristic
# and all preceding characteristics (that is, the
# lower triangle of the correlation matrix).
#

```

```

# Each agent's characteristics are assigned by taking
# draws from the supplied distribution. After each
# characteristic is drawn, it is correlated with each
# of the preceding characteristics via a probabilistic
# sort. The preceding characteristic is sorted in
# ascending order, and the new characteristic is then
# sorted in the same (corr > 0) or opposite (corr < 0)
# order, with unsorted values being swapped with
# probability abs(corr). Thus, specifying corr = 0
# means that values are never swapped--there is no
# correlation. Specifying corr = 1 or -1 means that
# values are always swapped--there is full correlation,
# either positive or negative.
#

def characteristic (count, characteristics=[]):

    # begin by creating agents with no characteristics
    agents = [{} for c in range (count)]

    # for each supplied characteristic...
    for (characteristic, distribution, correlations) in characteristics:

        # for each agent...
        for agent in agents:

            # draw values for each agent from the distribution
            agent[characteristic] = distribution()

        # for each correlation...
        for c in range (len (correlations)):

            # determine the sign of the correlation measure
            sign = 1 if correlations[c] > 0 else -1

            # sort agents by the correlated characteristic
            agents.sort (key=lambda agent: agent[characteristics[c][0]])

            # for each agent (in order of related characteristic)...
            for a in range (len (agents)):

                # if we select this agent to participate in the correlation...
                if random.random() < abs (correlations[c]):

                    # for each subsequent agent...
                    for aa in range (a+1, len (agents)):

                        # if this value is (greater than, less than) the current value...
                        if sign * agents[a][characteristic] > \
                            sign * agents[aa][characteristic]:

                            # swap the values
                            tmp = agents[a][characteristic]
                            agents[a][characteristic] = agents[aa][characteristic]
                            agents[aa][characteristic] = tmp

            # return the agents
            return agents

#
# Determine the profile of a list of agents.
# Given a list of agents, return a list
# containing the percent of agents with each
# unique combination of characteristics.
#

```

```

def profile (agents):

    # we haven't processed any agents yet
    groups = []
    counts = []

    # for each agent...
    for agent in agents:

        # we haven't found this agent's group yet
        found = False

        # for each unique collection of characteristics...
        for n, group in enumerate (groups):

            # if this agent belongs to this group...
            if agent == group:

                # we've seen another agent in this group
                counts[n] += 1

                # remember we've found the group
                found = True

                # no need to keep looping
                break

        # if we didn't find this agent's group...
        if not found:

            # create a new group
            groups.append (agent)

            # we've seen one agent in this group
            counts.append (1)

    # determine the percent of agents in each group
    percents = map (lambda count: count / len (agents), counts)

    # return the agent groups and their counts
    return list (zip (groups, percents))

# end of factories.py

```

3.A.3. likelihoods.py

```

#
# likelihoods.py
#
# Each agent votes correctly with some likelihood.
# The model extracts the likelihood function from
# the specification and calls it with a single
# argument: a list of agents. The function returns
# a list of likelihoods, one for each agent. Thus,
# the length of the returned list of likelihoods is
# equal to the length of the supplied list of agents.
#

```



```

#
# Define the null likelihood.
#

def null (agents):
    return [0.5 for a in agents]

#
# Define a constant likelihood.
#

def constant (agents, value=0.5):
    return [value for a in agents]

#
# Define a distribution-based likelihood.
#

def distribution (agents, distribution=lambda a: 0.5):
    return [distribution() for a in agents]

#
# Define a characteristic-based likelihood.
#

def characteristic (agents, function=lambda a: 0.5):
    return list (map (function, agents))

# end of likelihoods.py

```

3.A.4. confidences.py

```

#
# confidences.py
#
# Each agent possesses a level of confidence in
# its likelihood of correctness. Confidence is
# measured from 0 to 1.
#

#
# Define the null confidence.
#

def null (agents):
    return [1 for a in agents]

#
# Define a constant confidence.
#

def constant (agents, value=1):
    return [value for a in agents]

```

```

#
# Define a distribution-based confidence.
#

def distribution (agents, distribution=lambda: 1):
    return [distribution() for a in agents]

#
# Define a characteristic-based confidence.
#

def characteristic (agents, function=lambda a: 1):
    return list (map (function, agents))

# end of confidences.py

```

3.A.5. networks.py

```

#
# networks.py
#
# The model allows agents to be assigned to social
# networks. The model extracts the network function
# from the specification and calls it with a single
# argument: a list of agents. The network function
# may, but is not required to, use information from
# each agent to construct the social network. The
# function returns a list of social networks, indexed
# by agent. Thus, the length of the returned list
# of network assignments is equal to the length of the
# supplied list of agents.
#

import random
import distributions

#
# Define the null network.
#

def null (agents):
    return [n for n in range (len (agents))]

#
# Define a grid network. The caller supplies a
# list of (distribution, groups) pairs. Distribute
# agents in space according to the supplied distributions,
# then group agents into networks based on adjacency.
#

def grid (agents, distributions=[]):

    # we haven't created any networks yet
    networks = []

    # for each agent...
    for agent in agents:

```

```

    # determine the agent's network
    network = [int (d() * g) for d,g in distributions]

    # save this agent's network
    networks.append (tuple (network))

    # return the networks to the caller
    return networks

#
# Define a uniform network.  Agents are
# distributed evenly into the specified
# number of networks.
#

def uniform (agents, groups):
    return grid (agents, [(random.random, groups)])

#
# Define a characteristic-based network.
#

def characteristic (agents, function=lambda a: 0):
    return list (map (function, agents))

#
# Define a partition network.  Partition each characteristic
# into some number of groups, then take the Cartesian product
# of those groups.  For example, a profile of the form
# {"C1" : 2, "C4" : 3} will create six social networks--
# the result of partitioning agents into two groups based
# on characteristic C1 (less than half, greater than or
# equal to half) and three groups based on characteristic
# C4 (less than a third, greater than or equal to a third
# and less than two thirds, greater than or equal to two thirds).
#

def partition (agents, profile={}):

    # extract characteristics from the profile
    characteristics = list (profile.keys())

    # extract dimensions from the profile
    dimensions = profile.values()

    # expand each dimension from d to [0,1,2,...,d-1]
    expanded = list (map (range, dimensions))

    # normalize each range from [0,1,...,d-1] to [1/d,2/d,...,1]
    ranges = list (map (
        lambda l: list (map (
            lambda n: (n+1) / len (l), 1)), expanded))

    # we haven't created any networks yet
    networks = []

    # define a subfunction to handle recursion
    def recurse (pos, current):

        # if we've exhausted the possible ranges...
        if pos == len (ranges):

```

```

    # add the current network to the list
    networks.append (current)

# if we haven't exhausted the possible ranges...
else:

    # for each value in this range...
    for r in ranges[pos]:

        # make a copy of the current network
        network = current[:]

        # add this value to the network
        network.append (r)

        # recurse to capture the next range
        recurse (pos+1, network)

# build the network definitions from the ranges
recurse (0, [])

# we haven't assigned agents to networks yet
results = []

# for each agent...
for agent in agents:

    # for each network...
    for n in range (len (networks)):

        # assume this agent is a member of this network
        member = True

        # for each characteristic in the profile...
        for c in range (len (characteristics)):

            # if this agent's value is too big for this network...
            if agent[characteristics[c]] > networks[n][c]:

                # this agent isn't a member of this network
                member = False

                # no need to check the other characteristics
                break

        # if this agent is a member of this network...
        if member:

            # remember the agent's network
            results.append (n)

            # no need to check other networks
            break

# return the networks to the caller
return results

# end of networks.py

```

3.A.6. effects.py

```
#
# effects.py
#
# Apply network effects. The model extracts the
# effects function from the specification and calls it
# with a single argument: a list of (network, likelihood,
# confidence) tuples, with one tuple per agent. The
# function groups likelihoods of related networks and
# computes an aggregate effect for each network, then
# applies the appropriate effect to each agent's likelihood.
# The function returns the revised likelihoods for each
# agent. Thus, the length of the returned list is
# equal to the length of the supplied list of
# (network, likelihood, confidence) tuples.
#

#
# Define the null effect.
#

def null (tuples):
    return [1 for (n,l,c) in tuples]

#
# Apply an aggregation operator to each network
# and recompute each agent's likelihood.
#

def operator (tuples, function=lambda l: 0):

    # we haven't seen any networks yet
    networks = {}

    # for each triple...
    for network, likelihood, confidence in tuples:

        # if we haven't seen this network before...
        if network not in networks:

            # create an empty list for this network
            networks[network] = []

            # add this likelihood to the network
            networks[network].append (
                likelihood * confidence + 0.5 * (1 - confidence))

    # we haven't processed any values yet
    values = {}

    # for each network...
    for network in networks.keys():

        # calculate the network's value
        values[network] = function (networks[network])

    # we haven't revised any likelihoods yet
    revised = []

    # for each tuple...
    for network, likelihood, confidence in tuples:
```

```

    # determine the revised likelihood
    revised.append (
        likelihood + (1 - confidence) * (values[network] - likelihood))

# compute each agent's revised likelihood
return revised

#
# Define the pull-down effect:  each agent
# is pulled toward the minimum of its network.
#

def pull_down (tuples):
    return operator (tuples, lambda l: min (l))

#
# Define the mean-reversion effect:  each agent
# is pulled toward the mean of its network.
#

def mean_reversion (tuples):
    return operator (tuples, lambda l: sum (l) / len (l))

#
# Define the lift-up effect:  each agent
# is pulled toward the maximum of its network.
#

def lift_up (tuples):
    return operator (tuples, lambda l: max (l))

#
# Define the herd effect:  each agent votes
# with the majority.
#

def herd (tuples):
    return operator (tuples,
        lambda l: 1 if sum (l) / len (l) > 0.5 else 0)

#
# List the effects.
#

all_effects = [pull_down, mean_reversion, lift_up, herd]

# end of effects.py

```

3.A.7. strategies.py

```

#
# strategies.py
#

```

```

# Implement voting strategies. The model extracts
# the voting strategy from the specification and calls
# it with two arguments: a list of (likelihood,
# confidence) pairs, with one pair per agent, and the
# percent of correct votes required to pass (the
# majority). The function applies a voting strategy
# and tallies the vote. The function returns the
# percent of agents who voted correctly.
#

import math
import random

#
# Given a likelihood, the total number of votes
# cast, and the number of correct votes, calculate
# the probability that we would have seen exactly
# this number of correct votes.
#

def probability (likelihood, total, correct):
    tfact = math.factorial (total)
    cfact = math.factorial (correct)
    ifact = math.factorial (total - correct)
    cprob = likelihood ** correct
    iprob = (1 - likelihood) ** (total - correct)
    return tfact / cfact / ifact * cprob * iprob

#
# Given the total number of votes cast and the
# number of correct votes, calculate the most
# likely probability of correctness of all
# agents.
#

def mostlikely (total, correct):
    highl = 0
    highp = probability (highl, total, correct)
    for l in range (1, 101):
        p = probability (l / 100, total, correct)
        if p > highp:
            highl = l / 100
            highp = p
    return highl

#
# Define the null strategy (sincere voting).
#

def null (pairs, majority):

    # allow each agent to vote
    votes = list (map (lambda p: p[0] > random.random(), pairs))

    # determine whether the vote passed
    return sum (votes) / len (votes)

```

```

#
# Define the predominant strategy. Agents vote
# one at a time. An agent's vote is a weighted
# combination of two likelihoods: its private
# signal and its evaluation of the likelihood
# of the other agents' votes (so far).
#

def predominant (pairs, majority, order=lambda p: random.random()):

    # sort the agents into voting order
    pairs.sort (key=order)

    # extract the first likelihood and confidence
    likelihood, confidence = pairs[0]

    # the first agent votes sincerely
    correct = (likelihood > random.random())

    # one agent has voted so far
    total = 1

    # for each remaining agent...
    for likelihood, confidence in pairs[1:]:

        # if the agent recognizes s/he might be wrong...
        if confidence < 1 - probability (likelihood, total, correct):

            # determine the more likely likelihood
            likelihood = confidence * likelihood + (1 - confidence) * \
                mostlikely (total, correct)

            # cast a revised vote
            correct += (likelihood > random.random())

            # another agent has voted
            total += 1

    # determine whether the vote was correct
    return correct / total

#
# Define the pivotal strategy. An agent votes sincerely,
# except when it is the last agent to vote and its vote
# is pivotal, in which case the agent switches to the
# predominant strategy.
#

def pivotal (pairs, majority, order=lambda p: random.random()):

    # sort the agents into voting order
    pairs.sort (key=order)

    # determine the total number of votes, not counting the last vote
    total = len (pairs) - 1

    # determine the number of correct votes, except for the last vote
    correct = sum (map (lambda p: p[0] > random.random(), pairs[:-1]))

    # determine the last voter's likelihood and confidence
    likelihood, confidence = pairs[-1]

    # if the last voter is pivotal...
    if abs (correct / total - majority) < 0.000001:

```



```

    # if the agent recognizes s/he might be wrong...
    if confidence < 1 - probability (likelihood, total, correct):

        # determine the more likely likelihood
        likelihood = confidence * likelihood + (1 - confidence) * \
            mostlikely (total, correct)

    # cast a revised vote
    correct += (likelihood > random.random())

    # we've cast another vote
    total += 1

    # determine whether the vote was correct
    return correct / total

#
# List the strategies.
#

all_strategies = [null, pivotal, predominant]

# end of strategies.py

```

3.B.1. condorcet.py

```

#
# condorcet.py
#
# A Condorcet agent votes correctly with
# a specified likelihood.
#

import likelihoods

#
# Define the Condorcet specification.
#

def spec (likelihood):
    return {
        "likelihood": lambda a: likelihoods.constant (a, likelihood)
    }

# end of condorcet.py

```

3.B.2. gss.py

```

#
# gss.py
#
# A GSS agent is one characterized by the
# National Opinion Research Council's
# General Social Survey.
#

```

```

import random
import model
import distributions
import factories
import likelihoods
import networks
import effects
import strategies

#
# Define age distribution (age, count).
#

age = distributions.unspaced_list ([
    (18, 194), (19, 757), (20, 799), (21, 899), (22, 939), (23, 1100),
    (24, 1082), (25, 1200), (26, 1187), (27, 1221), (28, 1273), (29, 1149),
    (30, 1242), (31, 1165), (32, 1244), (33, 1193), (34, 1228), (35, 1212),
    (36, 1197), (37, 1165), (38, 1184), (39, 1045), (40, 1124), (41, 1052),
    (42, 1033), (43, 1073), (44, 1014), (45, 959), (46, 968), (47, 940),
    (48, 951), (49, 968), (50, 886), (51, 892), (52, 860), (53, 842),
    (54, 829), (55, 734), (56, 831), (57, 731), (58, 785), (59, 733),
    (60, 768), (61, 672), (62, 710), (63, 686), (64, 600), (65, 658),
    (66, 612), (67, 660), (68, 610), (69, 563), (70, 593), (71, 507),
    (72, 533), (73, 468), (74, 498), (75, 425), (76, 422), (77, 394),
    (78, 353), (79, 309), (80, 274), (81, 273), (82, 238), (83, 214),
    (84, 179), (85, 163), (86, 141), (87, 116), (88, 92), (89, 279)
])

#
# Define sex distribution.
#

male = distributions.unspaced_list ([
    (0, 30827), (1, 24269)
])

#
# Define Black distribution.
#

black = distributions.unspaced_list ([
    (0, 47462), (1, 7625)
])

#
# Define Hispanic distribution.
#

hisp = distributions.unspaced_list ([
    (0, 15235), (1, 1715)
])

```

```

#
# Define education distribution (years completed, count).
#

educ = distributions.unspaced_list ([
    ( 0, 148), ( 1, 39), ( 2, 139), ( 3, 232), ( 4, 299),
    ( 5, 382), ( 6, 725), ( 7, 837), ( 8, 2550), ( 9, 1873),
    (10, 2576), (11, 3295), (12, 16935), (13, 4579), (14, 5909),
    (15, 2414), (16, 6681), (17, 1604), (18, 1885), (19, 719),
    (20, 1086)
])

#
# Define word-score distribution (words correct, count).
#

word = distributions.unspaced_list ([
    ( 0, 209), ( 1, 465), ( 2, 856), ( 3, 1555), ( 4, 2648), ( 5, 4165),
    ( 6, 5552), ( 7, 4009), ( 8, 2674), ( 9, 2060), (10, 1445)
])

#
# Define income distribution (fraction of $250K/year, count).
#

inc = distributions.unspaced_list ([
    (0.00160, 29), (0.00170, 22), (0.00180, 53), (0.00190, 38),
    (0.00195, 44), (0.00210, 41), (0.00215, 42), (0.00220, 28),
    (0.00230, 55), (0.00240, 26), (0.00260, 36), (0.00270, 25),
    (0.00280, 36), (0.00290, 34), (0.00300, 52), (0.00310, 41),
    (0.00320, 42), (0.00330, 44), (0.00340, 49), (0.00360, 60),
    (0.00440, 45), (0.00500, 70), (0.00540, 60), (0.00570, 56),
    (0.00610, 92), (0.00640, 43), (0.00660, 39), (0.00680, 69),
    (0.00710, 75), (0.00750, 48), (0.00780, 69), (0.00830, 53),
    (0.00860, 42), (0.00900, 88), (0.00940, 58), (0.00960, 38),
    (0.01020, 34), (0.01070, 38), (0.01120, 41), (0.01125, 33),
    (0.01160, 53), (0.01165, 27), (0.01200, 72), (0.01220, 50),
    (0.01230, 47), (0.01260, 56), (0.01310, 27), (0.01315, 67),
    (0.01360, 37), (0.01370, 68), (0.01450, 23), (0.01453, 37),
    (0.01456, 78), (0.01490, 13), (0.01500, 40), (0.01570, 39),
    (0.01590, 53), (0.01640, 63), (0.01680, 33), (0.01690, 36),
    (0.01750, 24), (0.01760, 76), (0.01770, 26), (0.01790, 35),
    (0.01830, 19), (0.01860, 30), (0.01880, 37), (0.01930, 27),
    (0.01940, 41), (0.01960, 23), (0.02010, 113), (0.02020, 36),
    (0.02030, 44), (0.02060, 22), (0.02090, 28), (0.02100, 40),
    (0.02110, 43), (0.02140, 39), (0.02145, 37), (0.02150, 86),
    (0.02160, 16), (0.02170, 28), (0.02210, 47), (0.02270, 104),
    (0.02275, 30), (0.02290, 42), (0.02300, 33), (0.02305, 29),
    (0.02350, 32), (0.02390, 39), (0.02410, 19), (0.02415, 27),
    (0.02430, 28), (0.02460, 109), (0.02470, 39), (0.02490, 20),
    (0.02520, 28), (0.02530, 20), (0.02535, 61), (0.02580, 43),
    (0.02610, 21), (0.02650, 61), (0.02690, 33), (0.02700, 43),
    (0.02705, 116), (0.02750, 34), (0.02780, 43), (0.02810, 22),
    (0.02815, 35), (0.02840, 24), (0.02890, 34), (0.02920, 41),
    (0.02925, 26), (0.02950, 32), (0.02955, 39), (0.02990, 24),
    (0.03050, 30), (0.03080, 31), (0.03083, 41), (0.03086, 37),
    (0.03100, 28), (0.03130, 25), (0.03180, 61), (0.03190, 26),
    (0.03210, 41), (0.03260, 49), (0.03300, 37), (0.03320, 16),
    (0.03360, 28), (0.03370, 36), (0.03375, 35), (0.03470, 32),
    (0.03490, 21), (0.03500, 60), (0.03510, 48), (0.03520, 35),
    (0.03610, 55), (0.03615, 72), (0.03640, 28), (0.03720, 73),
    (0.03740, 47), (0.03760, 67), (0.03765, 38), (0.03770, 28),
    (0.03830, 14), (0.03850, 63), (0.03900, 35), (0.03960, 29),
    (0.03970, 26), (0.03973, 123), (0.03976, 59), (0.03980, 46),

```

(0.04020, 23), (0.04040, 73), (0.04100, 46), (0.04200, 18),
 (0.04210, 69), (0.04220, 105), (0.04270, 29), (0.04300, 70),
 (0.04330, 34), (0.04350, 35), (0.04380, 78), (0.04420, 38),
 (0.04440, 36), (0.04500, 48), (0.04510, 53), (0.04570, 63),
 (0.04580, 32), (0.04600, 41), (0.04650, 97), (0.04710, 49),
 (0.04730, 49), (0.04735, 34), (0.04810, 110), (0.04830, 34),
 (0.04835, 45), (0.04840, 45), (0.04850, 108), (0.04920, 41),
 (0.05040, 52), (0.05050, 118), (0.05110, 60), (0.05130, 39),
 (0.05150, 74), (0.05220, 40), (0.05225, 51), (0.05280, 117),
 (0.05350, 85), (0.05400, 64), (0.05405, 40), (0.05410, 62),
 (0.05430, 42), (0.05500, 46), (0.05520, 52), (0.05530, 65),
 (0.05680, 42), (0.05685, 100), (0.05720, 31), (0.05730, 110),
 (0.05750, 61), (0.05880, 89), (0.05900, 64), (0.05910, 52),
 (0.06020, 37), (0.06040, 70), (0.06080, 67), (0.06090, 70),
 (0.06150, 75), (0.06170, 112), (0.06230, 42), (0.06240, 45),
 (0.06290, 67), (0.06320, 69), (0.06450, 123), (0.06510, 78),
 (0.06520, 42), (0.06525, 84), (0.06590, 36), (0.06620, 111),
 (0.06625, 58), (0.06720, 84), (0.06750, 97), (0.06760, 50),
 (0.06830, 56), (0.06870, 73), (0.06950, 92), (0.06980, 59),
 (0.07020, 75), (0.07030, 58), (0.07060, 44), (0.07100, 77),
 (0.07290, 111), (0.07300, 86), (0.07382, 54), (0.07384, 55),
 (0.07386, 93), (0.07430, 64), (0.07500, 139), (0.07520, 44),
 (0.07630, 161), (0.07690, 117), (0.07695, 72), (0.07750, 97),
 (0.07820, 69), (0.07890, 76), (0.07910, 88), (0.07960, 99),
 (0.07970, 63), (0.07990, 47), (0.08020, 106), (0.08050, 57),
 (0.08140, 118), (0.08240, 97), (0.08270, 100), (0.08300, 71),
 (0.08380, 109), (0.08400, 82), (0.08420, 96), (0.08510, 44),
 (0.08680, 87), (0.08720, 60), (0.08780, 48), (0.08785, 99),
 (0.08800, 118), (0.08830, 66), (0.08890, 80), (0.09020, 63),
 (0.09025, 83), (0.09030, 80), (0.09090, 56), (0.09095, 127),
 (0.09140, 70), (0.09220, 59), (0.09240, 81), (0.09400, 74),
 (0.09420, 68), (0.09540, 136), (0.09580, 56), (0.09660, 103),
 (0.09700, 193), (0.09740, 65), (0.09820, 101), (0.09890, 107),
 (0.09920, 64), (0.09950, 117), (0.09980, 122), (0.10060, 57),
 (0.10130, 57), (0.10160, 111), (0.10220, 78), (0.10225, 57),
 (0.10250, 74), (0.10300, 110), (0.10440, 78), (0.10490, 72),
 (0.10660, 105), (0.10660, 55), (0.10700, 140), (0.10800, 94),
 (0.10860, 49), (0.10870, 49), (0.11060, 101), (0.11110, 64),
 (0.11150, 131), (0.11240, 79), (0.11290, 122), (0.11370, 140),
 (0.11400, 62), (0.11430, 66), (0.11450, 60), (0.11470, 222),
 (0.11760, 81), (0.11770, 160), (0.11830, 49), (0.11890, 49),
 (0.12050, 67), (0.12080, 119), (0.12090, 72), (0.12140, 56),
 (0.12160, 89), (0.12170, 144), (0.12300, 60), (0.12320, 50),
 (0.12340, 175), (0.12460, 71), (0.12650, 166), (0.12740, 128),
 (0.12810, 68), (0.12910, 189), (0.12980, 64), (0.13230, 169),
 (0.13235, 90), (0.13290, 59), (0.13410, 63), (0.13430, 182),
 (0.13570, 71), (0.13770, 71), (0.13790, 73), (0.13910, 169),
 (0.13940, 67), (0.14040, 101), (0.14050, 80), (0.14190, 166),
 (0.14240, 70), (0.14290, 69), (0.14450, 102), (0.14500, 41),
 (0.14520, 54), (0.14590, 182), (0.14595, 123), (0.14750, 77),
 (0.14760, 96), (0.14950, 125), (0.14980, 50), (0.15260, 154),
 (0.15370, 175), (0.15380, 79), (0.15390, 79), (0.15500, 106),
 (0.15580, 55), (0.15640, 89), (0.15880, 301), (0.15950, 83),
 (0.16050, 107), (0.16230, 61), (0.16300, 71), (0.16490, 67),
 (0.16495, 95), (0.16610, 72), (0.16790, 63), (0.16830, 106),
 (0.16850, 183), (0.16890, 155), (0.17190, 53), (0.17440, 61),
 (0.17445, 69), (0.17510, 170), (0.17600, 111), (0.17670, 88),
 (0.18040, 61), (0.18180, 60), (0.18270, 85), (0.18600, 178),
 (0.18690, 48), (0.18810, 52), (0.18850, 61), (0.18930, 118),
 (0.19160, 48), (0.19260, 178), (0.19410, 198), (0.19490, 70),
 (0.19690, 89), (0.19840, 54), (0.19860, 85), (0.20120, 48),
 (0.20130, 41), (0.20200, 167), (0.20500, 96), (0.20600, 105),
 (0.20880, 32), (0.20980, 50), (0.21120, 160), (0.21320, 34),
 (0.21400, 136), (0.21510, 59), (0.21650, 70), (0.21680, 98),
 (0.21720, 90), (0.21750, 38), (0.22420, 87), (0.22490, 48),
 (0.22730, 108), (0.22810, 23), (0.22900, 40), (0.23000, 57),

```

(0.23540, 109), (0.23640, 60), (0.23820, 176), (0.23830, 26),
(0.24140, 45), (0.24690, 102), (0.25180, 57), (0.25280, 123),
(0.25500, 17), (0.25540, 34), (0.25820, 78), (0.26090, 50),
(0.26270, 95), (0.26460, 38), (0.26500, 63), (0.26810, 64),
(0.26980, 55), (0.27410, 61), (0.27480, 31), (0.27660, 30),
(0.27890, 54), (0.27900, 94), (0.28110, 34), (0.28890, 64),
(0.29040, 59), (0.29110, 104), (0.29510, 16), (0.30300, 70),
(0.30400, 30), (0.30760, 31), (0.30770, 24), (0.30890, 68),
(0.31690, 49), (0.31890, 21), (0.32100, 41), (0.32120, 36),
(0.32480, 25), (0.32970, 78), (0.32980, 20), (0.33220, 38),
(0.33580, 14), (0.34100, 37), (0.34500, 12), (0.35280, 64),
(0.35310, 37), (0.36340, 23), (0.36940, 42), (0.37450, 43),
(0.37620, 50), (0.38540, 28), (0.38910, 23), (0.39860, 28),
(0.40260, 36), (0.41120, 30), (0.41340, 25), (0.41530, 26),
(0.42340, 50), (0.42360, 33), (0.42530, 26), (0.42800, 20),
(0.43290, 35), (0.43900, 87), (0.44470, 53), (0.44970, 14),
(0.45560, 29), (0.46510, 6), (0.47030, 66), (0.47300, 35),
(0.48540, 22), (0.48680, 24), (0.49400, 22), (0.53190, 18),
(0.55890, 47), (0.58050, 22), (0.62150, 88), (0.62530, 22),
(0.62550, 7), (0.64390, 45), (0.70140, 40), (0.75900, 61),
(0.83030, 13), (0.94280, 70)
])

#
# Define Republicanism (degree, count).
#

repub = distributions.unspaced_list ([
    (0, 8761), (1, 11697), (2, 6508), (3, 8126),
    (4, 4764), (5, 8755), (6, 5356)
])

#
# Define the GSS agent factory.
#

def factory (count):
    return factories.characteristic (count, [
        ("age", age, []),
        ("male", male, [ 0.01]),
        ("black", black, [-0.06, -0.08]),
        ("hisp", hisp, [-0.13, 0.03, -0.11]),
        ("educ", educ, [ 0.07, -0.04, -0.10, -0.24]),
        ("word", word, [ 0.17, -0.05, -0.17, -0.19, 0.43]),
        ("inc", inc, [ 0.23, 0.25, -0.11, -0.12, 0.37, 0.22]),
        ("repub", repub, [ 0.02, 0.11, -0.29, -0.10, 0.03, 0.06, 0.11])
    ])

```

```

#
# Define an unmeasured version of the GSS likelihood
# of correctness function.
#

def likelihood (agents):
    return likelihoods.characteristic (agents, lambda agent:
        random.gauss ( 0.636, 0.10) +
        random.gauss (-0.111, 0.11) * agent["age"] +
        random.gauss ( 0.062, 0.03) * agent["male"] +
        random.gauss (-0.148, 0.05) * agent["black"] +
        random.gauss (-0.111, 0.05) * agent["hisp"] +
        random.gauss ( 0.385, 0.12) * agent["educ"] +
        random.gauss ( 0.334, 0.09) * agent["word"] +
        random.gauss ( 0.123, 0.14) * agent["inc"] +
        random.gauss ( 0.031, 0.04) * agent["repub"])

#
# Define the GSS network profiles.
#

profile = {
    "age" : 4,
    "male" : 2,
    "black" : 2,
    "hisp" : 2,
    "educ" : 5,
    "word" : 2,
    "inc" : 5,
    "repub" : 3
}

#
# Define a GSS network.
#

def network (profile={}):
    return lambda agents: networks.partition (agents, profile)

#
# Define a complexity-adjusted likelihood function.
#

def complexity (comp, function=likelihood):
    return lambda agents: \
        [(1 - comp) * l for l in function (agents)]

#
# Define an agent's confidence.
#

def confidence (agents):
    return [1 for a in agents]

#
# Define a wise agent's confidence.
#

def wise (agents):
    return [1 + agent["educ"] * (likelihood - 1)
            for agent, likelihood in zip (agents, likelihood (agents))]
```

```

#
# Define an unwise agent's confidence.
#

def unwise (agents):
    return [4 * agent["educ"] * likelihood * (likelihood - 1) + 1
            for agent, likelihood in zip (agents, likelihood (agents))]

#
# Define the GSS specification.
#

def spec (comp=0.5):
    return {
        "factory"      : factory,
        "likelihood"   : complexity (comp),
        "confidence"   : unwise,
        "network"      : network (profile),
        "effect"       : effects.mean_reversion,
        "strategy"     : strategies.null
    }

#
# Define the GSS model.
#

def percent (p=None):
    return model.percent (spec=spec(), policy=p, count=1001, runs=10, avg=True)

# end of gss.py

```

3.B.3. wise.py

```

#
# wise.py
#
# Implement a "wise" agent.
#

import distributions
import factories
import likelihoods
import confidences
import networks
import effects

#
# Create agents with normally distributed
# likelihoods of correctness and constant
# wisdoms.
#

def factory (mean, std, wisdom):
    return lambda count: factories.characteristic (count, [
        ("L", distributions.normal (mean, std), []),
        ("W", distributions.constant (wisdom), [0])])

```

```

#
# Retrieve the likelihood of correctness.
#

def likelihood (agents):
    return likelihoods.characteristic (agents,
        lambda agent: agent["L"])

#
# Retrieve agent confidence.
#

def confidence (agents):
    return confidences.characteristic (agents,
        lambda agent: 1 + agent["W"] * (agent["L"] - 1))

#
# Define the specification.
#

def spec (mean, std, wisdom, effect=effects.mean_reversion):
    return {
        "factory"      : factory (mean, std, wisdom),
        "likelihood"   : likelihood,
        "confidence"   : confidence,
        "network"      : lambda agents: networks.uniform (agents, 1),
        "effect"       : effect
    }

# end of wise.py

```

3.B.4. unwise.py

```

#
# unwise.py
#
# Implement an "unwise" agent.
#

import distributions
import factories
import likelihoods
import confidences
import networks
import effects

#
# Create agents with normally distributed
# likelihoods of correctness and constant
# wisdoms.
#

def factory (mean, std, wisdom):
    return lambda count: factories.characteristic (count, [
        ("L", distributions.normal (mean, std), []),
        ("W", distributions.constant (wisdom), [0])]

```



```

#
# Retrieve the likelihood of correctness.
#

def likelihood (agents):
    return likelihoods.characteristic (agents,
        lambda agent: agent["L"])

#
# Retrieve agent confidence.
#

def confidence (agents):
    return confidences.characteristic (agents,
        lambda agent: 4 * agent["W"] * agent["L"] * (agent["L"] - 1) + 1)

#
# Define the specification.
#

def spec (mean, std, wisdom, effect=effects.mean_reversion):
    return {
        "factory"      : factory (mean, std, wisdom),
        "likelihood"   : likelihood,
        "confidence"   : confidence,
        "network"      : lambda agents: networks.uniform (agents, 1),
        "effect"       : effect
    }

# end of unwise.py

```

3.B.5. fragmented.py

```

#
# fragmented.py
#
# Implement agents belonging to fragmented
# social networks.
#

import distributions
import factories
import likelihoods
import confidences
import networks
import effects

#
# Create agents with normally distributed
# likelihoods of correctness, belonging to
# a specific number of networks that are
# correlated with likelihoods.
#

def factory (mean, std, netcount, netcorr):
    return lambda count: factories.characteristic (count, [
        ("L", distributions.normal (mean, std), []),
        ("N", distributions.quantized (netcount), [netcorr])])

```

```

#
# Retrieve the likelihood of correctness.
#

def likelihood (agents):
    return likelihoods.characteristic (agents,
        lambda agent: agent["L"])

#
# Retrieve agent confidence.
#

def confidence (agents):
    return confidences.characteristic (agents,
        lambda agent: 2 * agent["L"] * (agent["L"] - 1) + 1)

#
# Retrieve agent network.
#

def network (agents):
    return networks.characteristic (agents,
        lambda agent: agent["N"])

#
# Define the specification.
#

def spec (mean, std, count, corr, effect=effects.mean_reversion):
    return {
        "factory"      : factory (mean, std, count, corr),
        "likelihood"   : likelihood,
        "confidence"   : confidence,
        "network"      : network,
        "effect"       : effect
    }

# end of fragmented.py

```

3.C. distributions.py

```

#
# distributions.py
#
# Each distribution function returns another function
# which, when called with no arguments, returns values
# from the specified distribution, curtailed to the
# range [0,1].
#

import math
import random

```

```

#
# The curtail() function returns a function which
# takes a draw from a distribution and ensures that
# the result is between 0 and 1.
#

def curtail (distribution):

    # subfunction: invoke and curtail the underlying distribution
    def invoke (distribution):

        # fetch the next value from the distribution
        value = distribution()

        # if the value is less than 0, set to 0
        if value < 0:
            value = 0

        # if the value is greater than 1, set to 1
        elif value > 1:
            value = 1

        # return the value
        return value

    # return a function which returns a curtailed value
    return lambda: invoke (distribution)

#
# Provide a constant distribution with the specified value.
#

def constant (value=0.5):
    return curtail (lambda: value)

#
# Provide a bimodal distribution with the specified values.
#

def bimodal (value=0.5, low=0, high=1):
    return curtail (lambda: high if random.random() < value else low)

#
# Provide a uniform distribution.
#

def uniform():
    return random.random

#
# Provide a quantized distribution.
#

def quantized (count=1):
    return lambda: int (random.random() * count) / count

#
# Provide a normal distribution over the specified range.
#

```

```

def normal (mean=0.5, std=0.1):
    return curtail (lambda: random.gauss (mean, std))

#
# Provide a Cauchy distribution over the specified range.
#

def cauchy (median=0.5):
    return curtail (
        lambda: median + math.tan (math.pi * (random.random() - 0.5)))

#
# It is often convenient to express distributions in terms
# of numbers of individuals. For example, if we have a
# population of 1000 individuals, and each individual can
# have one of three education levels, it might be convenient
# to express the distribution as [329, 517, 154]. The
# spaced_list() function accepts a list of counts and
# returns a corresponding distribution function. In this
# example, the returned distribution function would return
# 0.0 with probability 0.329, 0.5 with probability 0.517,
# and 1.0 with probability 0.154.
#
# This function assumes that the counts are evenly spaced.
# If the counts are not evenly spaced, either insert zeroes
# to fill the gaps or use unspaced_list() instead. The
# unspaced_list() function takes pairs of (value, count)
# values, instead of the counts alone. For example,
# consider a population of 1000 individuals with one
# of four education levels, and suppose that the counts
# are [329, 517, 0, 154] (that is, no individuals in
# the sample have the third education level). In this
# case, spaced_list() can be used (with the zero included),
# because each value is evenly spaced from the others.
# But in a distribution representing, e.g., income,
# zero-filling many gaps would be tedious. In that
# situation, use unspaced_list() instead. In the above
# example, unspaced_list() would be called with the
# following argument: [(1, 329), (2, 517), (4, 154)].
#

def unspaced_list (pairs):

    # subfunction: select a value according to a probability
    def select (probabilities):

        # generate a random value from the uniform distribution
        probability = random.random()

        # for each probability...
        for (p, v) in probabilities:

            # if we've selected the value with this probability...
            if probability < p:

                # return the associated value
                return v

    # sort the pairs
    pairs.sort (key=lambda p: p[0])

    # identify the minimum value
    minval = pairs[0][0]

```

```

# identify the maximum value
maxval = pairs[-1][0]

# calculate the total count
total = sum (map (lambda p: p[1], pairs))

# we haven't seen any values yet
count = 0

# we haven't calculated any probabilities yet
probabilities = []

# for each value...
for (v, c) in pairs:

    # we've seen another count
    count += c

    # normalize the value associated with this count
    value = (v - minval) / (maxval - minval)

    # save the probability and its normalized value
    probabilities.append ((count / total, value))

# return the distribution function
return lambda: select (probabilities)

def spaced_list (counts):

    # enumerate the counts and call unspaced_list()
    return unspaced_list (list (enumerate (counts)))

# end of distributions.py

```

REFERENCES

REFERENCES

- Austen-Smith, David and Jeffrey S. Banks. 1996. Information Aggregation, Rationality, and the Condorcet Jury Theorem. *American Political Science Review* 90(1):34-45.
- Baker, Keith Michael. 1975. *Condorcet: From Natural Philosophy to Social Mathematics*. Chicago: University of Chicago Press.
- Ball, Philip. 2004. The Physical Modelling of Human Social Systems. *ComplexUs* 1:190-206.
- Barber, William J. 1996. *Designs within Disorder: Franklin D. Roosevelt, the Economists, and the Shaping of American Economic Policy, 1933-1945*. Cambridge: Cambridge University Press.
- Barro, Robert J. and David B. Gordon. 1983. A Positive Theory of Monetary Policy in a Natural Rate Model. *Journal of Political Economy* 91(4):589-610.
- Becker, Gary. 1983. A Theory of Competition Among Pressure Groups for Political Influence. *Quarterly Journal of Economics* 98(3):371-400.
- Bernstein, Marver H. 1955. *Regulating Business by Independent Commission*. Princeton: Princeton University Press.
- Black, Duncan. 1987. *The Theory of Committees and Elections*. Cambridge: Cambridge University Press.
- Blinder, Alan S. 1997. What Central Bankers Could Learn From Academics—and Vice Versa. *Journal of Economic Perspectives* 11(2):3-19.
- Boland, Philip J. 1989. Majority Systems and the Condorcet Jury Theorem. *Statistician* 38(3):181-189.
- _____, Frank Proschan and Y.L. Tong. 1989. Modeling Dependence in Simple and Direct Majority Systems. *Journal of Applied Probability* 26:81-88.

- Branch, William A. 2004. The Theory of Rationally Heterogeneous Expectations: Evidence from Survey Data on Inflation Expectations. *Economic Journal* 114:592-621.
- Buchanan, James and Robert Tollison. 1984. *The Theory of Public Choice-II*. Ann Arbor: University of Michigan Press.
- Condorcet, Marquis de. 1995 [1785]. Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix. In *The French Revolution Research Collection*. Oxford: Pergamon Press.
- Cukierman, Alex and Allan H. Meltzer. 1986. A Theory of Ambiguity, Credibility, and Inflation under Discretion and Asymmetric Information. *Econometrica* 54(5):1099-1128.
- Curtin, Deirdre. 2009. *Executive Power of the European Union*. Oxford: Oxford University Press.
- Dal Bo, Ernesto. 2006. Regulatory Capture: A Review. *Oxford Review of Economic Policy* 22(2):203-225.
- Dietrich, Franz. 2008. The Premises of Condorcet's Jury Theorem are not Simultaneously Justified. *CPNSS working paper* 4(2). London: London School of Economics, Center for Philosophy of Natural and Social Science.
- Downey, Allen B. 2012. *Think Complexity*. Sebastopol, California: O'Reilly Media.
- Downs, Anthony. 1957. *An Economic Theory of Democracy*. New York: Harper & Row.
- Epstein, Joshua M. and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up*. Cambridge: MIT Press.
- Fagiolo, Giorgio, Alessio Moneta, and Paul Windrum. 2007. A Critical Guide to Empirical Validation of Agent-Based Economics Models: Methodologies, Procedures, and Open Problems. *Computational Economics* 30(3):195-226.
- Feddersen, Timothy J. and Wolfgang Pesendorfer. 1996. The Swing Voter's Curse. *American Economic Review* 86(3):408-424.
- _____. 1997. Voting Behavior and Information Aggregation in Elections with Private Information. *Econometrica* 65(5):1029-1058.

- _____. 1998. Convicting the Innocent: The Inferiority of Unanimous Jury Verdicts under Strategic Voting. *American Political Science Review* 92(1):23-35.
- Funk and Wagnalls. 1971. *New Encyclopedia*. New York: Funk and Wagnalls.
- Gjerstad, Steven and John Dickhaut. 1998. Price Formation in Double Auctions. *Games and Economic Behavior* 22:1-29.
- Glaeser, Edward L., Bruce Sacerdote, and Jose A. Scheinkman. 1996. Crime and Social Interactions. *Quarterly Journal of Economics* 111(2):507-548.
- Goff, Brian. 1996. Why Regulation? In Michael A. Crew (ed.), *Regulation and Macroeconomic Performance*. New York: Kluwer Academic Publishers.
- Gomes, Orlando. 2006. Optimal Monetary Policy under Heterogeneous Expectations. *ICFAI Journal of Monetary Economics* 4:32-51.
- Guerrero, Omar A. and Robert Axtell. 2011. Using Agentization for Exploring Firm and Labor Dynamics. *Emergent Results of Artificial Economies: Lecture Notes in Economics and Mathematical Systems* 652(4):139-150.
- Haltiwanger, John and Michael Waldman. 1985. Rational Expectations and the Limits of Rationality: an Analysis of Heterogeneity. *American Economic Review* 75(3):326-340.
- Hayek, Friedrich A. 1976. *Law, Legislation and Liberty*. Chicago: University of Chicago Press.
- Hoeffding, Wassily. 1956. On the Distribution of the Number of Successes in Independent Trials. *Annals of Mathematical Statistics* 27:713-721.
- Huntington, Samuel P. 1952. The Marasmus of the ICC: The Commission, the Railroads, and the Public Interest. *The Yale Law Journal* 61(4):467-509.
- Ireland, Peter N. 1998. Expectations, credibility, and time-consistent monetary policy. Working paper 9812, Federal Reserve Bank of Cleveland.
- Johnson, Dominic D.P. and James H. Fowler. 2011. The Evolution of Overconfidence. *Nature* 477:317-320.

- Koriyama, Yukio and Balazs Szentes. 2007. A Resurrection of the Condorcet Jury Theorem. Unpublished manuscript dated September 25, 2007, available at <<http://home.uchicago.edu/~szentes/resurr.pdf>>.
- Kroszner, Randall S. and Philip E. Strahan. 1999. What Drives Deregulation? Economics and Politics of the Relaxation of Bank Branching Restrictions. *Quarterly Journal of Economics* 114(4):1437-1467.
- Landis, James M. 1938. *The Administrative Process*. New Haven: Yale University Press.
- List, Christian and Robert E. Gooden. 2001. Epistemic Democracy: Generalizing the Condorcet Jury Theorem. *Journal of Political Philosophy* 9(3):277-306.
- Lucas, Robert E. 1976. Econometric Policy Evaluation: a Critique. In K. Brunner and A. H. Meltzer (Eds.), *The Phillips curve and labor markets*. Amsterdam: North-Holland.
- Lutz, Mark. 2011. *Programming Python, Fourth Edition*. Sebastopol, California: O'Reilly.
- Macal, Charles M. & Michael J. North. 2010. Tutorial on Agent-Based Modeling and Simulation. *Journal of Simulation* 4:151-162.
- Markose, Sheri M. 2005. Computability and Evolutionary Complexity: Markets as Complex Adaptive Systems. *The Economic Journal* 115:F159-192.
- McCraw, Thomas K. 1975. Regulation in America, a Review Article. *The Business History Review* 49(2):59-83.
- Miller, Nicholas R. 1986. Information, Electorates and Democracy: Some Extensions and Interpretations of the Condorcet Jury Theorem. In B. Grofman and G. Owen (Eds.), *Information Pooling and Group Decision Making*. Greenwich, Connecticut: JAI Press.
- Mueller, Dennis C. 2003. *Public Choice III*. Cambridge: Cambridge University Press.
- Myerson, Roger B. 1998. Extended Poisson Games and the Condorcet Jury Theorem. *Games and Economic Behavior* 25:111-131.
- Niskanen, William A. 1968. The Peculiar Economics of Bureaucracy. *American Economic Review* 58(2):293-305.

- Olson, Mancur. 1965. *The Logic of Collective Action*. Cambridge: Harvard University Press.
- Pareto, Vilfredo. 1955 [1911]. Mathematical Economics. *International Economic Papers* 5:58-102.
- Peltzman, Samuel. 1976. Toward a More General Theory of Regulation. *Journal of Law and Economics* 19:211-48.
- _____. 1980. Growth of Government. *Journal of Law and Economics* 23:209-287.
- _____. 1989. The Economic Theory of Regulation after a Decade of Deregulation. In *Brookings Papers on Economic Activity: Microeconomics*. Washington: Brookings Institution.
- _____. 1993. George Stigler's Contribution to the Economic Analysis of Regulation. *Journal of Political Economy* 101(5):818-832.
- Popper, Karl. 1959. *The Logic of Scientific Discovery*. New York: Basic Books.
- Posner, Richard. 1974. Theories of Economic Regulation. *Bell Journal of Economics and Management Science* 5(2):335-58.
- Riechmann, Thomas. 2001. *Learning in Economics: Analysis and Application of Genetic Algorithms*. Heidelberg: Physica-Verlag.
- Riker, W. H. 1979. Is "A New and Superior Process" Really New? *Journal of Political Economy* 87:785-890.
- Rockoff, Hugh. 1984. *Drastic Measures: A History of Wage and Price Controls in the United States*. Cambridge: Cambridge University Press.
- Rubinstein, Ariel and Asher Wolinsky. 1985. Equilibrium in a Market with Sequential Bargaining. *Econometrica* 53(3):1133-1150.
- Smith, Adam. 1994 [1776]. *The Wealth of Nations*. New York: The Modern Library.
- Spaeth, Barbette S. 1996. *The Roman goddess Ceres*. Austin: University of Texas Press.
- Stigler, George. 1971. The Theory of Economic Regulation. *Bell Journal of Economics and Management Science* 2:3-21.

- _____ and Claire Friedland. 1961. What Can Regulators Regulate? The Case of Electricity. *Journal of Law and Economics* 5:1-16.
- _____, Stephen M. Stigler, and Claire Friedland. 1995. The Journals of Economics. *The Journal of Political Economy* 103(2):331-359.
- Surowiecki, James. 2004. *The Wisdom of Crowds*. New York: Anchor Books.
- Tobias, Robert and Carole Hofmann. 2004. Evaluation of Free Java-Libraries for Social-Scientific Agent Based Simulation. *Journal of Artificial Societies and Social Simulation* 7(1):6.
- Tesfatsion, Leigh. 2006. Agent-Based Computational Economics: a Constructive Approach to Economic Theory. In L. Tesfatsion & K. Judd (Eds.), *The Handbook of Computational Economics: Agent-Based Computational Economics, Volume 2*. Amsterdam: North-Holland.
- Tideman, T. Nicolaus and Gordon Tullock. 1976. A New and Superior Process For Making Social Choices. *Journal of Political Economy* 84:1145-1159.
- _____. 1981. Coalitions under Demand Revealing. *Public Choice* 36(2):323-328.
- Tullock, Gordon. 1966. Information Without Profit. *Papers on Non-Market Decision Making* 1:141-159.
- _____. 1967a. Hotelling and Downs in Two Dimensions. In *Toward a Mathematics of Politics*. Ann Arbor: University of Michigan Press, 50-61.
- _____. 1967b. Proportional Representation. In *Toward a Mathematics of Politics*. Ann Arbor: University of Michigan Press, 144-157.
- _____. 1967c. The Politics of Persuasion. In *Toward a Mathematics of Politics*. Ann Arbor: University of Michigan Press, 115-132.
- _____. 1967d. Political Ignorance. In *Toward a Mathematics of Politics*. Ann Arbor: University of Michigan Press, 100-114.
- _____. 1970. A Simple Algebraic Logrolling Model. *American Economic Review* 60:419-426.
- _____. 1971. The Paradox of Revolution. *Public Choice* 11:89-99.

- _____. 1972a. The Edge of the Jungle. In G. Tullock (Ed.), *Explorations in the theory of anarchy*. Blacksburg, Virginia: Center for Study of Public Choice, 65-75.
- _____. 1972b. Economic Imperialism. In J. M. Buchanan and R. D. Tollison (Eds.), *Theory of Public Choice: Political Applications of Economics*. Ann Arbor: University of Michigan Press, 317-329.
- _____. 1974. *The Social Dilemma: the Economics of War and Revolution*. Blacksburg, Virginia: Center for Study of Public Choice.
- _____. 1976. *An Essay in the Economics of Politics*. London: Institute of Economic Affairs.
- _____. 1980. Efficient Rent Seeking. In J. M. Buchanan, R. D. Tollison, and G. Tullock (Eds.), *Toward a Theory of the Rent-Seeking Society*. College Station: Texas A&M University Press, 97-112.
- _____. 1981. Why So Much Stability? *Public Choice* 37(2):189-202.
- _____. 1982. Income Testing and Politics: a Theoretical Model. In I. Garfinckel (Ed.), *Income-Tested Transfer Programs: the Case For and Against*, 97-116.
- _____. 1983a. Information and Logrolling. In *Economics of Income Redistribution*. Boston: Kluwer-Nijhoff, 33-48.
- _____. 1983b. Horizontal Transfers. In *Economics of income redistribution*. Boston: Kluwer-Nijhoff, 17-31.
- _____. 1987. Public Choice. In J. Eatwell, M. Milgate, and P. Newman (Eds.), *The New Palgrave: a Dictionary of Economics, Volume 3*. London: Macmillan, 1040-1044.
- _____. 2000. People Are People. In G. Tullock, A. Seldon, and G. L. Brady (Eds.), *Government: Whose Obedient Servant? A Primer in Public Vchoice*. London: Institute of Economic Affairs, 3-18.
- _____. 2005. *The Economics and Politics of Wealth Redistribution*. Indianapolis: Liberty Fund.
- _____. and Colin D. Campbell. 1970. Computer Simulation of a Small Voting System. *Economic Journal* 80:97-104.

- Vriend, Nicholas. J. 2002. Was Hayek an ACE? *Southern Economic Journal* 68(4):811-840.
- Wagner, Richard E. 2007. *Fiscal Sociology and the Theory of Public Finance*. Cheltenham, England: Edward Elgar Publishing.
- _____. 2008. Finding Social Dilemma: West of Babel, not East of Eden. *Public Choice*, 135(1/2):55-66.
- Walsh, Carl E. 1998. *Monetary Theory and Policy*. Cambridge: MIT Press.
- White, Matthew W. 1996. Power Struggles: Explaining Deregulatory Reforms in Electricity Markets. In *Brookings Papers: Microeconomics 1996*. Washington: Brookings Institution.
- Young, H. Peyton. 1997. Group Choice and Individual Judgments. In D. Mueller (Ed.), *Perspectives on Public Choice*. Cambridge: Cambridge University Press.
- Zhu, Jianjun and Lawrence G. Weiss. 2005. The Weschler Scales. In D. Hanagan and P. Harrison (Eds.), *Contemporary Intellectual Assessment: Theories, Tests, and Issues*. New York: The Guilford Press, 294-324.

CURRICULUM VITAE

Richard R. Wallick graduated from Taipei American School, Taipei, Taiwan, in 1987. He received his Bachelor of Science in Computer Science from the University of Calgary in 1992. He was employed as a software engineer in the Washington, D.C. area until 2005. He received his Master of Arts in Economics from George Mason University in 2007. He now works for the Bureau of Labor Statistics as a member of the Consumer Price Index team.