LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH
Ollscoil Átha Cliath

TRINITY COLLEGE LIBRARY DUBLIN
The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

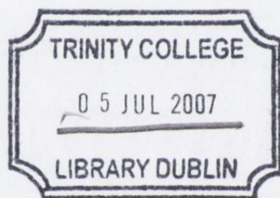
Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.



THESIS
8121

Self-Organising Skew Distributions in an Agent-Based Model With Applications to Gibrat's Law

Ana Nelson

Ph.D. Thesis

2007

Department of Economics
University of Dublin, Trinity College

DECLARATION

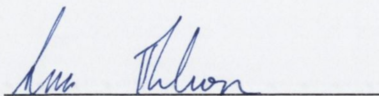
I hereby declare that:

a) this thesis has not been submitted as an exercise for a degree at this or any other University and

b) this thesis is entirely my own work.

I hereby agree that:

c) the Library may lend or copy this thesis upon request.



Ana Nelson

Summary

Three models for the firm size distribution are implemented using an Agent-Based Modelling (ABM) framework, and a theory explaining the source of a skew firm size distribution in one of the models is proposed. The variable effort model produces a skew distribution of firm sizes, we propose, due to the presence within the model of endogenous birth and proportional growth forces. These forces result from the instantiation within an agent-based model of contradictory microeconomic tendencies, namely the free riding tendency due to Cobb-Douglas income leisure preferences and the increasing returns to scale of the firm production function. All of these elements have a role to play in the resultant generation of a skew firm size distribution.

Acknowledgements

I am indebted to my supervisor, Jacco Thijssen, for his many helpful comments and insights, and for his support and encouragement. I am also grateful to the Department of Economics and the University for financial and academic support. I wish to acknowledge the developers and user communities of RePast, R, TeX and TextMate software in particular, and the open source community in general, for the many freely available resources which were utilised during the preparation of this work. Finally, numerous conference participants have made helpful comments and provided feedback, and I would particularly like to thank Robert Axtell for his input.

Contents

1 Introduction **3**

2 Skew Probability Distributions **9**

2.1 Skew Probability Distributions 9

2.1.1 The Power Law and Lognormal Distributions 12

2.1.2 The Fractal Distributions 15

2.2 Some Example Datasets 16

2.2.1 Words in Beowulf 17

2.2.2 Genera and Species of Snake 18

2.2.3 Cities in the United States 19

2.2.4 Discussion 19

2.3 Gibrat's Law 29

2.3.1 Simon's Approach 30

2.3.2 Gabaix's Approach 32

2.4 Scale Invariance and Power Laws 34

3 An Agent-Based Firms Framework **37**

4 A Variable Effort Model **43**

4.1 Introduction 43

4.2 Case Studies 45

4.2.1 Large Firm Case Study 45

4.2.2 Medium Firm Case Study 47

4.3 θ, Effort, Utility and the Firm 54

4.3.1 Founder's θ 54

4.3.2 Agent Income, Effort and Utility 55
4.4 Firm Size Distribution 70
4.5 Discussion 71

5 Extensions of a Variable Effort Model 77
5.1 Sequential Activation 77
5.2 Myopia 88
5.3 Discussion 89

6 An Exogenous Birth Model 97
6.1 Implementing Exogenous Birth with Agents 98
6.2 A Survey of the Parameter Space 99
6.3 Firm Size Distribution 105
6.3.1 A "Super" Firm 105
6.3.2 More Diversity 106
6.3.3 Small Firms Only 107
6.4 Discussion 110

7 A Cost Curve Model 113
7.1 Implementing Cost Curves with Agents 114
7.1.1 Static Economies of Scale 114
7.1.2 Dynamic Economies of Scale 116
7.2 Results 118
7.2.1 Static Economies of Scale 118
7.2.2 Dynamic Economies of Scale 120
7.3 Discussion 126

8 Conclusions 127
References 130

A Source Code 135

Chapter 1

Introduction

> It is time to see how the new ideas can usefully be applied to that immensely complex, but indisputably self-organizing system we call *the economy.* (Krugman 1996)[p. vi]

The economy, as Krugman and many others have observed, is self-organising. We all participate in this self-organisation on a daily basis, our ancestors have done so for tens of thousands of years. Most of us would think of such participation as a human right, if it were not so ubiquitous that we usually do not think about it at all. We choose where, when and how to sell our labour, or the fruits of it, and where, when and how to buy the fruits of others'. All of the larger structures in the economy, firms for example, result from the interaction over time of a large number of individual acts of economic participation. Even in the most highly-centralised ordered economies, underground markets flourish, driven by individuals seeking the benefits of trade. A self-organising economy is a seemingly unstoppable emergent property of human group behaviour.

Research into self-organizing systems has arisen out of the study of complex systems, which itself is rooted in the disciplines of complexity theory and chaos theory. Computer simulation gave birth to chaos theory, with Lorenz' famous meteorology simulation in which a small change in input resulted in a very different output, and is also the primary means we have to study it and its descendants. "To chaos researchers, mathematics has

become an experimental science, with the computer replacing laboratories full of test tubes and microscopes." (Gleick 1987)[p. 38] Whilst in principle a modern computer can only solve an extremely small subset of those problems which the human mind is capable of solving, in reality computers open up new realms of research with their two fundamental advantages, speed and memory. Instead of having to reduce complex systems to a small number of linear differential equations to make them analytically tractable, computer simulation allows us to make different types of simplifications which may prove to retain more of that which interested us in the original and which will, at the very minimum, give us a new perspective.

Since a computer is capable of remembering the preferences, attributes and current states of a very large number of objects, it is possible to construct a type of simulation which mimics the interaction of agents in an economy. Agents in such a simulation are discrete entities in virtual time and space. Even if they are homogeneous in design (though they seldom are) they are separate, distinct beings, and they interact as such. The economy moves forward, one step at a time, because one agent interacts in some way with another agent. The study of economics using Agent-Based Modelling (ABM) is necessarily about more than just the current state of affairs, it is also about *how* the economy arrived there. It is about interaction/process, as well as aggregation/state. The two are inseparable. To arrive at a particular state, the system must have experienced the path of a process leading to that state.

In an agent-based model, the macroeconomics of the system are an emergent property of the microeconomics. (Or, the *state* of the system is an emergent property of the *process*.) Behaviour is only programmed at the microeconomic level, agents interact locally with other agents. Macro-level regularities or patterns are said to be *emergent* since they are a consequence of interaction within the system, and not externally specified or programmed. This structure enables a researcher to study the impact of a micro-level change on the macro-level emergent properties of a system.

In this thesis, we will be employing various agent-based models to examine a question which has had a strong impact on the evolution of the theory

of the firm. Why does the distribution of firm sizes take the form that it does, namely a skew distribution resembling a power law? The skew distribution of firm sizes represents the state, at a point in time, of one aspect of the economy. We will be looking at processes which might lead to that state. Firms, being large-scale complex structures, will not be programmed into our simulation but will arise through the interaction of agents. Agents will be programmed to organise themselves into firms using a variety of behavioural rules, which will vary from model to model. Thus the simulated firms themselves, as well as information about the entire population of firms, including the size distribution, will be emergent properties of the simulation.

The empirical skew distribution of firm sizes has been a focal point for research into the growth of firms since at least the time of Gibrat. Gibrat proposed that a lognormal distribution would fit the empirical observations he made for French firm sizes, and based upon a stochastic process which leads to a lognormal distribution, he proposed that firm growth rates are random variables independent of firm size, or equivalently that absolute firm growth is proportional to firm size. Subsequent researchers have refined and modified this algorithm, either to produce a slightly different probability distribution such as the Yule or Pareto, or to improve the economic intuition of the model.

The power law or Pareto distribution is easily recognised since its probability density function (PDF) has a straight-line shape in double logarithmic coordinates. The lognormal PDF has a similar shape, it is a parabola in double logarithmic coordinates, which with certain parameter values can appear almost linear. There is a closer relationship between these distributions than simply a visual similarity, the generative models which lead to a lognormal can be altered slightly to produce a power law distribution instead. It is therefore not surprising that there has been considerable debate in the literature as to whether the lognormal or power law is a better fit to empirical data. There is no consensus. For some data sets the lognormal is a better fit, for others the power law. For many, neither fits particularly well, causing some researchers to wish to abandon this line of reasoning al-

together. Sutton, who has written extensively on the work descending from Gibrat's original book, summarises the state of play as follows:

> In particular, there is no obvious rationale for positing any general relationship between a firm's size and its expected growth rate, nor is there any reason to expect the size distribution of firms to take any particular form for the general run of industries. Most authors now claim only that the distribution will be skew, but do not specify the extent of the skewness, of the particular form the size distribution might take.
> This new agnosticism as to the form of the size distribution meshes well with the empirical finding ... according to which no particular form of size distribution can be justified as typical across the general run of industries. (Sutton 1998)[p. 245]

Thus, the emphasis seems to be moving away from seeking a definitive answer as to the precise nature of this distribution, in recognition of the fact that there probably isn't one. Furthermore, Gibrat's hypothesis, the Law of Proportional Growth, has not stood up to econometric scrutiny and is no longer intuitively appealing (Cefis, Ciccarelli, and Orsenigo 2004). Where to go from here? The agent-based models being explored in this thesis have opened up a new way of thinking about this situation. In one of the models, firms do exhibit a sort of proportional growth in agreement with Gibrat, but such growth is not an assumption of the model. This pattern emerges from the programmed agent-level behaviour. In this same model we also observe a skew distribution of firm sizes. The growth is not exactly like Gibrat, and the resulting distribution of firm sizes is not shaped precisely like a power law or lognormal distribution, but it is plausible that a process which behaves similarly to a Gibrat proportional growth process would result in a similar probability distribution, and if unrelated agent-level behaviours can mimic such a process at the firm level, then we can construct a new, more intuitively appealing foundation for the macroeconomic theory of firm size distribution based upon modern microeconomic theories of firm organisation and behaviour. Such work is beyond the scope of this thesis,

but what will be done here is to explore in a computational laboratory setting the principles that might guide such later work.

It must be stressed that the research herein is of a theoretical nature, and we are interested in the skew distribution of firm sizes as a stylised fact. It is not the purpose of this research to revisit, or even to delve deeply into, the question of which probability distribution best describes the empirical firm size distribution. We certainly will not attempt to fine-tune the models to match a particular shape or slope of an empirical distribution, or evaluate the models on their precise correspondence to empirical data. What is of interest is the fact that a highly abstracted model, the variable effort model, is capable of producing results in accordance with a stylised fact when many mainstream models either do not address the question or cannot reproduce it. We seek in this thesis to understand why and how this abstract model produces a skew distribution of firm sizes, and thus we hope to evaluate whether this model represents an amusing toy or a valuable tool for comprehending one of the most widespread empirical regularities in economics.

In the next chapter of this thesis, we will review the history of the skew distribution of firm sizes in economics, and also review several skew probability distributions including the power law. In Chapter 3, we will discuss ABM in general and how our particular models are actually implemented in code. The following four chapters will relate to the models and their simulation results.

The first model, and the one which will be dealt with in most detail, is a variable-effort model and is presented in Chapter 4 with additional extensions to the model in Chapter 5. The variable-effort model is the most typical agent-based model we will see here, agents are heterogeneous and the formation of firms will be strictly endogenous. The later two models are based on more conventional approaches to the firm size distribution, and have exogenous elements contributing to the resultant firm size distribution. In Chapter 6 we implement an exogenous birth model which is inspired by some of the modifications to Gibrat's Law intended to address the problem of increasing variance over time, an issue we will discuss in Secion 2.3. Our final

model in Chapter 7 tries to link the traditional cost curve analysis approach to the agent-based approach. Finally, we will discuss the implications of the three models and conclusions in Chapter 8. Source code is contained in an appendix.

Chapter 2

Skew Probability Distributions, Gibrat's Law and Scale Invariance

In this chapter we survey two well-known skew probability distributions along with two less well-known distributions. Following this discussion, we will consider Gibrat's Law and some of the models which descended from Gibrat's original economic interpretation of a generating function for the lognormal probability distribution. Finally, we discuss some of the theoretical implications of the concept of scale invariance, and how they may apply to this research.

2.1 Skew Probability Distributions

The power law and lognormal distributions are the most well-known skew distributions, they have been competing for "market share" amongst researchers interested in skew distributions since there have been researchers interested in skew distributions, but neither theoretical nor empirical work has managed to settle the apparent dispute between which of them is "the" skew distribution. As we shall see in this section, they are in fact related to each other in much deeper ways than just their superficial resemblance. We

also address two other distributions in this section which have been proposed in recent literature as alternatives to the power law. Many datasets which are usually described as following a power law (linear in double logarithmic coordinates) do in fact show some curvature. The lognormal distribution allows for curvature, as do the two other alternatives discussed here, the stretched exponential and the parabolic fractal. These four distributions are by no means an exhaustive list of the power law like distributions, a good survey can be found in a recent paper by Mitzenmacher (Mitzenmacher 2003).

Although we will present these four probability distributions and discuss how their parameters may be calculated, in practice in this thesis we will only consider one distribution, the discrete power law. We will not perform any statistical testing, but we will calculate the value of the power law slope parameter k as a summary statistic. The power law is the simplest distribution available to us for summarising skew data, with a single parameter which has an intuitive meaning. Although there is curvature visible in many skew datasets and the power law is strictly linear, it is still a useful first approximation. The slope of the power law distribution has been proposed as a measure of market concentration, most notably by Simon and Ijiri who argue that

> ...frequently used measures of concentration, like the Gini index, or the fraction of total industry sales that are accounted for by some fixed number of largest firms, do not have any clear theoretical foundation. It would seem more defensible to measure concentration by a parameter of the stochastic process that is being used to explain the data - for example the slope of the Pareto curve. (Ijiri and Simon 1977)[p. 13]

Steindl offers additional support for the significance of k

> It appears from the results of various authors that the Pareto coefficient is determined as the ratio of certain growth rates. In Simon's model it is the ratio of growth of the firm-population to the growth of the firm; in Wold-Whittle's model, the ratio

> of growth of wealth to the death rate of wealth-owners. These growth rates apparently represent the dissipative and the stabilizing tendencies in the process. This confirms the intuition of G. K. Zipf who regarded the Pareto coefficient as the expression of an equilibrium between counteracting forces. (Steindl 1965)

The focus of this paper is on the mechanisms which lead to skew distributions, not the precise shapes of those distributions. The thorny process of comparing the relative merits of probability distributions with differing assumptions, estimation techniques and numbers of parameters would be a significant research endeavour in itself, and is neither within the scope nor the time frame of this thesis. A further complication which warrants research is whether a simulated firm size distribution, that is a set of observations of firm sizes and their relative frequencies, consists of independent and identically distributed observations, since it arises from a closed system of a fixed number of agents who self-organize into firms. For these reasons, rather than go through the motions of an inappropriate statistical analysis, we prefer to set aside the issue of statistical testing in the belief that it would not add value to this analysis at this time. In this approach, we follow the example of numerous other authors including Herbert Simon

> A great deal has been written, at one time or another, as to whether a particular empirically observed distribution could, or could not, be approximated by a Yule distribution, the Pareto or the log normal. Such a question is difficult to answer, for several reasons. One reason . . . is that there does not really exist a body of statistical theory that tells one whether some data fit a particular extreme hypothesis. . . . Hence, we shall not be much concerned, in what follows, with significance tests, which are completely inappropriate for testing the fit of data to extreme models. (Ijiri and Simon 1977)[p. 4]

We will employ the following terminology to describe discrete data such as firm sizes. Observations $x_1, \ldots, x_n$ are sorted from largest to smallest so that the subscript i corresponds to the rank of the observation, the largest

Source	PDF	CDF	Rank
Gabaix	$\zeta + 1$	ζ	ζ (Rank is on y axis, Size on x axis)
Adamic	$a = 1 + k = 1 + \frac{1}{b}$	k	$b = \frac{1}{k}$ (Rank is on x axis, Size on y axis)

Table 2.1: The power law exponents used by two authors. $\zeta, k > 0$

observation being ranked first and the smallest n^{th}. Amongst our observations are $m \leq n$ unique values, and so we can convert our vector of observations into two vectors, the first of the unique values $F = (f_1, \ldots, f_m)$ and the second of the number of occurrences of each of the f_i, $C = (c_1, \ldots, c_m)$, so that $n = \sum_{i=1}^{m} c_i$.

For example, a vector of observations $X = (30, 5, 3, 2, 2, 1, 1, 1)$ would have corresponding $F = (30, 5, 3, 2, 1)$ and $C = (1, 1, 1, 2, 3)$. To make this more concrete, consider the vector X to be a vector of firm sizes. There is one firm of size 30, one firm of size 5, one firm of size 3, two firms of size 3 and three firms of size 1. The firm of size 30, the largest, has first rank. The three firms of size 1 are ranked 6th, 7th and 8th.

2.1.1 The Power Law and Lognormal Distributions

Power Law

The probability density function (PDF) of the power law distribution is given by

$$f(x) \propto x^{-(k+1)}, \tag{2.1}$$

for $x > 0$, where the parameter $k > 0$ determines the steepness of the slope. The probability mass function in the upper tail is

$$\bar{F}(x) \propto x^{-k}. \tag{2.2}$$

Taking logarithms of $y = x^{-k}$ gives

$$\log y = -k \log x, \tag{2.3}$$

which illustrates the power law's trademark linear relationship in double logarithmic coordinates (Adamic 2006).

It can be confusing to compare the terminology of different authors who may be referring to one of three distinct types of plot, each of which may show a roughly straight line which could be referred to as having a "power law slope". Table 2.1 compares the parameters of two such authors, Gabaix and Adamic. The power law slope has a canonical value of $a = 2$, or equivalently $k = b = 1$ (Adamic 2006). Gabaix, as we see in Section 2.3.2, also defines a standard case in which $\zeta = 1$.

Lognormal

The PDF of the lognormal distribution is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}x} \exp\left[-\frac{(\ln x - \mu)^2}{2\sigma^2}\right], \tag{2.4}$$

(Bi, Faloutsos, and Korn 2001).

Discrete Forms

The power law and lognormal can be discretised, and must be when applying them to discrete data.[1] The discrete probability function (point-mass function) of the power law is given by

$$p(x) = \frac{x^{-(k+1)}}{\sum_{j=1}^{\infty} j^{-(k+1)}}, \tag{2.5}$$

for x a positive integer, and can also be written more succinctly as

$$p(x) = \frac{x^{-(k+1)}}{\zeta(k+1)}, \tag{2.6}$$

where $\zeta(n) = \sum_{j=1}^{\infty} \frac{1}{j^n} = \sum_{j=1}^{\infty} j^{-n}$ is the Riemann zeta function (Weisstein 2005).

The distribution parameter k can be determined by maximum likelihood. Assuming independent, identically distributed data the likelihood function is

[1] Bi et al argue that any observations measured with finite precision should be considered discrete (Bi, Faloutsos, and Korn 2001).

$$L(k) = \prod_{i=1}^{n} P(x_i) = \prod_{i=1}^{n} \frac{x_i^{-(k+1)}}{\zeta(k+1)}. \tag{2.7}$$

The log-likelihood function is given by

$$l(k) = -(k+1)\sum_{i=1}^{n} \log x_i - n\log\left[\zeta(k+1)\right], \tag{2.8}$$

or equivalently in terms of F and C

$$l(k) = -(k+1)\sum_{i=1}^{m} c_i \log f_i - n\log\left[\zeta(k+1)\right]. \tag{2.9}$$

These equations having been derived following the example of Bi et al (Bi, Faloutsos, and Korn 2001) who provide the following discretised version of the lognormal distribution, which they refer to as the Discrete Gaussian Exponential or DGX

$$p(x) = \frac{A(\mu,\sigma)}{x} \exp\left[-\frac{(\ln x - \mu)^2}{2\sigma^2}\right], \tag{2.10}$$

for x a positive integer, where the normalisation constant A is given by

$$A(\mu,\sigma) = \left\{\sum_{j=1}^{\infty} \frac{1}{j} \exp\left[-\frac{(\ln j - \mu)^2}{2\sigma^2}\right]\right\}^{-1}. \tag{2.11}$$

Assuming independent, identically distributed data the log-likelihood function is

$$l(\mu,\sigma) = n\ln A(\mu,\sigma) - \sum_{i=1}^{n}\left[\ln x_i + \frac{(\ln x_i - \mu)^2}{2\sigma^2}\right], \tag{2.12}$$

or equivalently in terms of F and C

$$l(\mu,\sigma) = n\ln A(\mu,\sigma) - \sum_{i=1}^{m} c_i\left[\ln f_i + \frac{(\ln f_i - \mu)^2}{2\sigma^2}\right]. \tag{2.13}$$

The parameter k for the discrete power law, and the parameters μ and σ can be determined by numerically maximising the respective log-likelihood expressions. There is an unfortunate catch-22 in this technique, which is that the same parameter values which maximise the log-likelihood also cause the summations within those expressions to converge extremely slowly. For certain data sets, the numerical computation becomes extremely time consuming, to the point of being prohibitive in exploratory research.

2.1.2 The Fractal Distributions

Both the stretched exponential and parabolic fractal distributions are defined with reference to a type of plot known as a fractal display, in which observation values (the vector F defined previously) are plotted against their corresponding ranks with double logarithmic axes.

Stretched Exponential

The stretched exponential distribution is a generalisation of the exponential distribution (Laherrère and Sornette 1998). The PDF is defined as

$$f(x) = c\left(\frac{x^{c-1}}{x_o^c}\right)\exp\left[-\left(\frac{x}{x_0}\right)^c\right], \tag{2.14}$$

with Cumulative Distribution Function (CDF)

$$F(x) = P(X \leq x) = \exp\left[-\left(\frac{x}{x_0}\right)^c\right], \tag{2.15}$$

for $0 < c \leq 1$. When $c = 1$ this reduces to the exponential distribution (Laherrère and Sornette 1998). The Stretched Exponential produces a straight line when the natural logarithm of the rank is plotted against observed values raised to the power c

$$x_i^c = -a \ln i + b. \tag{2.16}$$

The three parameters of the distribution are a, b and c with $x_0 = a^{\frac{1}{c}}$. The proposers provide no algorithm for fitting the Stretched Exponential. Thus far the only practical method we have found is one of brute force. Allow c to take each of the values in $(0.001, 0.002, \ldots, 0.999, 1.000)$ (or the required search precision) and proceed to fit the linear model specified in Equation 2.16 to the vector of observations $x_1^c, \ldots, x_n^c$. Thus there will be, in this case, $1,000$ linear models fitted. Choose the value of c which corresponds to the highest regression R^2, and a and b are then obtained from the corresponding linear model.

Parabolic Fractal

Like the lognormal distribution, the parabolic fractal distribution is a 2nd order polynomial extension of the linear power law (Laherrère and Sornette 1998)[2]. Both distributions take the shape of a downward-facing parabola, the lognormal does so in a standard double logarithmic probability plot whilst the parabolic fractal does so in the fractal display, that is a double logarithmic rank-frequency plot. The parabolic fractal is defined in terms of observations by

$$\log x_i = \log x_1 - a \log i - b(\log i)^2. \tag{2.17}$$

When $b = 0$, this reduces to the power law. Since a concave parabola has a maximum value, the theoretical maximum observation (regardless of sample size) can be calculated

$$x_{max} = x_1 e^{\left(\frac{a^2}{4b}\right)}. \tag{2.18}$$

The parabolic fractal is fit using linear regression on $\log i$ and $(\log i)^2$.

2.2 Some Example Datasets

We will proceed to fit each of the four distributions to by the methods described. We can compare the power law and lognormal plot with one other using an error statistic, denoted ERR, defined by

$$ERR = \sum_{i=1}^{m} \frac{[nP(f_i) - c_i]^2}{m}, \tag{2.19}$$

which is a simple extension of the Mean Squared Error. We define a similar error statistic for Rank-Frequency data, denoted ERR_R, which we will use to compare the stretched exponential and parabolic fractal distributions, by

[2]The original article proposing the parabolic fractal distribution is in French, the parabolic fractal is discussed along with the stretched exponential distribution in this paper.

$$ERR_R = \sum_{i=1}^{n} \frac{[F(i) - x_i]^2}{n}, \tag{2.20}$$

where $F(i)$ is the predicted frequency for the observation of rank i.

2.2.1 Words in Beowulf

Beowulf, one of the earliest surviving poems in English, was a source text for Zipf's study of the frequency with which words appear in the written language (Zipf 1965). The text of Beowulf (Anonymous 1000) was analysed and a word count list (concordance) was prepared, the start of which is shown in Table 2.2.

Frequency	Word
1	ABANDONED
1	ABEL
2	ABIDE
1	ABJECT
3	ABLE
4	ABODE
6	ABOUT
2	ABOVE
1	ABROAD
2	ACCURSED

Table 2.2: Excerpt from concordance of Beowulf.

The concordance, which lists individual words, was then collated into a distribution of word appearance frequency which is shown in Table 2.3. In this table we have word frequencies in the first column, and the number/count of words which appear with said frequency in the second column. Rank is also given for the highest ranked observations. The interpretation of the first row of this table is that there are 1,611 words which appear only once within the text of Beowulf. The most frequent word is "the", which appears 1,587 times in the text. The full distribution of the Frequency of

word appearance and Count of each frequency is plotted in Figure 2.1, along with the fitted discrete power law and discrete lognormal.[3]

Frequency	Count	Rank	Word
1	1611		
2	548		
3	293		
4	180		
5	115		
...	...		
276	1	5	WITH
321	1	4	THAT
408	1	3	HIS
636	1	2	AND
1587	1	1	THE

Table 2.3: Excerpts from the distribution of word frequency appearance in Beowulf.

We see that the ERR statistic is lower for the lognormal than the power law, which is to be expected since the lognormal distribution is a generalisation of the power law. The stretched exponential and parabolic fractal are shown plotted in Figure 2.2. For this dataset, they give similar shaped fitted curves and similar error statistics. We see that both curves miss the handful of highest ranked observations by a considerable amount. This may be a feature of the ranked data which has a data point for each observation, rather than each observed value, and so places more of an emphasis on the common small events by listing them separately whereas in the Frequency-Count data, these small events are aggregated together.

2.2.2 Genera and Species of Snake

The number of species in a genus, for a family of plants or animals, has a skew distribution. We consider two similar datasets here. The first dataset

[3] Figures are located at the end of each section in this thesis.

is taken from Yule's original paper (Yule 1925), which was quoted from an earlier work by Willis, who in turn collated the data from the "Catalogue of the Snakes in the British Museum", by G. A. Boulenger, published in 1893. We also have an updated version with 2005 data, which we collated from the online EMBL Reptile Database (Uetz and Heidelberg 2005), a process which must have been almost infinitely simpler today than in Willis' day. There are 293 genera and 1,475 species in the 1893 dataset, and 463 genera and 3,002 species in the 2005 dataset. The data is plotted and the discrete power law and lognormal are fitted in Figure 2.3 (1893 data) and Figure 2.5 (2005 data). The stretched exponential and parabolic fractal are shown in Figure 2.4 (1893) and Figure 2.6 (2005).

For both 2005 and 1893 we see that the discrete lognormal has a lower error statistic, as we expect. The stretched exponential is also a better fit in both cases, and in the 2005 data it seems to match even the largest events. The parabolic fractal, in addition to having a poor fit, also has a positive coefficient for $\log i$ in the 2005 data which violates its specification.

2.2.3 Cities in the United States

Here we look at the distribution of population amongst cities in the United States with over 100,000 people. The discrete lognormal again outperforms the power law, as shown in Figure 2.7. The parabolic fractal in this instance has a slightly lower error statistic than the stretched exponential, but this is probably not a significant difference, see Figure 2.8.

2.2.4 Discussion

The discrete lognormal and discrete power law were fit using maximum likelihood, and their distribution functions explicitly acknowledged the discrete nature of the data. The stretched exponential and parabolic fractal were fit using linear regression, implicitly assuming continuous data, and were fit in the rank-frequency plot rather than a frequency-count or frequency-pdf plot. It is not clear at this point whether these two approaches will turn out to be complementary, each highlighting different and useful aspects of the data, or whether one will emerge to be "correct".

For the datasets considered here, the discrete lognormal was a better fit than the discrete power law, which was expected as the lognormal is a generalization of the power law and has more parameters. The parabolic fractal proved problematic as it should be strictly decreasing, but for several datasets the fit produced by linear regression led to negative values for the a coefficient. The stretched exponential did not have this difficulty, and it had a better or comparable error statistic to the parabolic fractal in the examples considered here, so where a Rank-Frequency-based distribution is desired the stretched exponential seems preferable.

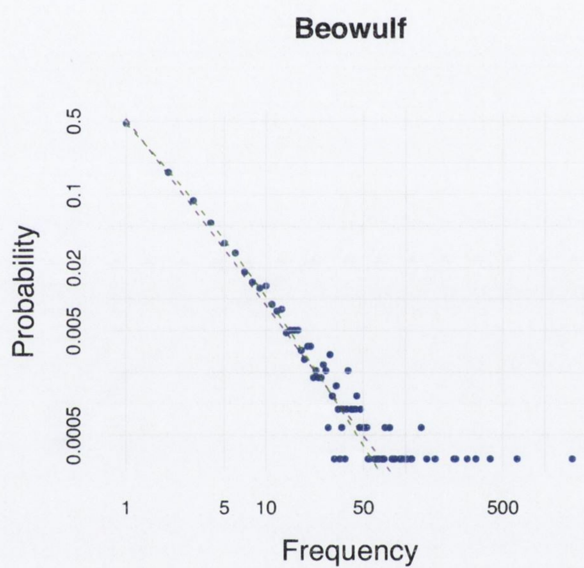


Figure 2.1: Frequency of word appearance in Beowulf. The fitted discrete power law is shown in purple, values are $k = 0.780$, $ERR = 358.0$. The discrete lognormal is shown in green, values are $\mu = -2.231$, $\sigma = 2.275$, $ERR = 12.3$

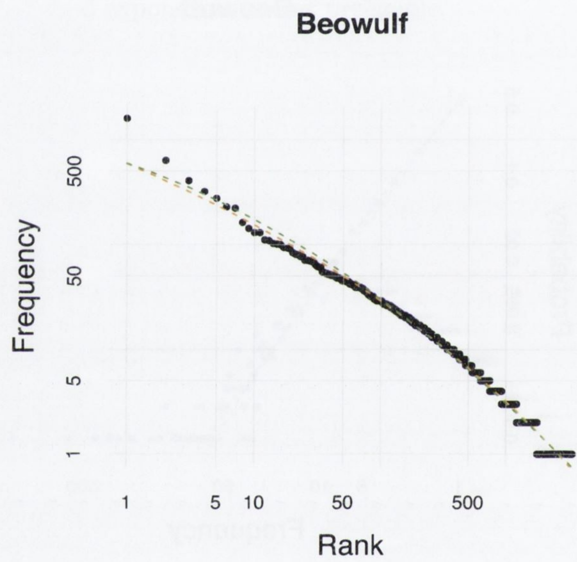


Figure 2.2: Frequency of word appearance in Beowulf and Rank of each Frequency. Fitted stretched exponential (orange) with $a = 0.189$, $b = 2.489$, $c = 0.143$ and $ERR_R = 318.1$. Fitted parabolic fractal (green) with $a = 0.423$, $b = 0.113$ and $ERR_R = 312.5$.

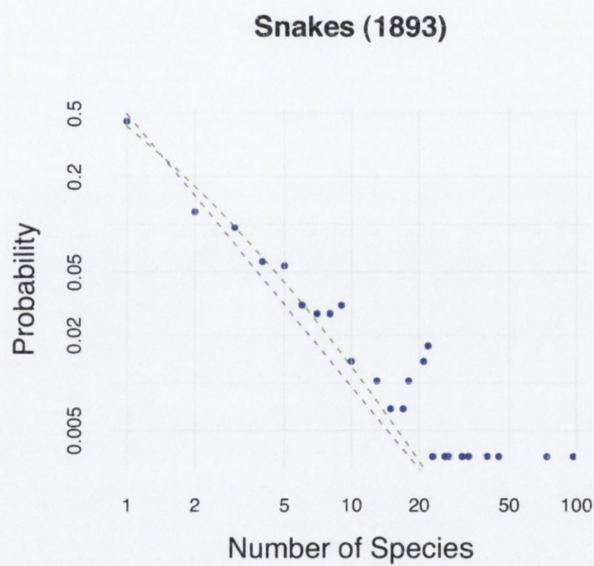


Figure 2.3: Number of species per genus of snake. The fitted discrete power law is shown in purple, values are $k = 0.722$, $ERR = 19.9$. discrete lognormal is shown in green, values are $\mu = -0.506$, $\sigma = 1.790$, $ERR = 16.3$

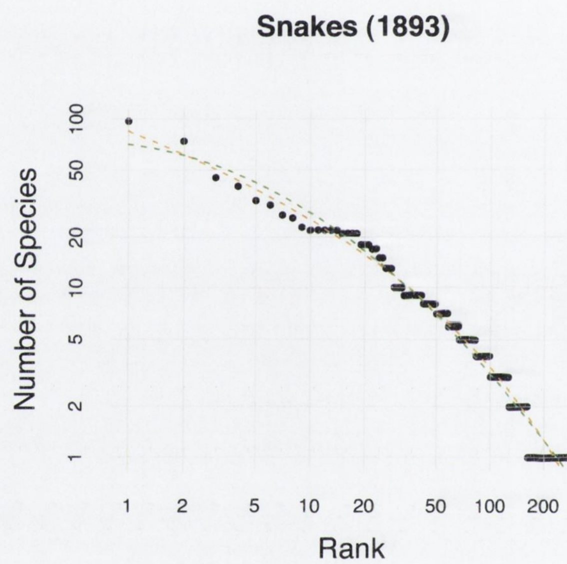


Figure 2.4: Number of species per genus of snake and rank of each genus. Fitted stretched exponential (orange) with $a = 0.580$, $b = 4.153$, $c = 0.321$ and $ERR_R = 1.9$. Fitted parabolic fractal (green) with $a = 0.131$, $b = 0.273$ and $ERR_R = 4.8$.

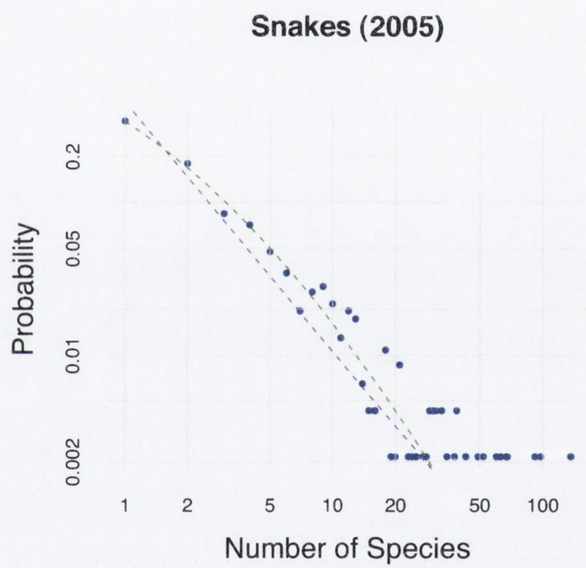


Figure 2.5: Number of species per genus of snake. The fitted discrete power law is shown in purple, values are $k = 0.631$, $ERR = 78.8$. discrete lognormal is shown in green, values are $\mu = 0.330$, $\sigma = 1.601$, $ERR = 5.0$

.

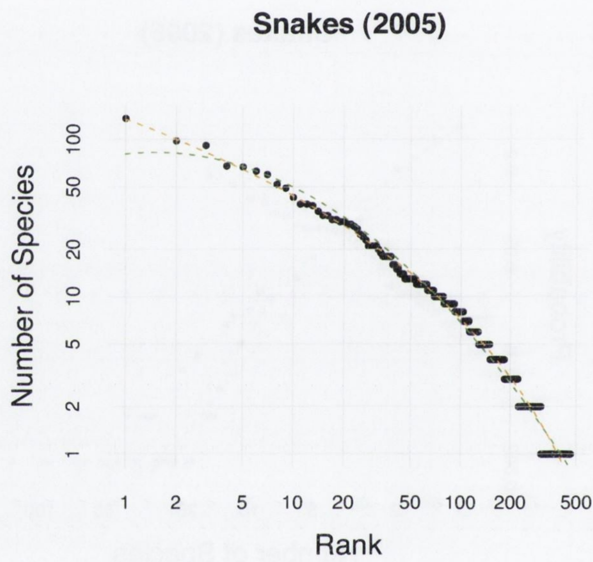


Figure 2.6: Number of species per genus of snake and rank of each genus. Fitted stretched exponential (orange) with $a = 0.687$, $b = 5.089$, $c = 0.332$ and $ERR_R = 0.6$. Fitted parabolic fractal (green) with $a = -0.116$, $b = 0.326$ and $ERR_R = 8.9$.

Figure 2.7: Frequency of city size. The fitted discrete power law is shown in purple, values are $k = 0.898$, $ERR = 123.0$. discrete lognormal is shown in green, values are $\mu = 0.131$, $\sigma = 1.144$, $ERR = 28.0$

.

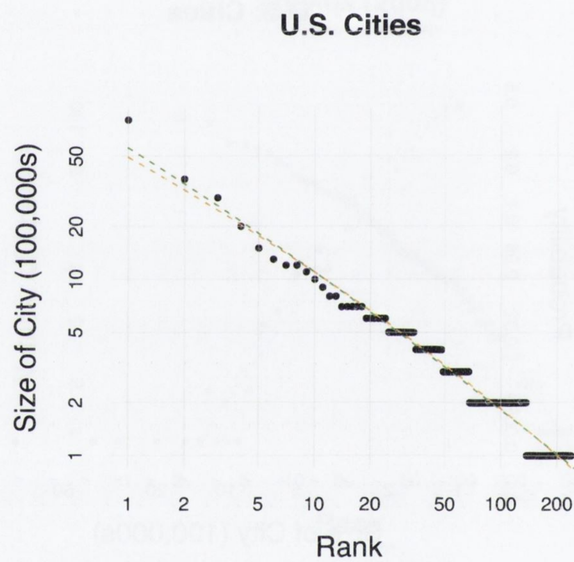


Figure 2.8: Frequency of city size and rank of each size. Fitted stretched exponential (orange) with $a = 0.094$, $b = 1.499$, $c = 0.104$ and $ERR_R = 4.4$. Fitted parabolic fractal (green) with $a = 0.666$, $b = 0.039$ and $ERR_R = 2.8$.

2.3 Gibrat's Law

> ...if the very same regularity appears among diverse phenomena having no obvious common mechanism, then chance operating through the laws of probability becomes a plausible candidate for explaining that regularity. Hence, the theoretical models we shall examine are stochastic models yielding the observed size distributions as their steady-state equilibria. (Ijiri and Simon 1977)[p. 3]

Inspired by a simple mathematical model and his compelling empirical observations, Gibrat's Law of Proportional Effect spawned "one of the most important strands in the literature on market structure" (Sutton 1997)[p40]. The model, as explained by Steindl, is based on the assumption that "growth in proportion to size is a random variable with a given distribution which is considered constant in time" (Steindl 1965)[p. 30], or

$$X_t - X_{t-1} = \varepsilon_t X_{t-1}, \tag{2.21}$$

and

$$X_t = (1 + \varepsilon_t)X_{t-1} = X_0(1 + \varepsilon_1)(1 + \varepsilon_2)\ldots(1 + \varepsilon_t), \tag{2.22}$$

where ε_t is an independent, identically distributed random variable representing the proportional growth rate and X_t represents the size of the object (firm size, firm's capital) we are interested in (Steindl 1965)[p. 30].

That is, the growth rate ε_t is random and does not depend upon the size of the firm, equivalently absolute growth is proportional to the size of the firm. It follows from the Central Limit Theorem that the logarithm of X_t is normally distributed with mean mt and variance $\sigma^2 t$ (Steindl 1965)[p. 30]. Gibrat's economic interpretation of this generating mechanism for the lognormal distribution is that the growth rate of firms is independent of their size. Gibrat presented empirical evidence showing that firm sizes in various sectors of the French economy do seem to follow a lognormal distribution (Sutton 1997).

An implication of Gibrat's model is that the variance of firm sizes, $\sigma^2 t$, will increase over time, eventually tending to infinity. Kalecki, who observed that this was unsatisfactory both theoretically, "for a priori reasons it is clear that changes in the standard deviation of the logarithm of a given variate are to a great extent determined by *economic* forces", and empirically, "no tendency for such an increase is apparent (for instance in distribution of incomes)" (Kalecki 1945)[p. 162], took the straightforward mathematical approach, held variance constant, and therefore derived an assumption of a negative second derivative, which solves this particular dilemma but at the cost of introducing a rather arbitrary assumption. Champernowne, whose work was inspired by both Pareto and Gibrat (Champernowne 1952), uses a Markov chain approach and brings birth and death into the model. His solution to the increasing variance problem is to assume that the expected value of the random growth rate is negative, which can be interpreted as assuming that older, high-income workers die off and are replaced by young, low-income workers (Steindl 1965). Champernowne also assumes a minimum level of income which is another method of ensuring the variance does not become infinite (Gabaix 1999)[p. 759]. Champernowne's model, which is actually an income model but which can also be applied to firm sizes, leads to a power law distribution rather than the lognormal distribution. In the following sections we consider two additional approaches to the issue of the increasing variance of firm sizes: those of Simon and Gabaix.

2.3.1 Simon's Approach

In his paper on skew distribution functions, Herbert Simon presents a modernisation of what is now widely viewed as the classic work in the field, by G. Udny Yule (Simon 1955). Simon's paper does not mention Gibrat at all, or firm sizes, but he does discuss the relationship of his work to that of Champernowne. Yule's original paper, which as an aside gives an interesting snapshot of the state of the theory of Darwinian evolution at the time, presented a mathematical model to explain the skew distribution of the number of species in a genus of plants or animals, evidently a stylised fact in biology (or on its way to becoming one) at the time (Yule 1925).

The resulting distribution, known as the Yule distribution (seemingly thus christened by Simon (Simon 1955)[p. 426]), is similar to the power law and can be approximated by it. Simon, noting that he, unlike Yule, had access to the "modern theory of stochastic processes", was able to make weaker assumptions and put forward a model intended to address a variety of well-known empirical skew distributions in various fields

> The empirical distributions to which we shall refer specifically are: (A) distributions of words in prose samples by their frequency of occurrence, (B) distributions of scientists by number of papers published, (C) distributions of cities by population, (D) distributions of incomes by size, and (E) distributions of biological genera by number of species. (Simon 1955)[p. 425]

Simon's model is extremely intuitive. Imagine we are undertaking an analysis of word count frequencies in Beowulf. We have already read the first k words and we denote $f(i,k)$ the number of *different* words that have occurred exactly i times thus far. Simon provides two possibilities for the $k+1^{th}$ word. With constant probability α, the $k+1^{th}$ word will be a new word, not present in the first k words. With probability $1-\alpha$, therefore, the $k+1^{th}$ word is one of the existing words, and the probability that it is a word that has already appeared exactly i times is proportional to $if(i,k)$. These behaviours can be referred to as *birth* and *proportional growth*, since the probability of repeating an existing word is proportional to the existing popularity of the word. Later in the paper, Simon translates his model in terms of income: "We picture the stream of income as a sequence of dollars allocated probabilistically to the recipients." (Simon 1955)[p. 438].

This model leads to the Yule distribution, given by

$$f(i) = AB(i, \rho+1), \tag{2.23}$$

where $A > 0$, $\rho > 0$ are constants and $B(i, \rho+1)$ is the Beta function of $i, \rho+1$, defined as

$$B(i, \rho+1) = \int_0^1 \lambda^{i-1}(1-\lambda)^{\rho} d\lambda \qquad (0 < i; 0 < \rho < \infty). \tag{2.24}$$

Simon shows that as $i \to \infty$, this distribution approaches

$$f(i) \sim \Gamma(\rho + 1) i^{-(\rho+1)} \tag{2.25}$$

where Γ is the Gamma function, and that therefore the Yule distribution approximates the power law distribution which Simon defines as

$$f(i) = \left(\frac{a}{i^k}\right) b^i, \tag{2.26}$$

where a, b and k are constants with b approximately equal to 1.

In his discussion of the application of this model to income distribution, Simon notes that Champernowne's model, although very different in appearance and inspiration, satisfies these assumptions and that therefore "the underlying structure is the same" (Simon 1955)[p. 438]. Simon's model incorporates the elements of *minimum size* and *birth and death*, death being introduced later in the paper and resulting in a very similar generalised Yule distribution, although it does not appear that these concepts were introduced specifically for the purpose of addressing the variance issue, rather they seem to be an natural feature of the model.

Simon updated and reframed his model specifically in terms of firms in the modern classic, "Skew Distributions and the Sizes of Business Firms" (Ijiri and Simon 1977). The *income stream* metaphor is now expressed in terms of new *opportunities* of which arise over time. An *opportunity* is simply a unit increase in the size of the economy, it can take the form of a new firm entering the economy or a unit increase in the size of an existing firm. The probability of a new entrant firm taking up the opportunity is a constant α, and, therefore, with probability $1 - \alpha$ an existing firm takes up the new opportunity, the probability of a particular firm doing so being proportional to the size of that firm.

2.3.2 Gabaix's Approach

Gabaix, in a paper on city sizes, proves that the power law distribution is the unique steady state distribution possible in a system in which Gibrat's Law holds (Gabaix 1999). He expresses Gibrat's Law as "homogeneity of

growth processes", i.e. every city's stochastic growth process has the same mean and variance as every other city. He further shows that the slope of the power law must be 1 in the standard case, and then gives conditions under which it may differ from 1.

> The existence of a power law can be thought of as due to a simple physical principle: scale invariance. Because the growth process is the same at all scales, the final distribution process should be scale-invariance. This forces it to be a power law. To see why the exponent of the power law is 1, a concrete situation might help. Suppose that cities are on a discrete grid, and that at each point in time a city might double, or halve in size. Because we must satisfy the constraint that the average size (understood as share of the total population) be constant, the probability of doubling has to be $\frac{1}{3}$, and the probability of halving $\frac{2}{3}$ (the expected growth is $\frac{1}{3} \cdot 2 + \frac{2}{3} \cdot \frac{1}{2} - 1 = 0$). To see how the number of cities of a given size can be constant, take a size S. One can quickly convince oneself that the number of cities of size $2S$ should be half the number of cities of size S, and the number of cities of size $\frac{S}{2}$ should be double. This is precisely an expression of Zipf's law. (Gabaix 1999)[pp. 744-745]

Gabaix illustrates the physical mechanism at work behind his mathematical reasoning with an economic example in which young workers migrate to a city of their choice, their utility being based upon wages and urban amenities, where they will remain for the rest of their lives. In physical terms, he describes his model as incorporating reflected Brownian motion, with a barrier which prevents cities falling below an exogenous minimum size. Gabaix shows that a power law distribution with an exponent of 1 holds provided the "appearance rate of new cities ν is lower than the growth rate of existing cities γ" (Gabaix 1999)[p. 751]. If, instead, $\nu > \gamma$, a power law distribution with slope greater than 1 will occur. Finally, Gabaix explains the irregularity that small and medium cities often have a power law exponent of less than 1 by showing that a relatively higher variance can account for this.

> For cities we have $\nu < \gamma$, so that the resulting exponent does not depend on the details of the country's situation: it is just 1, or very close to it. For incomes we have $\nu > \gamma$, in which case the exponent depends finely on the situation's parameters, ν, γ, σ, which explains why [the power law exponent] loses its constancy across economic structures and has cross-sectional and possibly time series variations. (Gabaix 1999)[p. 760]

In particular, when $\nu > \gamma$ the power law exponent ζ can be calculated as the positive root of

$$h(\zeta) = \zeta^2 - (1 - 2\frac{\gamma}{\sigma^2})\zeta - 2\frac{\nu}{\sigma^2}, \tag{2.27}$$

where σ^2 is the variance of γ.

2.4 Scale Invariance and Power Laws

The power law distribution is closely tied to the concept of *scale invariance*, as we have just seen in our discussion of Gabaix's model. "Speaking about a (material or mathematical) object, scale invariance refers to its invariance over changes of scales of observation." (Dubrulle, Graner, and Sornette 1997)[p. 2]. A fractal, which looks similar at any level of magnification, has no obviously "correct" scale and is a canonical visual example of scale invariance. While it is possible to calculate the mean size of a firm in the Irish economy, a firm which happens to be that size is in no way a prototypical or stylised example of a firm. Unlike in normally distributed systems, the average (mean) does not imply average (typical). It is quite trivial to establish the link between scale invariance and the power law. If an observable variable $O(l)$ changes by factor $\mu(\lambda)$ when the scale is changed by a factor λ, then the observable must have solution $O(l) = Cl^\alpha$ (Dubrulle, Graner, and Sornette 1997)[p. 3], i.e. a power law, with $\alpha = \frac{-\ln \mu}{\ln \lambda}$.

Power laws also have a deeper link with the concept of scale invariance, and the idea of a phase transition. In physics, researchers trying to understand phase transitions, for example water freezing and thereby transforming from a random, chaotic liquid into an ordered, structured solid, found

power laws in their observations of many phenomena around the point of transition. Kenneth Wilson developed the theory of *renormalization* by assuming that "in the vicinity of a critical point the laws of physics applied in an identical manner at all scales" (Barabási 2003)[p. 76], i.e. scale invariance, and was therefore able to predict the appearance of power laws.

> If power laws are the signature of systems in transition from chaos to order, what kind of transition is taking place in complex networks? If power laws appear in the vicinity of a critical point, what tunes real networks to their own critical point, allowing them to display a scale-free behavior? (Barabási 2003)[p. 78]

The boundary between chaos and order is a region in which complex structures can exist. Chaos itself is not a fertile environment, complex structures do require some regularity. While a complex structure *can* exist in a highly ordered environment, such a structure will be rigid and brittle, and will not survive in a competitive and changing world. Stuart Kauffman, in an epic yet readable tome, explores theoretical mathematical underpinnings of evolutionary biology and argues that complex systems "achieve a 'poised' state near the boundary between order and chaos, a state which optimizes the complexity of tasks the systems can perform and simultaneously optimizes evolvability." (Kauffman 1993)[p. 173]. Biological evolution, Kauffman argues, results from two ingredients: self-organisation and selection. Self-organisation creates complex entities, selection destroys those which are less fit, allowing those which survive to evolve further. The two forces, combined and iterated over time, give us the biological diversity we observe in nature, and also the ecological context in which those biological organisms exist.

Returning to economics, could the presence of power laws be a signature of the phase transition between order and chaos, and therefore be an indicator of a healthy self-organising complex dynamic system, ideally situated to perform complex tasks and remain resilient and adaptable in the face of inevitable internal and external shocks? Is the power law distribution of firm sizes and the highly inequitable Pareto distribution of income a fundamental property of a vibrant market-based diversified economy?

We can begin, not to answer these questions but to explore them, by experimenting with a self-organising system which generates a power law distribution.

Chapter 3

An Agent-Based Firms Framework

We face the paradox in agent-based modelling that in order to study our models in a simulated context, we must make seemingly arbitrary decisions about implementation. Should our agents make decisions sequentially or randomly? Synchronously or asynchronously? Should they be organised into a lattice structure with a geographic neighbourhood, or into an unstructured soup where neighbours are defined by a social network? How much information should agents have access to, how accurate should it be, how up-to-date should it be? All of these decisions, many of which are not specifically relevant to the economic question being considered, will have an impact on the simulation. How, then, do we isolate the pure effects of the economics from our simulation results? One answer is that we cannot, and furthermore we should not! The specifics of the implementation are a fundamental part of the simulation, and the strength of ABM is that it allows us to understand our economic models and ideas in a physical context (which is of course where they will be in the real world). What we can do is improve our understanding of the interrelationship between economics and environment by (1) treating the environmental assumptions as experimental variables and observing the impact that changes have on the results and (2) creating one or more standard reference environments and substituting a

variety (or a combination) of economic models into these well-understood environments, i.e. treating the economic models as experimental variables. In this thesis we will be doing both of these. We implement all of our models, having very different economic content, in a single framework so that we can focus on the differences between models, and we also experiment with many of the environmental parameters to learn about the impact they have on the results.

We consider a general framework in which agents within a simulation create, join and leave firms with no centralised co-ordination, that is, firms are created and structured endogenously. The economy consists of N agents, indexed by $i = 1, \ldots, N$. Agents in a firm are also referred to as employees. Whilst the population of agents remains fixed throughout the simulation, firms can be created or destroyed. A firm *dies* when its last employee leaves.

Agents are randomly assigned n *friends* during the initialisation of the simulation, the default is $n = 2$. Friendship is unidirectional, and so the friendship network forms a random directed graph (digraph). Agents can only obtain information about their own firm and their friends' firms, hence agents are said to be myopic. In principle, the more friends, the more information is available to agents although it is possible for some or all friends to be in the same firm, in which case there would be fewer unique observable firms. Whilst each agent has n friends, i.e. n outward links in the graph, the number of inward links is variable. For example, in Figure 3.1, no agents are linked to Agents 4, 8, 13 or 16 whilst five agents are linked to Agent 2.

Firms are created endogenously in this framework, agents create new firms and move between firms in order to maximise their utility. Agents are activated at random, and when activated will join the firm at which they can maximise their utility. The choices available to an activated agent are represented in Figure 3.2. An agent can remain in their current firm, join the firm of a friend, or create a new firm. If the agent creates a new firm then, initially, they will be the only employee of that firm. This is referred to as a *singleton* firm. The simulation is initialised with each agent in a singleton firm. An example showing agents grouped into firms with

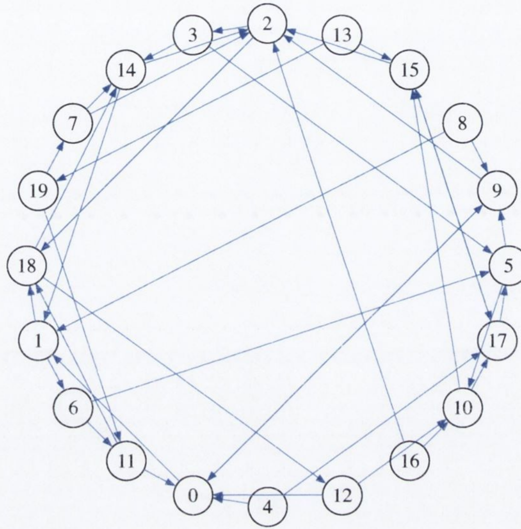


Figure 3.1: Example of a friendship network with 20 agents, $n = 2$.

friendship links between agents is shown in Figure 3.3.

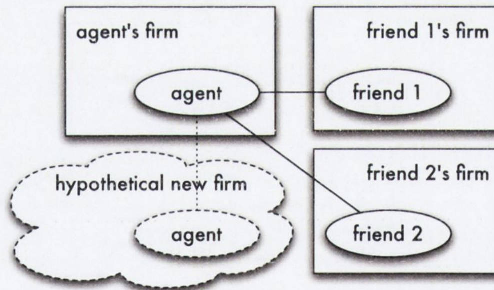


Figure 3.2: Options available to an agent when activated.

Whilst the precise nature of an agent's utility function and a firm's production function depend upon the specific model under investigation, we expect that a firm will have a production function which will depend somehow upon the labour of the firm's employees. Employee income will be derived from the firm's production, either explicitly or implicitly, and agent utility will depend in part or in full upon the level of income they receive from being an employee of a particular firm.

All models implemented using this framework will produce the same basic simulation output. Every simulation will produce a distribution of

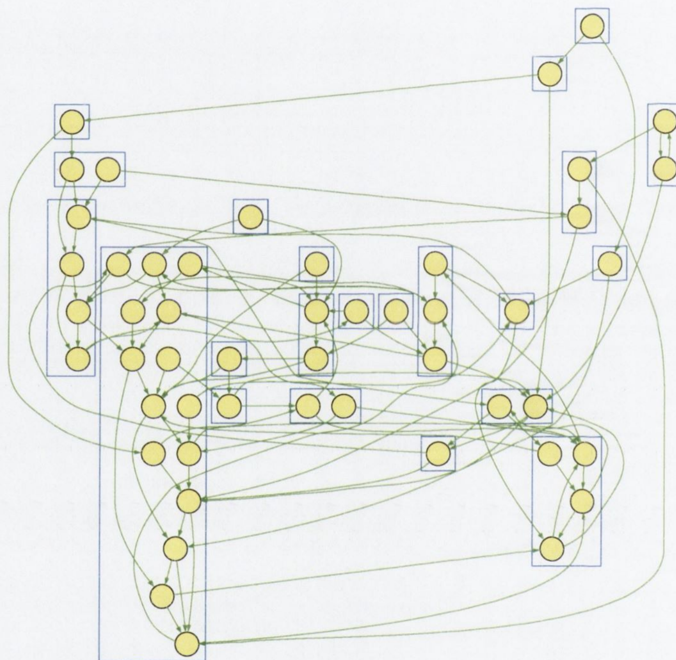


Figure 3.3: Friendship links (green directed lines) between agents (yellow circles) in firms (blue rectangles).

firm sizes, and distributions of firm output and agent utility, although in the latter two cases scales may differ between models. Additionally, each model will have input parameters and output statistics particular to it.

Chapter 4

A Variable Effort Model

4.1 Introduction

The model in this chapter is based upon a model by Robert Axtell (Axtell 1999). The friendship framework used for all the models is also taken from Axtell's original model, although in that model it was not a separate component as it is here. While Axtell's emphasis was on the analytical foundations of his model, with some descriptive statistics, here we will focus on gaining a more detailed understanding of the mechanism which generates the power law distribution of firm sizes. There are numerous graphs and plots in this thesis which have been carefully designed to reveal the inner workings of the simulations. Frequently this process involved returning to the simulation's source code and defining new data sources, then running fresh simulations and evaluating the usefulness of the new statistics. This iterative process took far more time than implementing the basic structure of the model itself, and it can be argued that the creative process of defining and extracting data from simulations is the real "work" of the technique.

The agents in this model have Cobb-Douglas utility functions for income and leisure. Income is derived from being an employee within a firm and receiving a share of that firm's output. Agents exert a level of effort between 0 and 1, e_i, and leisure is defined to be $1 - e_i$. The combined effort of employees within a firm is converted to firm output with a production

function that has increasing returns to scale. A firm j has a set of employees denoted J with cardinality N_j. The aggregate effort of agents within a firm j, E_j, is

$$E_j = \sum_{k \in J} e_k, \tag{4.1}$$

where e_k represents the effort level of Agent k. The corresponding output function is

$$O(E_j) = aE_j + b\left(E_j\right)^2, \tag{4.2}$$

where a and b are simulation parameters which determine the extent of the returns to scale. The default values are $a = 1$ and $b = 1$. We can make explicit a single employee's contribution to the output function

$$O(\hat{E}_j, e_i) = a\left(\hat{E}_j + e_i\right) + b\left(\hat{E}_j + e_i\right)^2, \tag{4.3}$$

where $\hat{E}_j$ represents the combined effort of all employees in Firm j excluding Agent i.

Agents in a firm share the firm's production equally, so for an agent in Firm j, income will be $\frac{O_j}{N_j}$. Agents have Cobb-Douglas utility functions, they seek both income and leisure and the parameter θ determines the relative value of income versus leisure. High θ agents obtain more utility from income, low θ agents prefer leisure. That is,

$$U_i = \left(\frac{O_j}{N_j}\right)^{\theta_i} (1 - e_i)^{1-\theta_i} \qquad \theta \in (0, 1). \tag{4.4}$$

In a singleton firm, where firm production and hence agent income is completely dependent upon the agent's own effort, the relationship between effort and utility is:

$$U_i = \left(ae_i + be_i^2\right)^{\theta_i} (1 - e_i)^{1-\theta_i}. \tag{4.5}$$

Each agent is randomly assigned a value of θ_i at initialization, chosen from a uniform distribution between 0 and 1.

The effort of the other employees in a firm is taken as given, and so using Equation 4.3 we can express utility as a function of individual effort

$$U_i(e_i) = \left(\frac{a\left(\hat{E}_j + e_i\right) + b\left(\hat{E}_j + e_i\right)^2}{N_j} \right)^{\theta_i} (1 - e_i)^{1-\theta_i} \qquad \theta \in (0, 1). \tag{4.6}$$

In the following sections we examine results from simulation runs. Unless otherwise specified, the simulation was run with $N = 10,000$ for 1,000 time periods. Some time series data is gathered at the end of each time period. For a small detail time period, a snapshot is taken after each agent is activated, resulting in $N = 10,000$ time series data points per period.

4.2 Case Studies

In this section we look at the histories of two firms, one very large firm and a more modest medium-sized firm (which is nonetheless much larger than the mean firm size of 2.7). These case studies allow us to gain an intuitive feel for the processes at work in this simulation before we move on to more impersonal aggregate statistics in later sections.

4.2.1 Large Firm Case Study

In this section we will examine the history of a firm which, starting as a singleton, grows to a size of 218 before declining. The firm was formed in Period 161 of the simulation by an agent with $\theta = 0.981$. This is a very high θ, close to 1, and we will discuss the relationship between firm size and the θ of the founding agent of a firm in Section 4.3.

In Figure 4.1, we see that the singleton entrepreneur starts out with a high level of effort (red), close to 1, but average effort decreases sharply as new agents join the firm. Utility (black) is high initially, as the employees enjoy the benefits of the increasing returns to scale, but it begins to decline along with total output after Period 168 (see Figure 4.2 for the time series of firm size and output). During Period 168, even though the firm size continues to increase, the total output of the firm begins to decrease due to the decline in average effort levels. In Period 169, the population of the firm peaks and begins to decrease.

In Figure 4.3, we can observe the utility of agents at various time periods during the life of the firm. The red points represent the employees at the end of Period 166. By this time the firm is well established, and about to begin its dramatic size increase. We see that almost all employees are "above" the purple curve which represents the level of utility which a singleton agent of given θ can achieve. Any agent above this line is better off than they would be as a singleton. No agent will choose to be below the purple curve as they have the option to create a singleton firm and thus achieve a utility level on the purple curve.

By the end of Period 167, the firm has increased to a population of 146 agents, and we see that agents are better off than they were in the previous time period. All but a handful of very high θ agents are above the purple curve now, and these few are on or just below the curve. By the following period, however, things have changed dramatically. As the olive green points show, agents with $\theta > 0.6$ are now below the purple curve. By Period 169, the population of the firm is collapsing as high θ agents leave. It is apparent that the longevity of the firm is due in part to the fact that agents are unable to leave whenever they wish. Agents can only move when they are activated, and with random activation an agent may have to wait for several time periods before being activated. In a model with sequential activation, the longest an agent would remain with a suboptimal firm would be 1 time period.

The mean θ of agents in the firm decreases throughout the life of the firm, as shown in Figure 4.1, caused initially by low θ agents joining the firm and later by higher θ agents leaving. In this example, the firm attracts some high θ agents and remains small until approximately period 166, when a large number of low θ agents "discover" the firm and the mean value of θ within the firm can be seen steadily declining. The ability of agents to free ride, to enjoy high levels of income based upon the effort of other agents, causes both the rise and fall of the firm. It attracts the large numbers, but those large numbers dilute the income of the high θ agents and cause them to leave. It is this process, on a large scale such as in this (unusually large and long-lived) firm, or on a small scale in a more typical small firm, which

drives the dynamic firm population in this model.

4.2.2 Medium Firm Case Study

In this section we perform an abbreviated case study with a medium-size firm. As shown in Figure 4.5, this firm achieves a maximum size of 13. The life span of this firm is shorter, it lasts for 6.0 periods as compared with the large firm's life span of 13.3 periods. (The relationship between firm size and age is explored in Figure 4.11.) However, despite these differences, a similar pattern of growth followed by decline is seen in the life of this firm.

Figure 4.4 shows a time series profile of the firm. We see that the founder remains a singleton for more than a period at the beginning of the firm's life. After the second member joins the firm, the firm grows quickly and then declines at a slightly more leisurely pace, see Figure 4.5. Output peaks before the firm size peaks, again suggesting that many agents are joining the firm and contributing little effort, they are joining with the intention of free-riding. (The output peak now suggests itself as a leading indicator of a firm's decline.) Although the plot in Figure 4.6 is more sparse than its counterpart in the previous section, we can still observe the relationship between the singleton optimum utility curve and the declining slope of the agents' utility-θ "line".

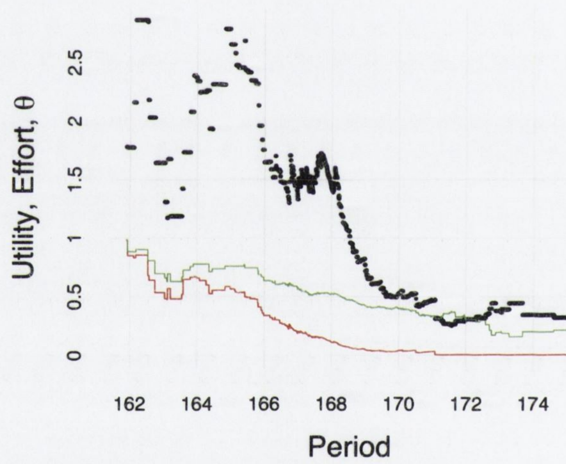


Figure 4.1: Average employee utility (black), effort (red) and θ (green) in the large case study firm over time.

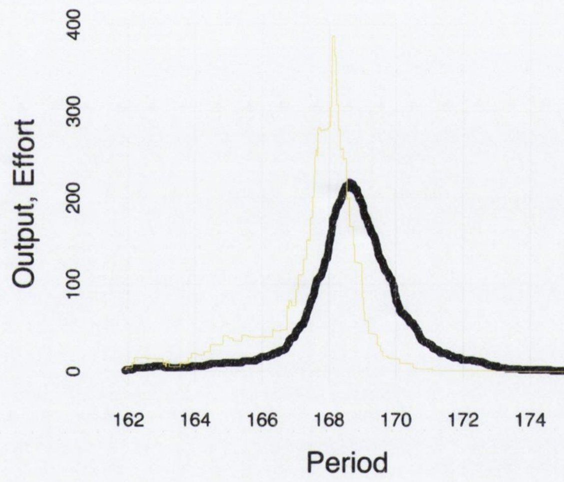


Figure 4.2: Firm size (black) and total output (yellow) in the large case study firm over time.

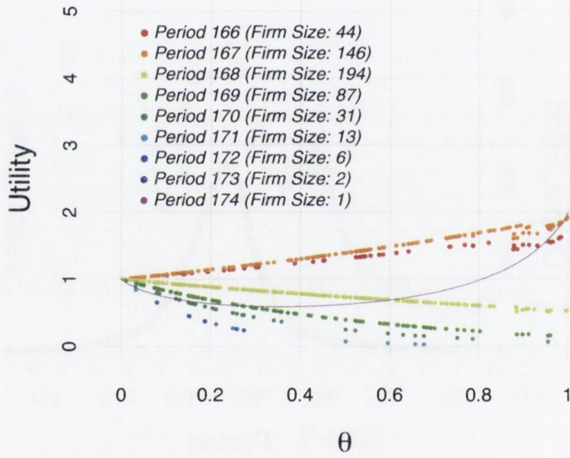


Figure 4.3: Utility of members of the large case study firm at the end of the indicated time periods, plotted against their θ.

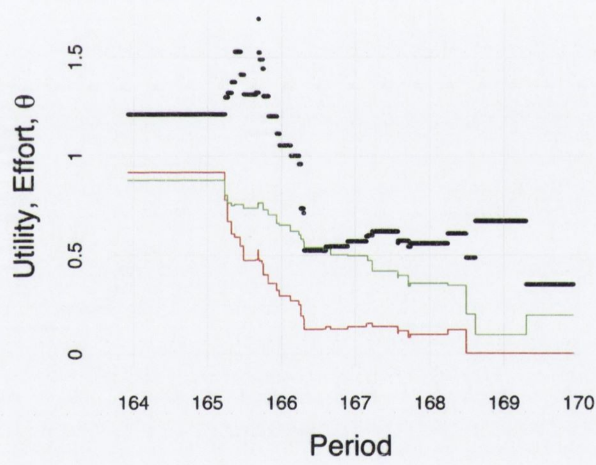


Figure 4.4: Average employee utility (black), effort (red) and θ (green) in the medium case study firm over time.

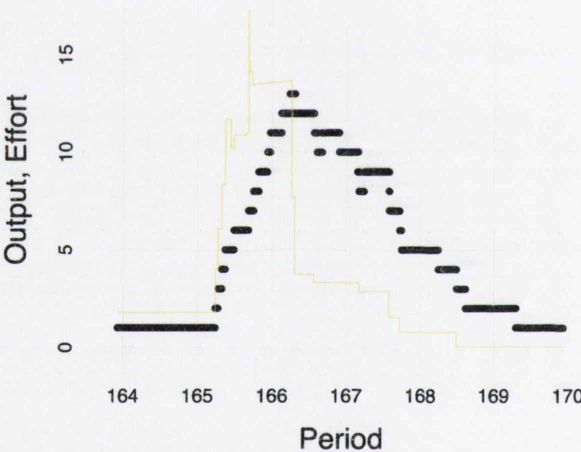


Figure 4.5: Firm size (black) and total output (yellow) over time in the medium case study firm over time.

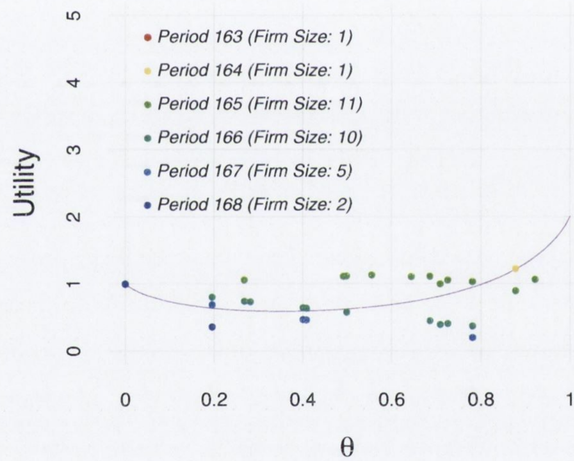


Figure 4.6: Utility of members of the medium case study firm at the end of the indicated time periods, plotted against their θ.

4.3 θ, Effort, Utility and the Firm

The case studies demonstrated a typical life-cycle pattern for medium and large firms. In this section we explore some of the factors which might determine whether a firm will grow to a large size.

4.3.1 Founder's θ

Being founded by an agent with high θ is a helpful but not sufficient condition for a firm to grow to a large size. Figure 4.7 shows the relationship between the maximum size a firm reaches in its lifetime and the θ of the agent which founded the firm. Our large case study firm, shown in red, has both large size and very high founder's θ, but it is evident that many other firms with equally high or higher founder's θ stay as singletons or very small firms. The medium case study firm, visible in green, has a relatively high founder's θ but we see that other firms with this founder's θ grew to be much larger.

Figure 4.8 confirms the intuition that high-θ agents are more likely to be founders of firms. This histogram shows the number of firms created by agents of various θ, and the relationship is very obvious. Another perspective on this is shown in Figure 4.9, which shows that as θ increases, agents are likely to create a higher number of firms over the course of the simulation, but the variance is extremely high. Some high-θ agents are extremely prolific, others not at all. The histogram in Figure 4.10 shows the total number of firms founded per agent. The most prolific agent founded 728 firms, whilst 1181 agents didn't voluntarily create any firms.

The relationship between the age of a firm and its maximum size is shown in Figure 4.11. In our two case studies, the larger firm had the longer lifespan, and we see that there is indeed a positive relationship between lifespan and maximum size, at least initially. Firms which die at a very young age don't have time to reach a large size. However, we also observe in this graph that the very long-lived firms tend to be quite small. So some extreme firms live fast and die young, others have a long, quiet life of isolation. The majority of firms, however, have a short lifespan, as shown in

the histogram in Figure 4.12. The extremely long-lived "hermit" firms are all founded by very high-θ agents, as shown in Figure 4.13. This agrees with our intuition that a low-θ agent is unlikely to remain a singleton for long, being willing to join even a relatively unproductive firm, rather than having to exert effort to maintain income, whereas a high-θ agent would prefer to work in isolation than to join an unproductive firm.

If there is a guaranteed predictor of a newly-formed firm's fate, we have not uncovered it in this section. We have seen that factors such as the θ of the founder of a firm do have an impact on the expected size and life span of the firm, but it is not a simple, predictable relationship and certainly not a linear one. An agent, whose θ and whose neighbours haven't changed, can start several identical singleton firms which may have vastly different outcomes, a point which is clearly illustrated by Figure 4.14.

The difference in outcomes between two firms founded by the same agent can only arise from the different state of the rest of the system. Other competing opportunities are available. At one time, the agent's new firm may be the best thing going in the neighbourhood and it gains momentum quickly as other agents flock to join, at another time the founder may find a better opportunity and leave before the firm has a chance to grow.

4.3.2 Agent Income, Effort and Utility

When agents are activated, they have three options and the proportion of agents choosing each option is remarkably stable throughout the simulation. 47% will move to an existing firm (a friend's firm), whilst 41.5% will stay where they are, and the final 11.4% will start a new firm.

The relationship between agent utility and θ is complex and interesting, and understanding this relationship sheds light on the dynamics of the simulation. An agent with a very low θ, close to 0, will always have a utility of close to 1. An agent with higher θ may achieve a higher utility in some periods, but at the risk of very low utility in other periods. High θ agents are workaholics with a high risk, high reward lifestyle. In Figure 4.15, the blue dots represent the utility of a low θ agent with $\theta = 0.1$, whilst the black dots represent the utility of a high θ agent with $\theta = 0.8$. The low θ agent

has a utility level of approximately 1 which does not change throughout the simulation. In contrast the high θ agent has highly volatile utility.

Figure 4.16 shows the relationship between utility and θ for a single time period of the simulation. This plot is similar to Figure 4.3 and Figure 4.6, but it contains all agents in the simulation for a single time period rather than snapshots of agents in a particular firm over several time periods. The risk-reward relationship can be seen clearly here, along with some other interesting patterns. The points in green represent agents who are members of the largest firm in the simulation. The points in yellow are agents in singleton firms, and the violet line represents the highest possible utility for a singleton firm. All agents above this line are better off than they would be in a singleton firm, those below this line worse off. The green "line" made up of agents in the largest firm crosses the yellow line at around $\theta = 0.4$, so firm membership is beneficial for agents with $\theta < 0.4$, but the high θ agents are stuck in a sub-optimal firm. At some point in the past, the largest firm was a utility-maximising option for these high θ agents, or they would not have joined it. Another feature of this plot is the red curve marking minimal utility. This is actually an artefact of the numerical optimisation algorithm and it represents the lowest achievable effort level, effectively zero effort.

A few agents have high levels of utility, but the majority have a utility of less than 1, shown in Figure 4.17. When plotted in double logarithmic coordinates (see Figure 4.18), the distribution of employee utility takes a strongly kinked shape, perhaps a double-Pareto distribution (Mitzenmacher 2003). The plot of the distribution of employee effort, shown in Figure 4.19, is in linear coordinates and shows an apparent linear decline in effort until the level zero is reached, and we observe that nearly half of all agents are contributing negligible effort.

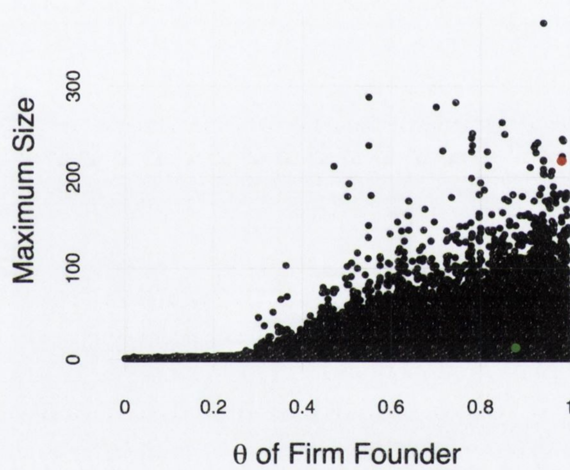


Figure 4.7: The maximum size which a firm will reach in its lifetime plotted against the θ of the founder of the firm. The red firm is that in the large firm case study, the green firm is that in the medium firm case study. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

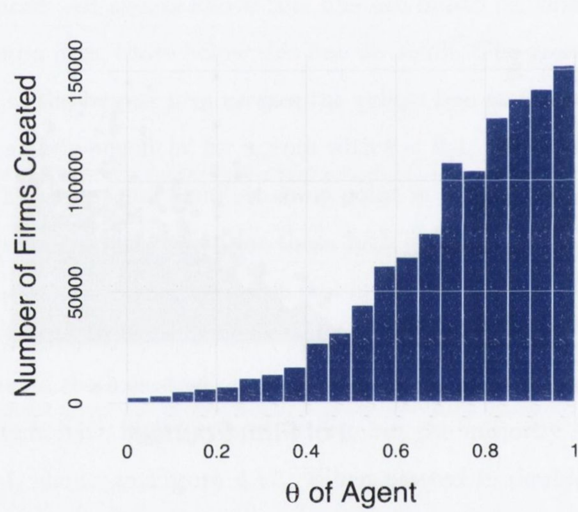


Figure 4.8: Histogram of the number of firms created per agent, categorised by θ. Includes all firms created after the first period of the simulation.

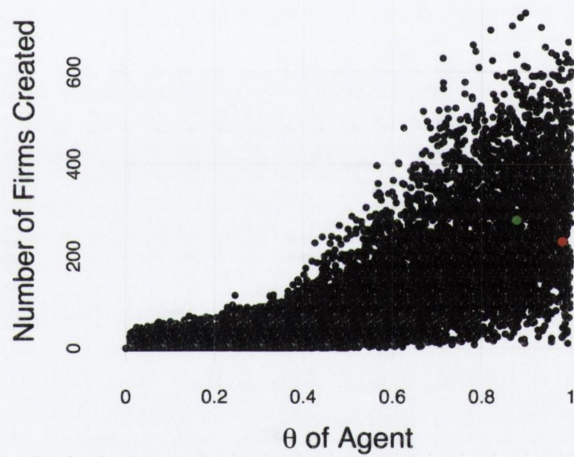


Figure 4.9: For each agent in the simulation, the number of firms created by that agent plotted against the agent's θ. The red agent is the founder of the large case study firm, the green agent is the founder of the medium case study firm. Includes all firms created after the first period of the simulation.

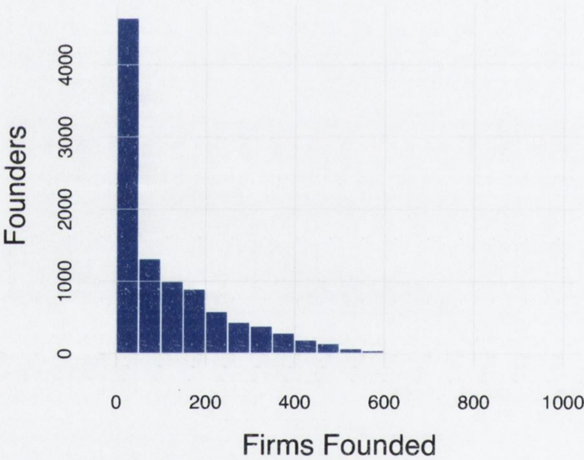


Figure 4.10: Histogram of the number of firms created per agent. Includes all firms created after the first period of the simulation.

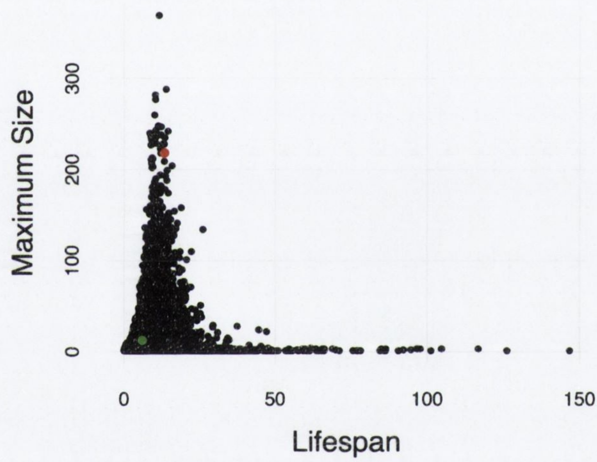


Figure 4.11: The largest size a firm achieves in its lifetime, plotted against the lifespan of the firm in periods. The red firm is that in the large firm case study, the green firm is that in the medium firm case study. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

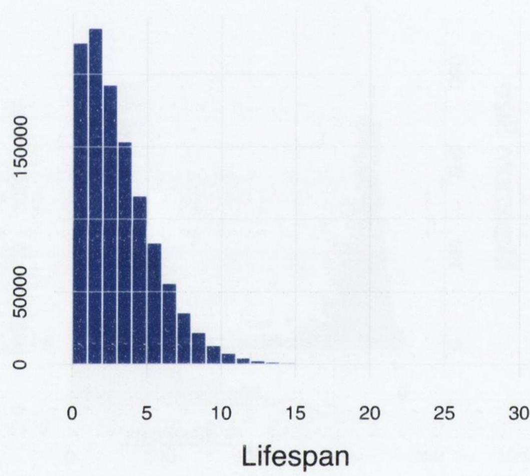


Figure 4.12: Histogram of lifespan of firms. Includes all firms which died after the first period and prior to the end of the last period of the simulation, except that 281 firms (0.02% of dataset) with lifespan longer than 30 periods are excluded.

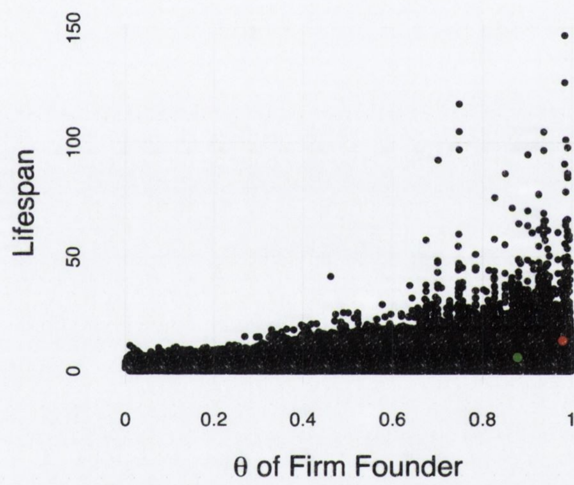


Figure 4.13: Lifespan of a firm plotted against the founder's θ. The red firm is that in the large firm case study, the green firm is that in the medium firm case study. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

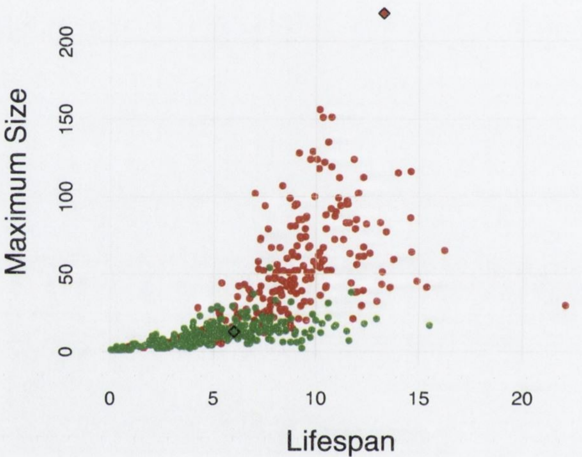


Figure 4.14: Lifespan and maximum size of firms started by a particular agent over the course of the simulation run. The red firms are those started by the founder of the large case study firm, the large case study firm itself is indicated by a black diamond. The green firms are those started by the founder of the medium case study firm, the medium case study firm itself is indicated by a black diamond. Includes all firms founded by these two agents which died after the first period and prior to the end of the last period of the simulation.

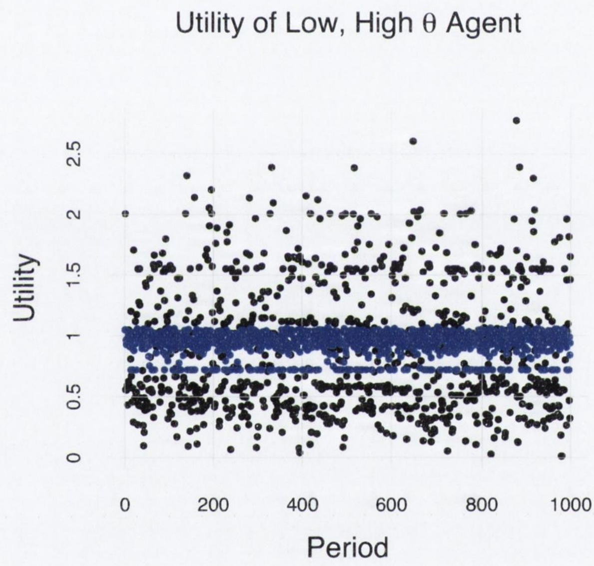


Figure 4.15: Time series of the utility of an agent with low θ (0.062) shown in blue and an agent with high θ (0.815) shown in black.

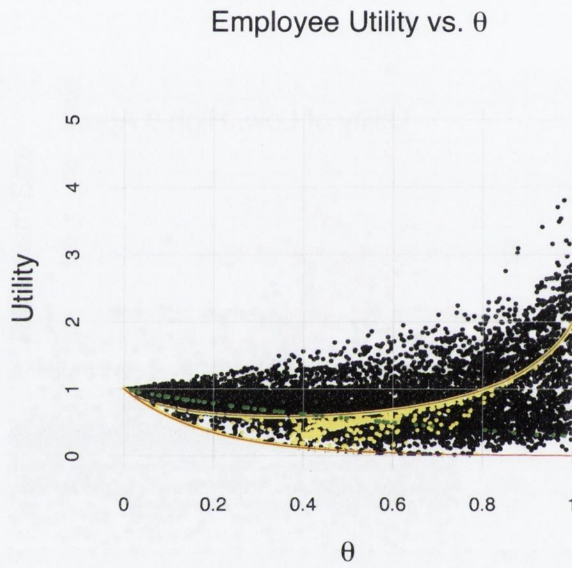


Figure 4.16: Utility of all agents in the simulation at the end of Period 1000, plotted against their θ. The green agents are members of the largest firm in the simulation at this time. The yellow agents are singleton agents. The purple curve represents optimum singleton utility. The red curve represents the utility of singleton agents with the minimum possible nonzero effort level of 0.00502.

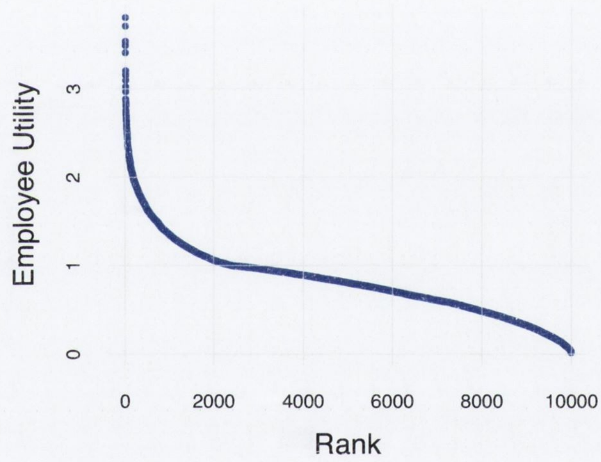


Figure 4.17: Distribution of employee utility for all agents in the simulation at the end of Period 1000.

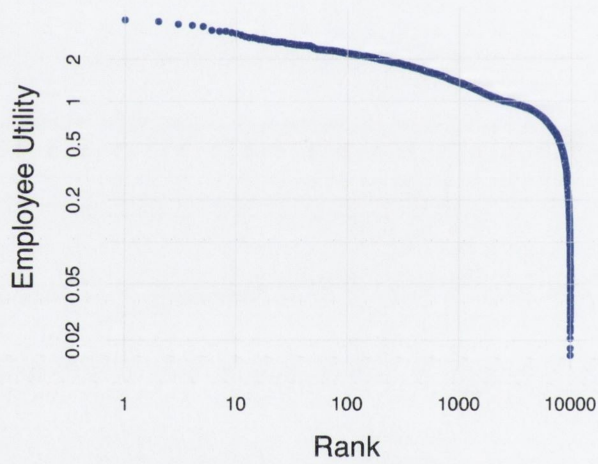


Figure 4.18: Distribution of employee utility for all agents in the simulation at the end of Period 1000, in double logarithmic coordinates.

Figure 4.19: Distribution of employee effort for all agents in the simulation at the end of Period 1000.

4.4 Firm Size Distribution

For all the chaos and unpredictability at the micro level, the aggregate statistics of the simulation show a remarkable stability. After an initial adjustment period, the average firm size settles down to a constant value of approximately 2.7 agents. With $N = 10,000$, this results in an average number of $3,650.3$ firms. Whilst the total number of firms remains more or less the same throughout the simulation, the firms themselves are constantly changing. An average of $1,142.6$ firms (31.3% of firms) die in every time period (see Figure 4.20). The birth rate is equal to the death rate and so the total number of firms is stable (see Figure 4.21).

The firm size distribution is presented as a bar chart in Figure 4.22. We see straight away that there is a highly skew distribution, with a large number of singleton firms present in the firm population, $1,867$ or 50.3% of firms.

Figure 4.23 shows the firm size distribution plotted with double logarithmic axes. The relationship here is not strictly linear, but it is a robust skew distribution which is at least power-law like. The data points have been overlaid with a fitted discrete power law. Whether or not these data do in fact have an underlying power law distribution, the power law exponent is useful as a summary statistic. As a reminder, the discrete form of the probability density function for the power law distribution is:

$$p(x) = \frac{x^{-(k+1)}}{\sum_{j=1}^{\infty} j^{-(k+1)}}. \tag{4.7}$$

Using maximum likelihood estimation, we obtain a fitted value for k of 0.951. The distribution of firm sizes remains skew with roughly the same shape throughout the simulation, despite the constant turnover of firms.

The productive output of a firm is determined by aggregating the contributions of employees' effort and applying the output function which has positive returns to scale, given by Equation 4.2. An individual employee's effort can range between 0 and 1, thus we have a theoretical maximum economic output represented by the situation of all employees exerting an effort of 1 in a single firm. This would result in $100,010,000$ units of production.

While some agents exert close to their maximum possible effort, others do almost no work and the average effort is around 0.3. The actual total output in the entire economy averages around 8,306.329 units, or 0.00831% of the potential output. The average output per firm remains fairly stable throughout the simulation at a level of 2.3 units of production. The minimum output is close to zero, whilst the maximum output is highly volatile, varying between 50.6 and 526.2.

4.5 Discussion

Returning to the discussion in Section 2.3 of the importance of birth/death and proportional growth to a skew distribution, it is easy to identify the elements of birth and death in our simulation. Firms are born when an agent decides that their utility would be highest in a new singleton firm, and they die when their last agent leaves to seek higher utility elsewhere. Proportional growth, too, seems likely, at least to an extent. What is particularly interesting about this model is that the counteracting forces of birth (increasing the number of small firms) and proportional growth (increasing the size of large firms) are both achieved. The ingredients within the model that lead to this are the economies of scale (output is of the order of effort *squared*), and the potential for free riding. Economies of scale promote the growth of larger firms, but are counteracted by free riding.

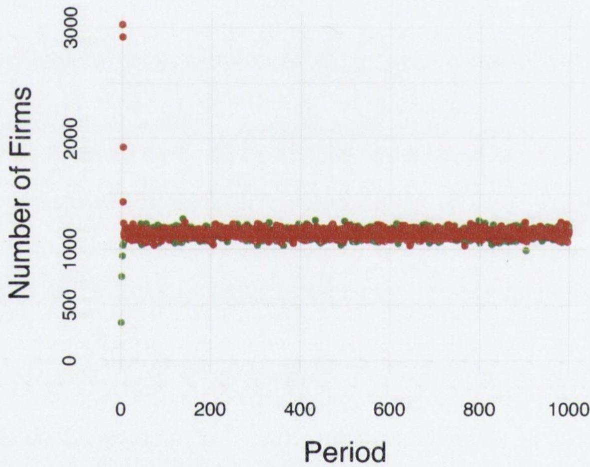


Figure 4.20: Time series of the number of firms which are born (green) and die (red) in each period.

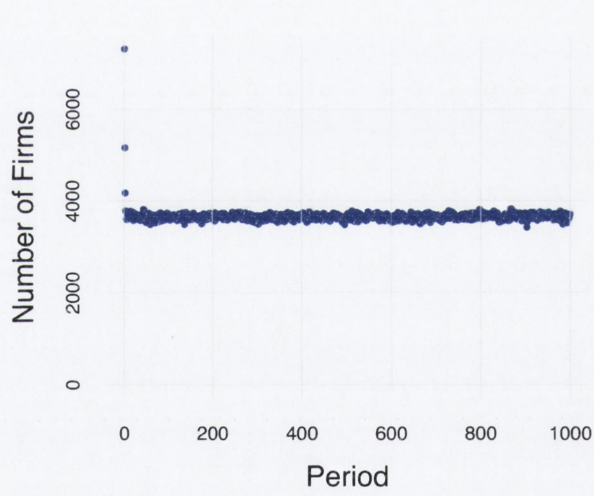


Figure 4.21: Time series of the total number of firms in each period.

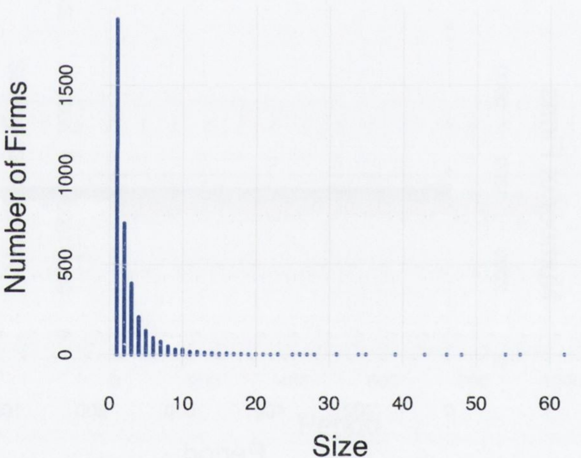


Figure 4.22: Firm size distribution.

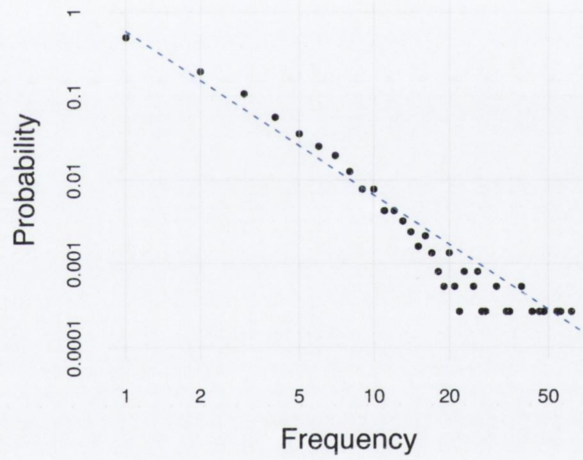


Figure 4.23: Firm size distribution plotted with double logarithmic axes and showing fitted discrete power law with $k = 0.951$.

Chapter 5

Extensions of a Variable Effort Model

Just as extreme and unusual medical conditions can help to illuminate the functioning of a healthy human body, extending the basic variable effort model with the intention of "breaking" the skew firm size distribution may help us to understand it better. Some of the modifications in this section will have only a minimal effect on the firm size distribution, others will be more noticeable.

5.1 Sequential Activation

In our original simulation runs, agents were activated randomly. A *period* of simulation time is defined to be every N agent activations where N is the total population of agents. If the agents are activated in sequence, then each agent will be activated exactly once per period. If the agents are activated randomly, then an agent may be activated once, more than once, or not at all in a given period. We saw in the case studies presented in Sections 4.2.1 and 4.2.2 that some agents remained in suboptimal firms for several time periods because they had not been activated and hence were unable to leave their firm. In this section, we examine results from simulations where agents are activated sequentially.

The plot in Figure 5.1 confirms the intuition that, with sequential activation, agents would not remain in a sub-optimal firm for more than 1 period. This firm's life cycle is delightfully blunt. The singleton firm in Period 167 is, of course, on the purple optimum singleton utility curve. At the end of Period 168, there are 9 agents enjoying extremely high utility, well above the purple curve. Things are still going well in Period 169 and the firm has grown dramatically to 163 agents. During the course of Period 170, however, things decline rapidly, and by the end of the period all agents having $\theta > 0.2$ are in a suboptimal situation. It is no surprise that by the end of Period 171 the firm is defunct.

Firms are larger on average but shorter-lived in a simulation with sequential activation. The mean firm lifespan is 2.2 and the median is 1.9, compared with the standard case's mean firm lifespan of 3.2 and median of 2.6. The mean firm size (calculated over all the firms in the simulation, rather than a cross-sectional snapshot) is 5.8 in the case of sequential activation with a median firm size of 3.0. For non-sequential standard activation the mean is 4.2 and the median 2.0.

The firm size distributions for both the standard and sequential models are shown in Figure 5.2, there are slightly fewer firms of size 1 and 2 in the sequential case, but more firms of almost every larger size, with a larger maximum firm size. As shown in Figure 5.3, however, large firms have a very short lifespan with sequential activation. This same plot also indicates that other firms, very small firms, have extremely long lives and can last for several hundred periods. We have already seen why we can expect such a short lifespan for most firms: sequential activation allows agents to escape from suboptimal firms within 1 period, but why does sequential activation also allow some firms to survive for such a long life?

The histogram in Figure 5.4 has a very striking and telling feature, a spike indicating that some agents found as many as 999 firms, that's a new firm *in every period.* (Compare this with the standard case shown in Figure 4.10.) Why might this be? With sequential activation, it is possible for agents to spend much or all of the simulation trapped in repeating loops. One such loop might be as follows: an agent in a singleton firm is joined by

a neighbour, making the firm a twosome. When the original agent is next activated, he decides to leave and create a new singleton firm, leaving the neighbour by himself. If no other agents become involved, and if no more tempting choices present themselves to either of these two, it is possible for this loop to repeat indefinitely since both agents will always be activated in the same sequence and, given the same situation, will always make the same decisions. This cyclicality, although not that particular scenario, can be seen in Figure 5.5, which shows the utility choices available to an agent. The pattern shows distinctly repetitive elements, if not a purely cyclical pattern. By contrast, a similar diagram for a typical agent under non-sequential activation shown in Figure 5.6 shows no regularity or predictability in the utility choices available. Every time an agent is activated, the simulation space of firms will be very different. In theory, it might be possible for a repeating loop to develop with non-sequential activation, but a scenario in which the order of activation of the two agents didn't matter is more difficult to construct. Even if one did appear by chance it would be a rare exception and not a dominant feature of the simulation. In some small simulations with sequential activation, the entire simulation can become a repeating loop, see Figure 5.7 for an illustration. As an aside, this has implications for the study of business cycles using agent-based models, sequential activation of agents can introduce artificial cyclicality. Returning to the question of the long-lived very small firms, we can easily envisage a scenario now where, with two agents locked in the repeating loop described above, a third singleton agent nearby is left in peace for hundreds of periods.

If we think of all the possible configurations of the simulation as a state space, we can think of sequential activation as resulting in a smaller search of such space, restricted by cyclical behaviour which limits the number of states that can be explored by the system. Clearly, unless there is a compelling reason to choose sequential activation, it should not be a default choice as it introduces many arbitrary behaviours into a system. The impact of sequential activation on the firm size distribution can be quantified by considering the fitted power law slope parameter k. The fitted value for sequential activation is 0.857, compared with the standard model's 0.951.

This lower value indicates a flatter slope, as can be seen in Figure 5.2, and corresponds to a higher level of market concentration (more large firms, fewer small firms).

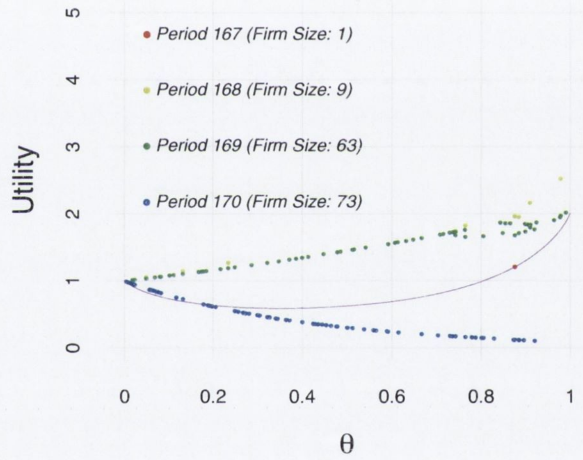


Figure 5.1: Utility of firm's employees, plotted against θ, at the end of various time periods.

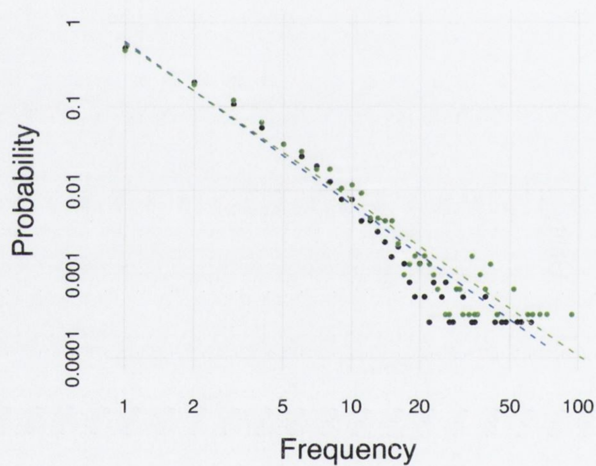


Figure 5.2: Comparison of firm size distribution for standard model (black) and with sequential activation (green) plotted with double logarithmic axes and showing fitted discrete power law with $k = 0.951$ (standard model) and $k = 0.857$ (sequential model).

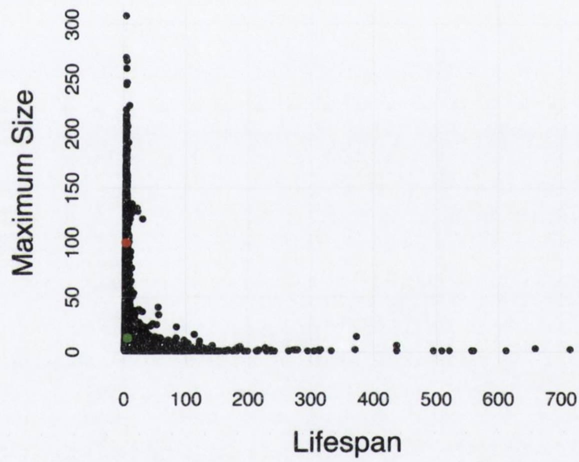


Figure 5.3: The largest size a firm achieves in its lifetime, plotted against the lifespan of the firm in periods. The red firm is that in the firm case study.

Figure 5.4: Histogram of the number of firms founded per agent during a simulation with sequential activation.

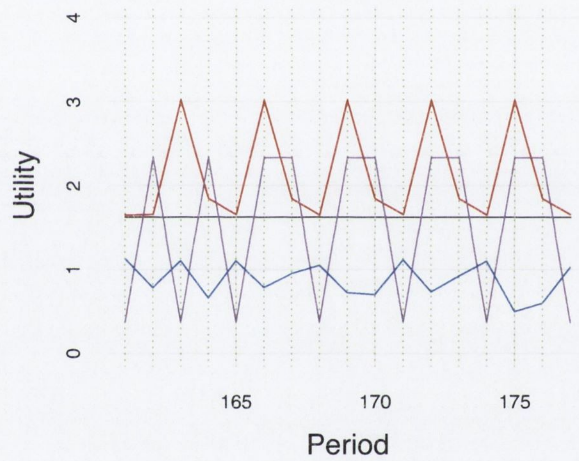


Figure 5.5: Utility calculated by an agent in a simulation with sequential activation for the agent's various options: staying in current firm (blue), creating a new firm (black), joining the first friend's firm (red) or joining the second friend's firm (purple). Vertical green lines indicate a change of firm.

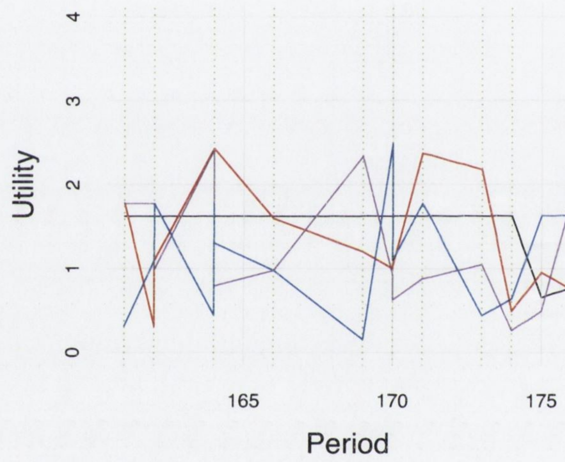


Figure 5.6: Utility calculated by an agent in a simulation with random activation for the agent's various options: staying in current firm (blue), creating a new firm (black), joining the first friend's firm (red) or joining the second friend's firm (purple). Vertical green lines indicate a change of firm.

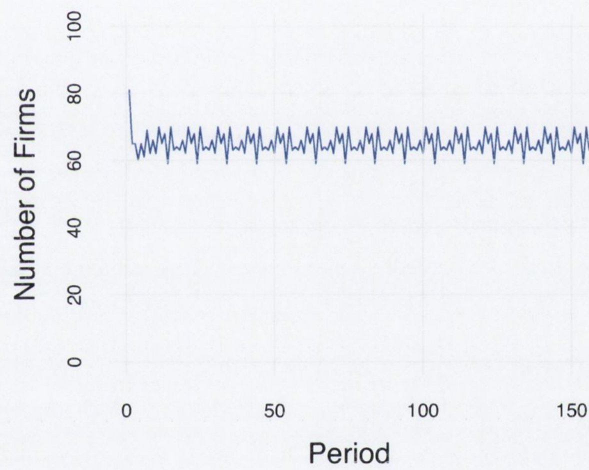


Figure 5.7: Time series of the number of firms present in a simulation with $N = 200$ agents, showing a loop which repeats every 10 periods.

5.2 Myopia

Agents have a limited number of firms to choose from. In the standard model, each agent is assigned $n = 2$ friends and those friends' firms are the only ones an agent is able to "see" and potentially move to. Limiting agents to local information is referred to as *myopia*. In this section we examine the impact of assigning more friends to an agent, reducing the degree of myopia.

Figure 5.8 shows the firm size distribution for the standard $n = 2$ case and also for $n = 3$, $n = 5$, $n = 10$, $n = 20$ and $n = 50$, along with the fitted power law line in each case (except $n = 50$). Increasing the number of friends has a definite effect on the firm size distribution. The more friends each agent has, the larger firms grow, and the fewer small firms are seen. Only 50 agents are in singleton firms when $n = 50$, compared with $1,867$ agents in the standard case of $n = 2$.

The source of this difference can be seen in Figure 5.9, which illustrates how the choices agents make change as the number of friends increase. When $n = 2$, 47.0% of agents choose to join a friend's firm, by $n = 50$, this value has increased to 92.5%. With so many more firms to choose from, it stands to reason that one of these firms is likely to offer a higher level of utility than either staying in one's current firm or creating a new firm. Reducing the agents' myopia in this way has virtually cut off the birth component in our model, it seldom makes sense to create a new firm when you have 50 friends, leaving the proportional growth component free to concentrate agents in larger firms.

In Figure 5.10, we see that firms are only founded by extremely high-θ agents, which is to be expected. Low-θ agents are very unlikely to find that creating a new singleton firm is their best option. In fact, no agents with θ less than 0.8 create firms in this simulation. More surprising is the extremely large size which is achieved by some firms, over 6,000 agents (60% of the population) in a few cases. Firms live longer, the mean firm lifespan with $n = 50$ is 4.0, compared with the standard mean firm lifespan of 3.2. Looking at Figure 5.12, this appears to be a simple consequence of the larger firm size, it takes longer for larger firms to complete their life cycle.

With $n = 50$, we observe an important tradeoff in this model. Agents

have higher utility and exert lower effort due to the greater efficiency of large firms, as shown in Figure 5.13. Agents spend more time in large firms since they are able to observe and take part in good opportunities. Large firms last longer, since more agents join them. The price to be paid for this is that we have a higher market concentration, there are virtually no singleton firms left. In some (albeit exceptional) periods, a majority of agents are in a single firm. Of course, it is not really possible to make value judgements about the ideal firm size in this model without a frame of reference. And, in this model even the very large firms are not able to exert any monopoly influence as they begin to decline in due course. What we can say is that myopia has a very strong role in determining the slope of the firm size distribution in a model such as this. The fact that agents can only see or join a small number of firms is responsible for the high rate of new firm creation and hence the high proportion of singleton firms for the low n simulation runs.

5.3 Discussion

In this chapter we reviewed the impact of sequential iteration and myopia on the simulation behaviour in general and the firm size distribution in particular. The relationship between myopia and the power law slope parameter k was quite striking, illustrating the application of that parameter as a market concentration indicator. With $n = 2$ friends, the slope was approximately 1, with $n = 20$, the slope was 0.5. If this parameter belatedly comes to be employed regularly as a market concentration indicator, it will be interesting to observe the development of intuition as to what characterises, say, a $k = 0.9$ economy. In the next two chapters we discuss additional models, and we return to a discussion and analysis of all the models in Chapter 8.

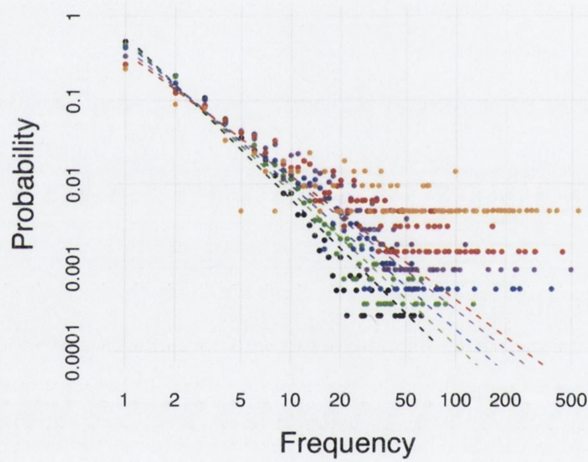


Figure 5.8: The firm size distribution for 2 (black), 3 (green), 5 (blue), 10 (purple), 20 (red) and 50 (orange) friends per agent, plotted in double logarithmic coordinates with fitted discrete power law having k parameter of 0.95 ($n = 2$), 0.83 ($n = 3$), 0.71 ($n = 5$), 0.56 ($n = 10$) and 0.47 ($n = 20$).

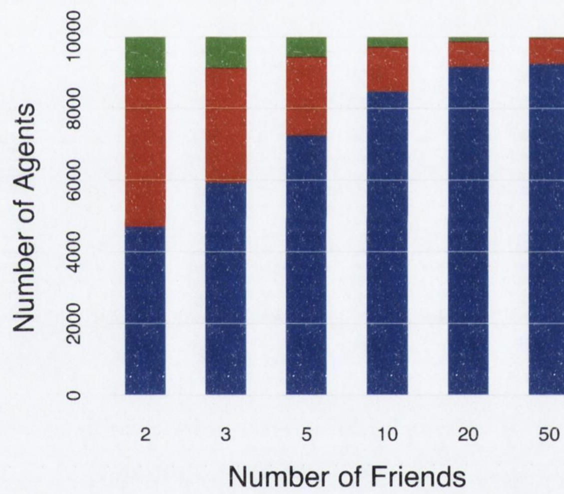


Figure 5.9: Number of agents in each period who join a friend's firm (blue), stay in their current firm (red) and create a new firm (green) for various values of n.

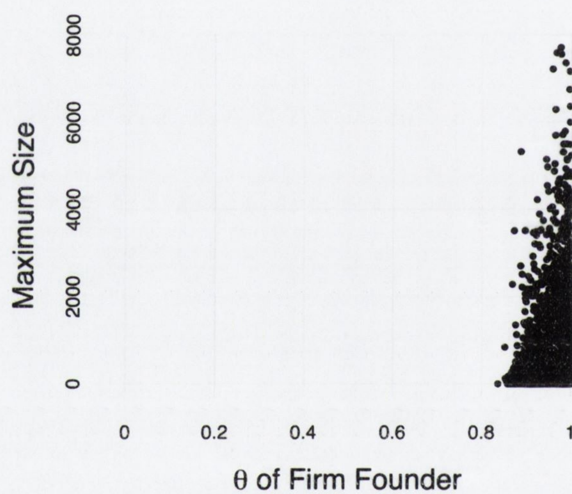


Figure 5.10: Maximum size of a firm plotted against the founder's θ. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

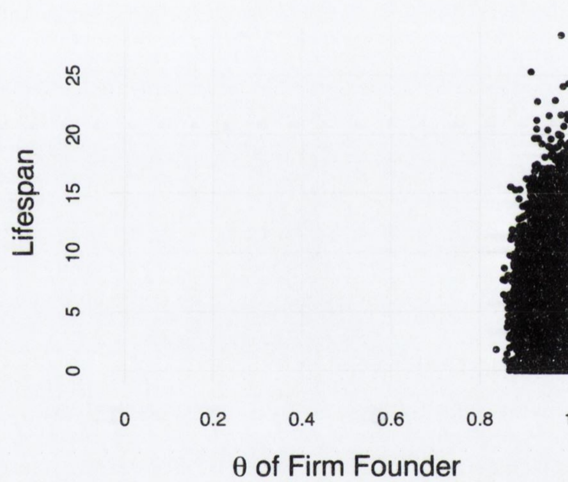


Figure 5.11: Lifespan of a firm plotted against the founder's θ. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

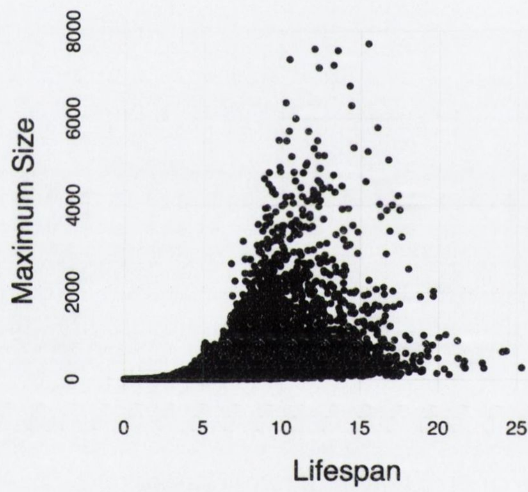


Figure 5.12: The largest size a firm achieves in its lifetime, plotted against the lifespan of the firm in periods. Includes all firms which died after the first period and prior to the end of the last period of the simulation.

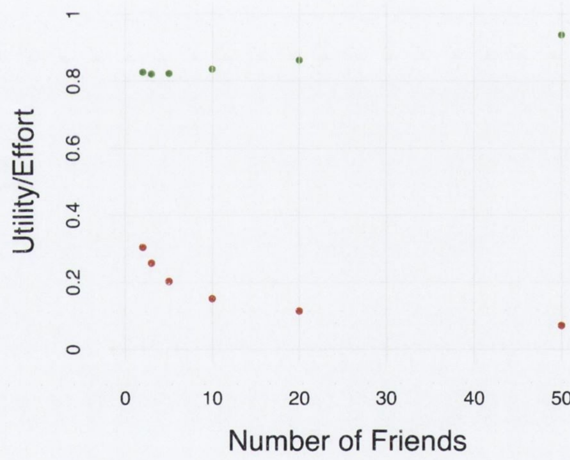


Figure 5.13: Mean agent utility (green) and effort(red) for various n.

Chapter 6

An Exogenous Birth Model

In our discussion of the history of Gibrat's Law in Section 2.3, we saw how Champernowne, Simon and others proposed a variety of mechanisms to prevent the undesirable increase in firm size variance over time which was the result of Gibrat's basic model.[1] The Simon and Ijiri firms model specifies an exogenous increase in the size of the overall economy in each time period, with a proportion α of new *opportunities* leading to the creation of new singleton firms (in our terminology) and the remaining proportion $1-\alpha$ being allocated to existing firms with probability proportional to their size. Mathematically, this leads to a Yule distribution, close to a power law distribution(Ijiri and Simon 1977).

We ask two questions in this chapter. Firstly, does an exogenous birth model such as the Simon-Ijiri model lead to a skew firm size distribution when implemented as an agent-based model? Secondly, what are the differences, if any, between an exogenous birth process and the endogenous birth and growth observed in the variable effort model presented in Chapter 4 and discussed further in Chapter 5. In the variable effort model the birth rate of new firms and the growth patterns of existing firms are not programmed

[1] Simon's original work was not, as we saw, specifically intended by him to address this problem in Gibrat's model but it did so nonetheless.

into the simulation, instead they are emergent properties of the interaction of agents.

6.1 Implementing Exogenous Birth with Agents

We will, of course, implement this model as an Agent-Based Model using the framework developed in Chapter 3. Thus the concepts of friendship, firm membership and agent activation are such as we have already discussed. In this instance, due to the nature of this model, there is no real meaning for firm output or agent utility.

Agents will obey two simple behavioural rules each time they are activated. Firstly, with probability α an agent will create and join a new singleton firm (unless they are already in a singleton firm in which case they take no action and remain there). With probability $1 - \alpha$ the agent joins one of firms visible to it (via its friends) with probability proportional to the size of each firm. Hence, for example, if the agent belongs to a firm of size s_0 and has two friends belonging to firms of size s_1 and s_2 respectively then the probability of remaining in the same firm after activation is:

$$(1 - \alpha)\frac{s_0}{s_0 + s_1 + s_2}. \tag{6.1}$$

Figure 6.1 illustrates all the options available to the agent in this scenario and their corresponding probabilities.

This is not entirely identical to the Simon-Ijiri model, in particular we are not actually increasing the size of our simulation in each time step. We are creating new firms at the rate α, but to create these firms we are removing agents from existing firms.

This model has two system-wide parameters, α and n, the number of friends per agent. Agents are homogeneous in this simulation, there are no agent-specific parameters. The parameter α, the birth rate of new firms, can take values between 0 and 1, with both 0 and 1 permissible values corresponding to degenerate cases. When $\alpha = 0$, agents will never create a new singleton firm, when $\alpha = 1$, agents will never join an existing firm. In Section 6.2, we explore the impact of different values for α and n on the

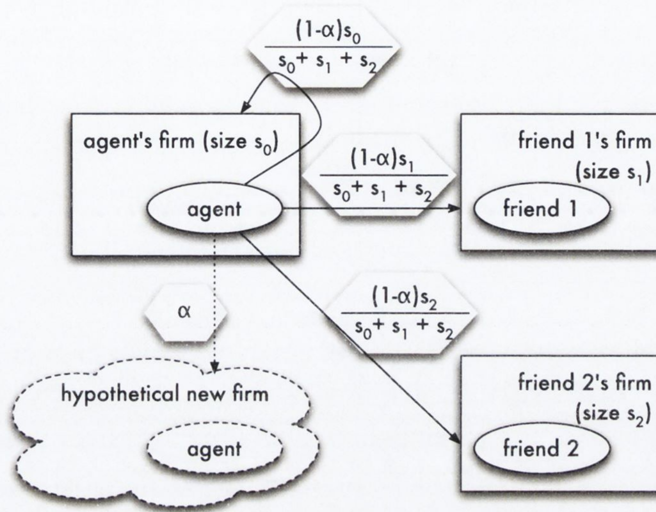


Figure 6.1: Example of the options and corresponding probabilities available to an activated agent in the Simon-Ijiri firms model.

simulation.

6.2 A Survey of the Parameter Space

The contour plot in Figure 6.2 shows the mean number of firms present in the simulation for various values of α and n. This plot incorporates the results of 99 individual simulation runs with a population of $N = 10,000$ agents in each case. α was assigned values between 0 and 1 inclusive with an increment of 0.1, and n took integer values from 1 to 9 inclusive. The number of firms was averaged over 700 periods of a 1,000 period simulation, the first 300 periods were excluded to allow the system to stabilise.

We can see that the number of firms in the simulation increases as the birth rate α of new firms increases, which agrees with our intuition. The light beige coloured regions show close to 10,000 individual agents, or almost every agent, in a singleton firm. By contrast, the purple and dark blue regions indicate a very small total number of firms. It is clear from this contour plot that the firm size distribution generated by this simulation will look very different depending upon the value of α chosen.

The number of friends also has an impact on the mean number of firms, but it appears that this is a secondary effect, much less strong than the impact of α. In general, having more friends results in a smaller number of firms, ceteris paribus, however the number of friends only has an impact for some values of α. The behaviour of the system with extreme values of α does not change due to the number of friends.

The maximum firm size, averaged over the last 700 periods, shown in Figure 6.3, decreases as the birth rate increases. For middling values of α, an increase in the number of friends also increases the maximum firm size. Maximum firm size is plotted again in Figure 6.4 for fixed $n = 2$ to show more clearly the relationship with the parameter α. Increasing the number of agents changes the scale but not the shape of the relationship between maximum firm size and α. We see from this plot that there seems to be three distinct regions of behaviour: one for $0 \leq \alpha < 0.2$, another when $0.2 \leq \alpha < 0.6$ and a third for $0.6 \leq \alpha \leq 1$, and we will discuss the firm size distribution in each of these three regions in the next section.

Considering again the total number of firms in the simulation, in Figure 6.5 we observe that the variance of the number of firms is very small near $\alpha = 0$ and $\alpha = 1$, increasing significantly between them. When $\alpha = 0$ or $\alpha = 1$, the number of firms will be close to 1 or 10,000 respectively. As α increases from 0 or decreases from 1, the number of firms can take many different values which will be determined by random interactions in the simulation and so the increased variance makes sense given what we have already learned about the system.

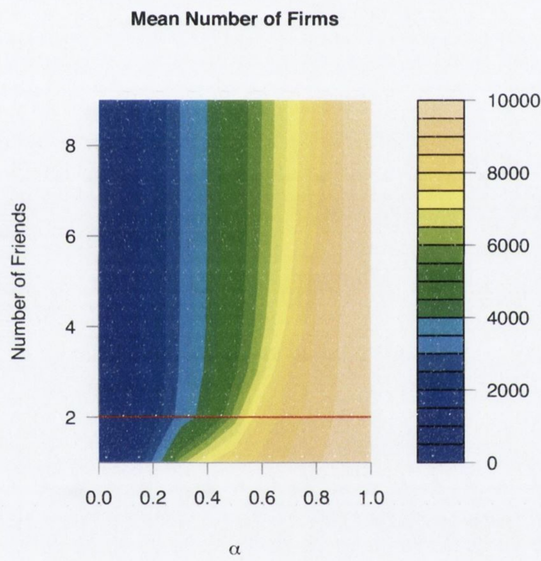


Figure 6.2: Mean number of firms averaged over Periods 300-1000 for various values of α and n.

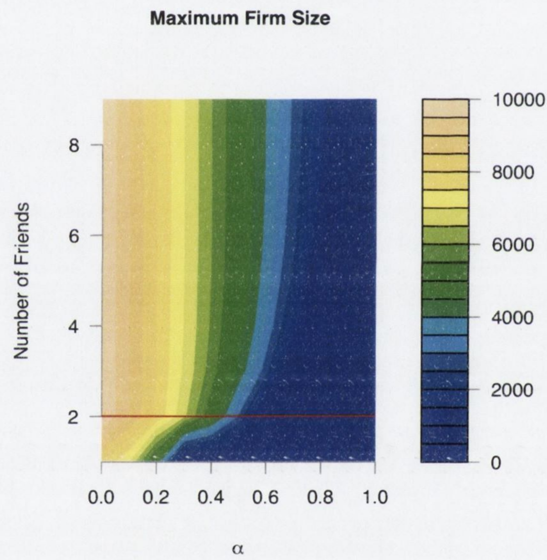


Figure 6.3: Mean maximum firm size averaged over Periods 300-1000 for various values of α and n.

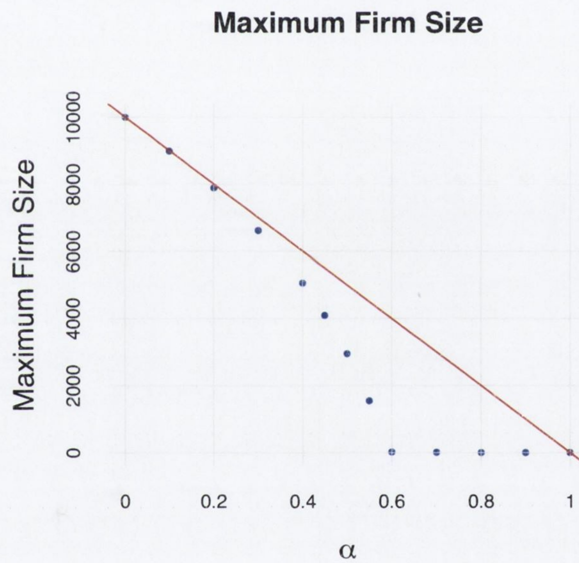


Figure 6.4: Mean maximum firm size averaged over Periods 300-1000 for various values of α with $n = 2$.

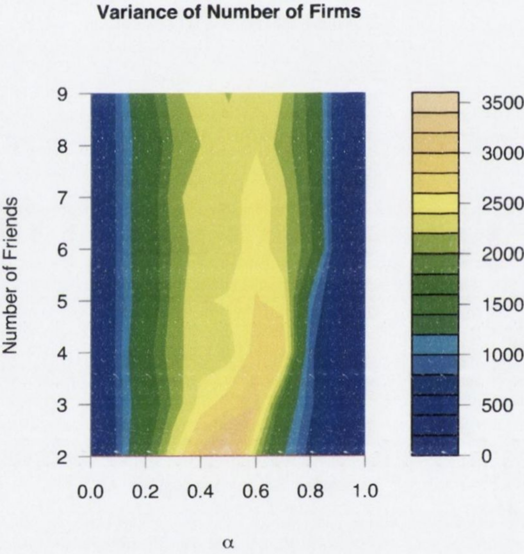


Figure 6.5: Variance of number of firms averaged over Periods 300-1000 for various values of α and n.

6.3 Firm Size Distribution

In this section we assume $n = 2$, unless stated otherwise.

6.3.1 A "Super" Firm

Firm Size	Number of Firms
1	1012
2	6
8976	1

Table 6.1: Period 1000 of Simon Model with $n = 2$, $\alpha = 0.1$ and 10000 agents.

In the region $0 \leq \alpha < 0.2$, the maximum firm size is close to the total number of firms in the simulation. This implies that most of the agents in the simulation are members of a single "super" firm. For example, Table 6.1 shows the firm size distribution for Period 1000 of a simulation run with $\alpha = 0.1$. In this example, most of the agents in the simulation are members of a single firm of size $8,976$.

The red line in Figure 6.4 shows what the maximum firm size would be if all agents, except the proportion α just forced into creating new singleton firms, were grouped into a single "super" firm. For $\alpha < 0.2$, this in fact appears to be the situation, the observed maximum firm sizes are almost on the red line. The singleton firms, and the handful of 2-agent firms, shown in Table 6.1, are present because 10% of agents are assigned to new singleton firms in each time period. The 90% (or $1-\alpha$) of activated agents who select a new firm based on proportional growth will, with high probability, choose to join (or remain in) the super firm. Not only is a super firm attractive in itself, since the agents are programmed to prefer large firms, but a super firm is likely to be the only option available to an agent as most if not all of the agent's friends will already be members of the super firm. For $\alpha = 0.1$ and $n = 2$, the average number of distinct firms available to an agent is 1.3, much less than the theoretical maximum of 3. If we increase the number

of friends in the simulation to 5 or 10, then the reduction in the number of available firms from the theoretical maximum is more marked. With 5 friends, the mean number of firms available is 1.6, with 10 friends it is 2.1. Figure 6.6 illustrates for various values of α and n, the steeper slope with which the purple ($\alpha = 0.8$) points rise indicating the more normal increase in the number of available firms as the number of friends increase.

6.3.2 More Diversity

Firm Size	Number of Firms
1	5490
2	481
3	106
4	33
5	8
6	2
7	3
10	1
3015	1

Table 6.2: Period 1000 of Simon Model with $n = 2$, $\alpha = 0.5$ and 10000 agents.

For $0.2 \leq \alpha < 0.6$, there is still a very large firm dominating the simulation, but the deviation away from the red line in Figure 6.4 indicates that some agents are choosing to join or remain in smaller firms. As shown in Table 6.2, the distribution of firm sizes now includes many more sizes of small firm. The growing diversity of firm sizes reflects the fact that, with so many singletons, many agents will be singletons and also have both friends as singletons, so will not have the option of joining the "super" firm. These agents will cluster into larger firms which may, eventually, lead them to be able to join the "super" firm, but in the mean time these agents support a greater number of firm sizes. Figure 6.7 shows the number of distinct firm sizes present for various parameter values.

6.3.3 Small Firms Only

Firm Size	Number of Firms
1	8544
2	552
3	81
4	20
5	3
6	1
8	1

Table 6.3: Period 1000 of Simon Model with $n = 2$, $\alpha = 0.8$ and 10000 agents.

Finally, in the region $0.6 \leq \alpha \leq 1$ the very large firm disappears entirely. The firm size distribution as shown in Table 6.3 consists entirely of small to medium-sized firms. The firm size distributions for all three sections are shown plotted together in Figure 6.3.3.

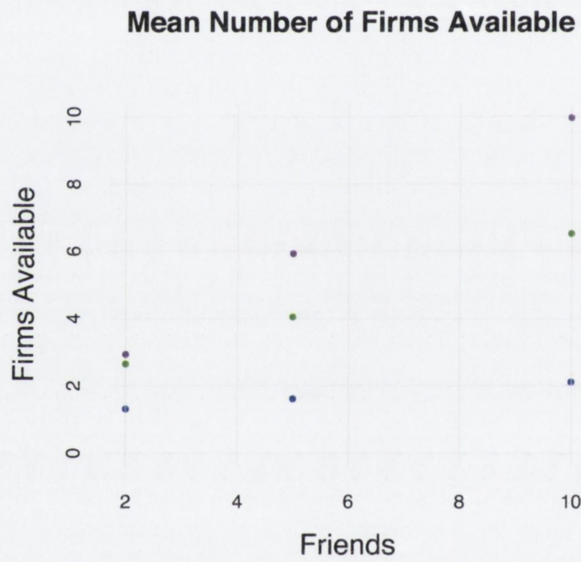


Figure 6.6: Mean number of firms available for $\alpha = 0.1$ (blue), $\alpha = 0.5$ (green) and $\alpha = 0.8$ (purple) and various values of n.

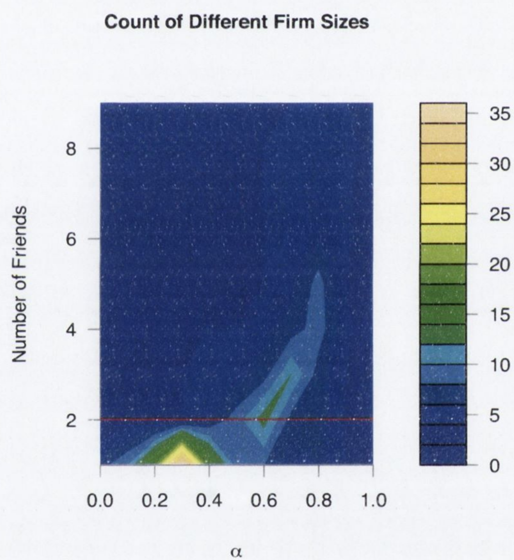


Figure 6.7: Count of number of distinct firm sizes in Period 1000 of a run of Simon's model over various values of α and n with 1,000 agents.

6.4 Discussion

The exogenous birth model presented here does result in a skew firm size distribution, however the presence of a "super" firm for many parameter values distorts this distribution away from the power law like distributions we observed with the variable-effort model. In order to prevent the appearance of a "super" firm we were obliged to set the α parameter to an extremely high value, it is not likely that Simon intended for the α parameter to be anything like as large as we have experimented with in this model, given its interpretation as a growth rate for the economy. We do see hints of a skew distribution in places, and it is easy to imagine that if the "super" firm were not present at lower values of α then the system would achieve a skew distribution closer to the lognormal or power law.

In answer to our questions posed at the beginning of the chapter, we do not obtain a skew firm size distribution in the way we expected, and so there does seem to be a difference between an exogenous birth process and an endogenous one. Although occasionally very large firms were seen in the variable effort model, they did not persist and did not distort the firm size distribution in the way that the "super" firm in the exogenous birth model did.

The endogenous instability of large firms in the variable effort model is the key to understanding the difference between these two models. The birth of new firms in the variable effort model came at the expense of older large firms, due to the older firms' inherent unattractiveness. By contrast, the exogenous birth of new firms does not mean that large firms are any less attractive, and the presence of the "super" firm confirms this. This model could, perhaps, be modified to prevent the appearance and persistence of a "super" firm. Large firms could be culled or divided into smaller firms when they reach a certain size. Such an approach, however, would be ultimately less satisfying than a model in which the large firms regulated themselves.

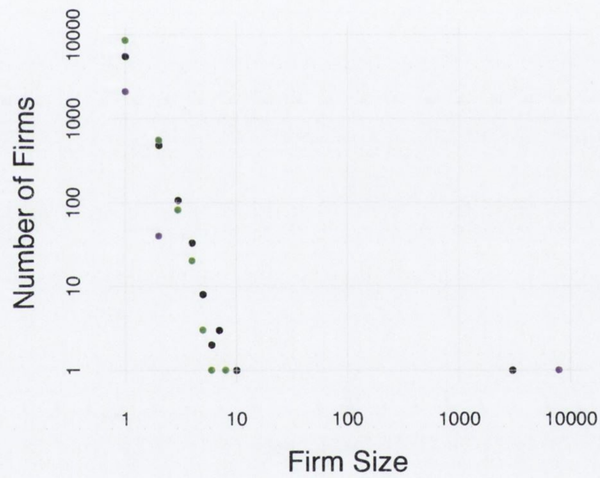


Figure 6.8: Firm Size Distribution for $\alpha = 0.2$ (purple), $\alpha = 0.5$ (black) and $\alpha = 0.8$ (green).

Chapter 7

A Cost Curve Model

In this chapter we attempt to forge stronger links between our agent-based analysis of firm size distributions and mainstream industrial organisation. This quote from Sutton nicely summarises the contemporary dilemma concerning firm sizes:

> The two aspects of structure — the cross-industry differences in concentration, and the within-industry skewness — sat uneasily side by side. The students made no attempt to relate the two stories. Neither did their textbooks. (Sutton 1998)[p. xiii]

The skew distribution has not been a central theme in industrial organisation, either as a source of inspiration or as a stylised fact which theories of the firm should endeavour to address, as Simon and Ijiri lament: "The classical theory would admit a normal distribution, a rectangular one, or a single size for all the firms in an industry as readily as it would admit the skew distributions (whether they be Pareto, Yule or log normal) that are actually observed." (Ijiri and Simon 1977)[p. 8-9]. Industrial organisation theory, when it addresses firm size at all, does so in terms of the cost curve facing a firm and hence tells the story that "the size and the number of the firms in an industry are related to the degree of returns of scale." (Tirole 1997)[p. 18]. This can be interpreted to mean that all firms within an industry will have the same size, if they share a common cost curve, or that the firm size distribution will depend upon the distribution of cost curves where

they are heterogeneous. We will explore an agent-based implementation of a cost-curve firm size model in this chapter.

7.1 Implementing Cost Curves with Agents

Within our standard friendship network framework, each agent is initialised into a singleton firm which is allocated a random unit cost. Agents will have a utility function which tells them to seek the firm with the lowest unit cost. The intuition behind seeking a firm with low unit cost is that the profits of such a firm will be higher, corresponding to higher agent income. Since we ignore the demand side in this model in the interest of simplicity, the relationship between low unit cost and high agent income/utility is very simple, agent utility is equal to 1 minus the effective unit cost of the firm they are a member of.

With this as our starting point, we then consider a variety of scaling regimes, where the unit cost of a firm changes with the market share of the firm, where market share is defined to be the proportion of agents in the simulation who are members of that firm. The taxonomy of scaling regimes is based upon work by Mazzucato who used replicator dynamics rather than agent based modelling to consider the relationship between cost curves and market structure (Mazzucato 2000). We distinguish here between initial unit cost, which is the random value allocated to the firm as its starting point, and the effective unit cost which is the initial unit cost adjusted for the market share of the firm according to the relevant scaling regime.

7.1.1 Static Economies of Scale

Each firm, indexed by an integer j, has a unit cost function c_j which depends upon the market share s_j of the firm (measured by the number of employees divided by total employee population):

$$c_j(s_j) = \hat{c}_j\left(1 + \phi\left(2\nu - s_j\right)s_j\right) \quad (7.1)$$

$$= \hat{c}_j\left(1 + 2\phi\nu s_j - \phi s_j^2\right) \quad (7.2)$$

$$\frac{dc_j}{ds_j} = \hat{c}_j (2\phi\nu - 2\phi s_j) \tag{7.3}$$

$$= 2\hat{c}_j\phi(\nu - s_j) \tag{7.4}$$

$$\frac{d^2c_j}{ds_j^2} = -2\hat{c}_j\phi \tag{7.5}$$

Constant Returns To Scale

$$\phi = 0 \tag{7.6}$$

$$c_j = \hat{c}_j \tag{7.7}$$

$$\frac{dc_j}{ds_j} = 0 \tag{7.8}$$

$$\frac{d^2c_j}{ds_j^2} = 0 \tag{7.9}$$

In this scenario, the market share does not have an impact on fitness. Eventually, all employees will end up in that firm which happened to be given the lowest unit cost at the initialisation of the simulation (unless some employees are disconnected from that firm, although this is highly unlikely).

Decreasing Returns to Scale

$$\phi < 0 \tag{7.10}$$

$$\frac{dc_j}{ds_j} = 2\hat{c}_j\phi(\nu - s_j) \tag{7.11}$$

$$\frac{dc_j}{ds_j} > 0 \text{ if } s_j > \nu \tag{7.12}$$

$$\frac{dc_j}{ds_j} < 0 \text{ if } s_j < \nu \tag{7.13}$$

$$\tag{7.14}$$

When the parameter $\phi < 0$, we have static decreasing returns to scale. That is, the unit cost increases as the firm size grows above a market share of $s_j = \nu$. If $\gamma = 0$, that is all firms have same initial unit cost, we would expect employees to organise themselves into firms of size $\hat{S} = \nu N$ or as close as they can achieve.

Increasing Returns to Scale

$$\phi > 0 \quad (7.15)$$

$$\frac{dc_j}{ds_j} = 2\hat{c}_j\phi\left(\nu - s_j\right) \quad (7.16)$$

$$\frac{dc_j}{ds_j} < 0 \text{ if } s_j > \nu \quad (7.17)$$

$$\frac{dc_j}{ds_j} > 0 \text{ if } s_j < \nu \quad (7.18)$$

$$(7.19)$$

In static increasing returns to scale, unit costs decrease as the firm market share grows above the critical value of ν. Thus we should probably assume $\nu = 0$ to have sensible behaviour. Just as in the case of constant returns to scale, all employees will end up in one firm, and it will be a firm which had a relatively low initial unit cost, but not necessarily the lowest. A firm which grows quickly, thus reducing its unit cost, will be more competitive than a firm which perhaps started with a lower initial unit cost but added employees at a slower rate. There are many possible equilibrium outcomes in this scenario. The friendship network may come into play strongly here, the number of inward links into a particular firm will determine its growth rate in the first time period of the simulation and may be crucial in determining the final winner.

7.1.2 Dynamic Economies of Scale

We multiply the last term in Equation 7.1 by a factor of $(1 - \frac{1}{3}\beta s_j)$. We can set $\beta = 0$ and revert to Equation 7.1. Since dynamic returns are an extension of the Increasing Returns scenario, we assume $\nu = 0$, $\phi > 0$.

$$c_j(s_j) = \hat{c}_j\left(1 + \phi\left(2\nu - s_j\right)s_j\left(1 - \frac{1}{3}\beta s_j\right)\right) \quad (7.20)$$

$$= \hat{c}_j\left(1 - \phi s_j^2 + \frac{1}{3}\beta\phi s_j^3\right) \quad (7.21)$$

$$\frac{dc_j}{ds_j} = \hat{c}_j\left(-2\phi s_j + \beta\phi s_j^2\right) \quad (7.22)$$

$$= \hat{c}_j\phi s_j\left(\beta s_j - 2\right) \quad (7.23)$$

Since $\phi > 0$, $0 < \beta \leq 1$, $0 \leq s_j \leq 1$, we have $\frac{dc_j}{ds_j} < 0$. This corresponds to static increasing returns, i.e. the cost is decreasing as firm size increases.

$$\begin{aligned} \frac{d^2 c_j}{ds_j^2} &= \hat{c}_j \left(-2\phi + 2\beta\phi s_j\right) & (7.24) \\ &= -2\hat{c}_j \phi \left(1 - \beta s_j\right) & (7.25) \end{aligned}$$

If $\beta = 1$, then $\frac{d^2 c_j}{ds_j^2} \leq 0$ for all values of s_j. This corresponds to increasing returns to scale, as firm market share increases, unit cost decreases at an increasing rate. If $0 < \beta < 1$, then for $s_j < \beta$ we have increasing returns and for $s_j > \beta$ we have decreasing returns. This case corresponds to Mazzucato's definition of dynamic decreasing returns which are decreasing above an implicit cutoff point. Note that if $1 < \beta < 2$, we have dynamically increasing returns for all s_j with no critical point within the domain of the s_j. We would expect such a system to behave similarly to the case of simple increasing returns to scale, i.e. $\beta = 0$.

Table 7.1 summarises the coefficients and parameters of the system[1].

Symbol	Description
s_j	Market Share of Firm j
$\hat{c}_j$	Original Unit Cost of Firm j (Assigned Randomly At Initialisation of Simulation)
γ	Std Deviation of Probability Distribution used to generate $\hat{c}_j$
ϕ	Coefficient for Increasing/Decreasing Returns
ν	Critical Point for Increasing/Decreasing Returns
β	Critical Point for Dynamically Increasing/Dynamically Decreasing Returns

Table 7.1: Parameters for the cost curve model.

[1] Whilst these equations are similar in intent and were inspired by the equations given by Mazzucato, there are some differences relating to the differing methods of computational implementation (Mazzucato 2000). The parameters ν and β, explicit here, are implicit in the Mazzucato equations and similarly the Mazzucato parameters λ and α, which determine the speed of adjustment, are implicit here. The parameter ϕ, which determines the nature of the scaling regime, corresponds to Mazzucato's f and g functions.

7.2 Results

7.2.1 Static Economies of Scale

Constant Returns to Scale

Recall that we expect an equilibrium outcome with all agents in the firm which had lowest unit cost at initialisation, and this is indeed the outcome we observe. A time series of the number of firms in the simulation, shown in Figure 7.1, shows that the firm population drops from its initial value of 1,000 to its final value of 1 within 10 time periods. The simulation has reached a steady state at this point and does not change after this. The minimum, maximum and mean unit cost is plotted in Figure 7.1, these lines converge as all the agents join the firm with lowest unit cost. By contrast, if we assume a homogeneous distribution of initial unit costs, then agents will remain in singleton firms (or whatever size firm they are initialised in), there being no benefit to joining a different firm.

Decreasing Returns to Scale

If we initialise all firms with homogeneous unit cost of 0.5 (i.e. $\gamma = 0$), then with $N = 1,000$ agents and $\nu = 0.1$ we would expect to see an equilibrium outcome of 10 firms of approximately 100 agents each. In fact, Table 7.2 shows that instead agents group into 6 equally sized firms. This is not the only possible outcome, but it is the most common. Occasionally we observe a grouping into 7 firms.

Firm Size	Number of Firms
166	2
167	4

Table 7.2: Firm size distribution for $\phi = -1.00$, $\nu = 0.10$, $\beta = 0.00$ and $\gamma = 0.00$.

The reason for this suboptimal outcome can be determined by considering the unit cost function, which for these parameter values is $c_j(s_j) =$

$\hat{c}_j\left(1-0.2s_j+s_j^2\right)$. This equation is plotted in Figure 7.3 (the purple curve) and it shows that while the optimum market share is indeed 0.1 (indicated by the red vertical mark), the equilibrium outcome of 6 firms (indicated by the orange dot) is superior to the market share of singleton firms (indicated by the green horizontal line and the green dot), and this fact is sufficient for the 6 firm state to be a stable outcome.

All agents begin in singleton firms, at the green dot. After 1 period, the blue dots indicate that the largest firms in the simulation are approaching the optimal market share. However, once they reach that market share agents will continue to join them and they will continue to grow, despite the fact that the unit cost is now increasing. These large firms are still preferable to the agents' existing firms. When they reach a size of approximately $\frac{1}{6}$ market share, agents finally stop joining them and join smaller firms which are now preferable, until eventually the simulation reaches the distribution in Table 7.2.

The agents do achieve a balanced equilibrium, they divide themselves evenly into firms, however they are not able to divide themselves into optimal firms. This is the result of the agents' simplicity, they are not able to strategize or to understand that 0.1 is an optimal value. It is also the result of allowing free entry into firms. If the agents within a firm were able to restrict new members when it was not in their interest at accept them, then the large firms would "freeze" when they reached 100 agents. However this might not result in a superior overall outcome, some agents might become isolated in small, sub-optimal firms if all of their friends were in 100-agent frozen firms. The process as it stands is rapid, equitable and robust, if not optimal. With heterogeneous initial unit costs, agents will again organise themselves into firms with equal unit costs adjusted for market share, but these will not necessarily be equal sized firms. With homogeneous initial unit cost, the identity of the six surviving firms is random, with heterogeneous initial unit costs the final six firms will be ones which happened to have low initial unit cost.

Increasing Returns to Scale

With increasing returns to scale, the equilibrium outcome is for all agents to be in a single firm, the outcome is effectively indistinguishable from the case of heterogeneous constant returns to scale. The convergence of the maximum, minimum and mean unit costs for this case is shown in Figure 7.4.

7.2.2 Dynamic Economies of Scale

Dynamic economies of scale, as an extension of increasing returns to scale, has a similar outcome, in that all agents end up in a single firm. The convergence of the maximum, minimum and mean unit costs for this case is shown in Figure 7.5.

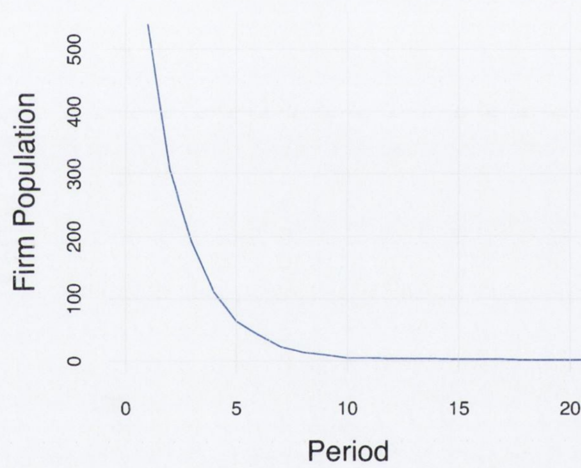


Figure 7.1: Time series of the number of firms in the simulation for $\phi = 0$. Data is collected after the end of each time period and so the initial firm population of 1,000 which corresponds to time 0 is not visible here.

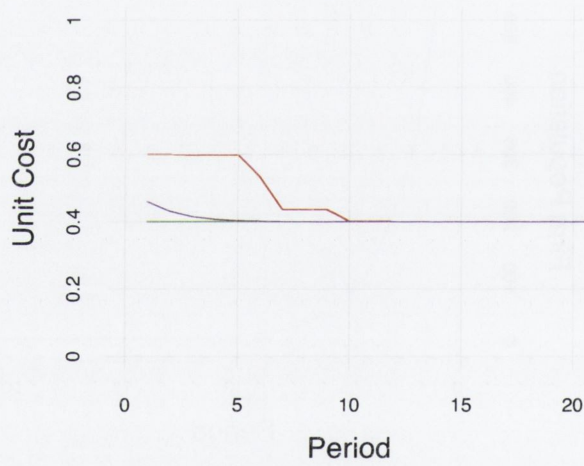


Figure 7.2: Time series of the maximum (red), minimum (green) and mean unit cost (purple) for $\phi = 0$.

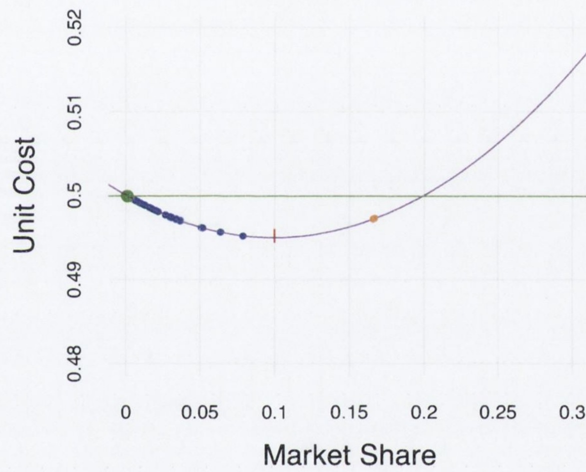


Figure 7.3: The purple curve represents the unit cost function. The green horizontal line represents the achievable singleton market cost. The red line indicates the optimal market share. The market share distribution after period 2 is shown in blue, and after period 15 in orange.

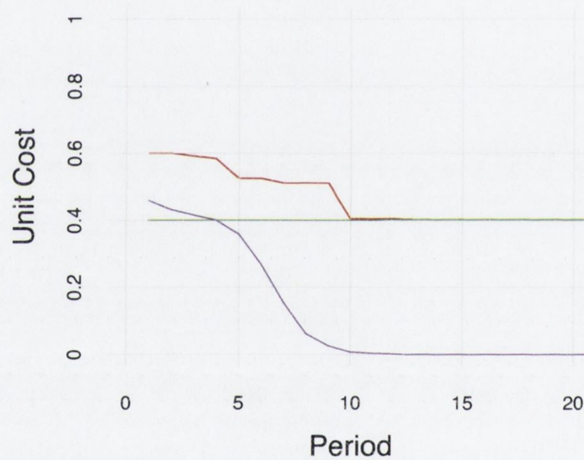


Figure 7.4: Time series of the maximum (red) and minimum (green) initial unit cost, and the mean effective unit cost (purple) for $\phi = 1$. The effective unit cost reaches 0 when the largest firm reaches a market share of 1.

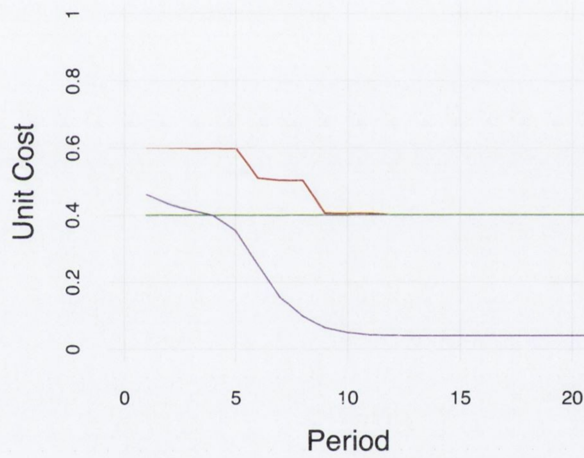


Figure 7.5: Time series of the maximum (red) and minimum (green) initial unit cost, and the mean effective unit cost (purple) for $\beta = 0.1$. The effective unit cost reaches a minimum value, which is greater than 0, when the largest firm reaches a market share of 1.

7.3 Discussion

The models in this chapter result in either a single firm containing all agents in the simulation, or uniformly-sized firms within which the population is evenly divided (or close to it). This is a very different outcome from our other models. Also noteworthy is that, as we progressed from constant returns to scale, to increasing returns to scale, to dynamically increasing returns to scale, ever more refined and subtle models, the outcome and dynamics of the agent-based simulations scarcely changed, despite the very different economic implications of these scenarios. The case of decreasing returns to scale provided another important illustration of how intuition about an analytical model can be misleading when the model is implemented in an agent-based framework.

Chapter 8

Conclusions

The variable effort model produces a skew distribution of firm sizes, we propose, due to the presence within the model of endogenous birth and proportional growth forces. These forces result from the instantiation within an agent-based model of contradictory microeconomic tendencies, namely the free riding tendency due to Cobb-Douglas income leisure preferences and the increasing returns to scale of the firm production function. All of these elements have a role to play in the resultant generation of a skew firm size distribution.

The agent-based model implementation provides an environment where information is gathered and decisions are made asynchronously, with myopia/local neighbourhood interaction. We saw that myopia was an important part of the variable effort model, when this was relaxed the skew distribution changed shape, first slightly and then more dramatically at extreme values such as $n = 50$. The agent-based implementation provided the scaffold which allowed the counteracting microeconomic tendencies to be translated into macroeconomic forces having an impact on simulated firms which resembled birth and proportional growth sufficiently to yield a skew firm size distribution.

An important possibility is that the specifics of the variable effort model are not of fundamental importance. This analysis does not suggest that the specific factors of income-leisure preferences and increasing returns to

scale are necessarily responsible for the skew distribution of firm sizes, but rather it suggests that any two (or more) microeconomic attributes which interact with each other in a similar way can result in similar macroeconomic behaviour. How do such factors need to interact with each other? The specifics can be determined experimentally (perhaps with another model implemented in this same framework), but a good starting point may be that the income-leisure preferences and increasing returns to scale can potentially interact both positively and negatively with each other. Innovation is a category which jumps to mind as having many microeconomic models and the potential for positive and negative interactions, and so may have the necessary dichotomous nature to lead to a skew distribution.

This necessary endogenous tension is reminiscent of the discovery in physics that a power law is a sign of a phase transition, and it is evocative of the concept that an economy demonstrating a skew distribution is poised in the complex and challenging border between order and chaos. If we take the view that these tensions are healthy and necessary, that an economy which becomes stable will soon become stagnant, then we are left with the implication that the inequality implied by a skew distribution is a fundamental fact of life. The Pareto distribution of income, the 80/20 rule, is not something which can ever be permanently remedied, although we may be able to alter its slope.

The cost curve and exogenous birth models each provided interesting counterexamples to the ideas brought out by the variable effort model. Exogenous birth, the blunt force solution proposed by Simon to counteract the monopolistic trends of a proportional growth regime, proved to be ineffective as an add-on. The cost curve model provided many interesting insights into what can change when an analytical or computational model is implemented in an agent-based framework. The model quickly reached a steady state, making it have limited interest compared with the rich turnover apparent in the variable effort model.

The unification, or at least reacquaintance, of the Gibrat's Law branch of the theory of the firm with the modern industrial organisation strand is a promising and fertile field for future research, with the aid of new

perspectives which can combine the positive attributes of both approaches. We can view Gibrat's Law as a consequence, rather than an assumption, of economic theory. The skew distribution of firm sizes can be seen as a stylised fact which should be explained by a model *in an agent-based context*, and not necessarily outside of that. Perhaps the industrial organisation theorists were right to ignore the skew distribution of firm sizes after all. It may not be an implication of microeconomic theory itself, but of the imperfect physical manifestation of that theory in an asynchronous, myopic world.

Bibliography

Adamic, Lada A. (2006). Zipf, Power-laws, and Pareto - a ranking tutorial. `http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html`.

Anonymous (c. 1000). Beowulf. `http://www.gutenberg.org/etext/981`.

Axtell, Robert (1999). The Emergence of Firms in a Population of Agents: Local Increasing Returns, Unstable Nash Equilibria, and Power Law Size Distributions. Brookings Center on Social and Economic Dynamics (CSED) Working Paper No. 3. `http://www.brookings.edu/es/dynamics/papers/firms/Firms.pdf`.

Barabási, Albert-László (2003). *Linked.* Plume.

Bi, Zhiqiang , Christos Faloutsos, and Flip Korn (2001, August). The "DGX" Distribution for Mining Massive, Skewed Data. In *The Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

Cefis, Elena , Matteo Ciccarelli, and Luigi Orsenigo (2004). Testing Gibrat's Legacy: A Bayesian Approach to Study the Growth of Firms. Technical Report 05-02, Tjalling C. Koopmans Research Institute.

Champernowne, D. G. (1952, October). The Graduation of Income Distribution. *Econometrica 20*(4), 591–615.

Dubrulle, B. , F. Graner, and D. Sornette (1997). Foreword. In B. Dubrulle, F. Graner, and D. Sornette (Eds.), *Scale Invariance and Beyond.* Springer.

Gabaix, Xavier (1999, August). Zipf's Law for Cities: An Explanation.

The Quarterly Journal of Economics 114(3), 739–767.

Gleick, James (1987). *Chaos.* Abacus.

Ijiri, Yuji and Herbert A. Simon (1977). *Skew Distributions and the Sizes of Business Firms.* Amsterdam: North Holland.

Kalecki, Michael (1945, April). On the Gibrat Distribution. *Econometrica 13*(2), 161–170.

Kauffman, Stuart A. (1993). *The Origins of Order.* Oxford University Press.

Krugman, Paul R. (1996). *The Self-Organizing Economy.* Blackwell.

Laherrère, Jean and Didier Sornette (1998). Stretched exponential distributions in nature and economy: "fat tails" with characteristic scales. *The European Physical Journal B*, **2**, 525–539.

Mazzucato, Mariana (2000). *Firm Size, Innovation and Market Structure.* Edward Elgar.

Mitzenmacher, Michael (2003). A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics 1*(2), 226–251.

Simon, Herbert A. (1955, December). On a Class of Skew Distribution Functions. *Biometrika 42*(3/4), 425–440.

Steindl, Josef (1965). *Random Processes and the Growth of Firms: A Study of the Pareto Law.* Griffin.

Sutton, John (1997, March). Gibrat's Legacy. *Journal of Economic Literature 35*(1), 40–59.

Sutton, John (1998). *Technology and Market Structure: Theory and History.* The MIT Press.

Tirole, Jean (1997). *The Theory of Industrial Organization.* The MIT Press.

Uetz, Peter and EMBL Heidelberg (2005). The EMBL Reptile Database. `http://www.reptile-database.org` Accessed 19 September 2005.

Weisstein, Eric W. (2005). Riemann Zeta Function. From Mathworld – A Wolfram Web Resource.

Yule, G. Udny (1925). A Mathematical Theory of Evolution, Based on the Conclusions of Dr. J. C. Willis, F.R.S. *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, **213**, 21–87.

Zipf, George Kingsley (1965). *Human Behavior and The Principle of Least Effort.* Hafner Publishing Company.

BIBLIOGRAPHY

Appendix A

Source Code

Contents

A.1 Framework **137**

A.1.1 SimpleFirmsModel.java 137

A.1.2 Employee.java 167

A.1.3 Firm.java 176

A.2 Data Collection **179**

A.2.1 CollectionSummary.java 179

A.2.2 MaxNumericDataSource.java 182

A.2.3 ArrayDataSource.java 183

A.2.4 DistributionDataSource.java 185

A.2.5 ArrayIntDistributionDataSource.java 187

A.2.6 GraphvizDataSource.java 189

A.3 Utilities **191**

A.3.1 BoundedDouble.java 191

A.3.2 Function.java 193

A.4 Variable Effort Model **199**

A.4.1 VariableEffortModel.java 199

A.4.2 VariableEffortEmployee.java 211

A.4.3 VariableEffortUtilityFunction.java 216

A.4.4 VariableEffortFirm.java 218

A.4.5 VariableEffortFirmOutputFunction.java 226

A.5 Exogenous Birth Model 227

A.5.1 ExogenousBirthModel.java 227

A.5.2 ExogenousBirthEmployee.java 230

A.5.3 ExogenousBirthFirm.java 234

A.6 Cost Curve Model 236

A.6.1 CostCurveModel.java 236

A.6.2 CostCurveEmployee.java 240

A.6.3 CostCurveFirm.java 241

A.6.4 CostCurveFirmCostFunction.java 244

A.1 Framework

The classes in this section: SimpleFirmsModel, Employee and Firm, form the core of the friendship-firms framework. They are intended to be subclassed with the model-specific behaviour, for example in the Variable Effort model they are subclassed by VariableEffort-Model, VariableEffortEmployee and VariableEffortFirm respectively. For some models, it might not be necessary to subclass all three, the default behaviour may suffice.

A.1.1 SimpleFirmsModel.java

```
package ie.tcd.economics.firms;
import java.lang.Runtime;
import java.lang.Process;
import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Date;
import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.engine.SimpleModel;

public class SimpleFirmsModel extends SimpleModel {
    /**
     * How many total employees/agents in simulation?
     */
    protected int numberEmployees;

    /**
     * How many friends per agent>
     */
    protected int numberFriends;

    /**
     * How many periods to run simulation for?
     */
```

```
protected int stopAtPeriod;

/**
 * How many employees per firm should we initialise?
 * Usually set to 1 to start in singleton firms.
 */
protected int initialFirmSize = 1;

/**
 * Define a time period during which highly detailed simulation data
 * are collected. Collecting detailed data for the entire simulation
 * is unnecessary and time-consuming.
 */
public int detailStartPeriod;
public int detailStopPeriod;

/**
 * RePast provides a general integer Tick, this customTick
 * lets us break each Tick into numberEmployees subticks.
 * So if we have 100 agents in the simulation, after the
 * first one is activated we will be at time 0.01, then 0.02 etc.
 * This allows for more detailed time series taking a snapshot
 * after each agent activation.
 */
protected double customTick;

/**
 * This is a variable to hold a desired value for the randomSeed
 * until it can be passed to the underlying RePast framework.
 */
private long randomSeed;

/**
```

```
 * A reference to all currently alive firms is stored in this
 * ArrayList. Used for iterating through all firms.
 */
protected ArrayList<Firm> firmsList;

/**
 * Should agents have the option to create a new (singleton) firm?
 */
protected boolean allowNewFirms = true;

/**
 * Iterate agents in sequence or randomly?
 */
protected boolean iterateSequentially = false;

/**
 * If agents can be forced to create new singleton firms,
 * with what probability will they create new firms?
 */
protected double forceNewSingletonProbability;

/**
 * Variables relating to data collection. Each DataRecorder has a
 * corresponding boolean preference for whether it is active or not,
 * allowing them to be easily switched off when not needed.
 */
private int newFirmsThisPeriod;
private int deadFirmsThisPeriod;

private int[] sizeOfFirmJoined;
private int[] sizeOfFirmLeft;
private int[] sizeOfFirmStayed;
```

```
protected boolean recordModelDetail = true;
protected DataRecorder recorder;

protected boolean recordFirmSizeDistribution = true;
protected int recordFirmSizeDistributionInterval = 1;
private DataRecorder firmSizeDistributionRecorder;

protected boolean recordCumulativeFirmSizeDistribution = true;
private DataRecorder cumulativeFirmSizeDistributionRecorder;

protected boolean recordFirmSizeJoinedLeftStayed = false;
private DataRecorder firmSizeJoinedRecorder;
private DataRecorder firmSizeLeftRecorder;
private DataRecorder firmSizeStayedRecorder;

public static boolean recordEmployeeTimeSeries = true;
public static DataRecorder employeeTimeSeriesRecorder;

public static boolean recordEmployeeCrossSection = true;
public DataRecorder employeeCrossSectionRecorder;

public static boolean recordEmployeeCrossSectionCaseStudy = true;
public DataRecorder employeeCrossSectionCaseStudyRecorder;

public boolean recordFirmCrossSection = true;
public DataRecorder firmCrossSectionRecorder;

public boolean recordFirmTimeSeries = false;
public int firmTimeSeriesInterval = 20;
public DataRecorder firmTimeSeriesRecorder;

public static boolean recordFirmTimeSeriesDetail = false;
```

```
public static boolean recordFriendshipNetwork = true;
public DataRecorder friendshipNetworkRecorder;

public static boolean recordFirmLifeStats = false;
public DataRecorder firmLifeStatsRecorder;
private double firmLifeStatAge;
private double firmLifeStatWho;
private double firmLifeStatMaxSize;

public static boolean recordFirmBirthStats = false;
public DataRecorder firmBirthStatsRecorder;

public static boolean recordFirmJoiningStats = true;
public DataRecorder firmJoiningStatsRecorder;
private double firmJoiningStatSize;
private int firmJoiningStatWhoEmployee;
private int firmJoiningStatWhoFirm;

public static boolean recordEmployeeDecisionStats = true;
public DataRecorder employeeDecisionStatsRecorder;
protected double employeeDecisionStatWho;
protected double employeeDecisionStatFirmStayWho;
protected double employeeDecisionStatFirmStaySize;
protected double employeeDecisionStatStayUtility;
protected double employeeDecisionStatFirmJoin1Who;
protected double employeeDecisionStatFirmJoin1Size;
protected double employeeDecisionStatFirmJoin1Utility;
protected double employeeDecisionStatFirmJoin2Who;
protected double employeeDecisionStatFirmJoin2Size;
protected double employeeDecisionStatFirmJoin2Utility;
protected double employeeDecisionStatFirmCreateUtility;

protected int numberStayingCurrentFirm;
```

```
protected int numberCreatingNewFirm;
protected int numberJoiningFriendsFirm;

/**
 * getInitParam() usually returns an array of strings specifying
 * parameters. Here we extend this to allow subclassing models to
 * add their own params to list.
 *
 * mainInitParam() gives parameters relevant to SimpleFirmsModel
 * additionalInitParams() reads the parameters added by subclassing
 * models.
 */
public String[] mainInitParam() {
    String[] params = {"modelName", "codeVersion",
        "numberEmployees", "numberFriends", "stopAtPeriod",
        "randomSeed", "initialFirmSize", "allowNewFirms",
        "iterateSequentially", "forceNewSingletonProbability",
        "detailStartPeriod", "detailStopPeriod"};
    return params;
}

public String[] getInitParam() {
    return mainInitParam();
}

public String[] additionalInitParams(String[] additionalParams)
{
    String[] mainParams = mainInitParam();
    String[] params = new String[mainParams.length +
        additionalParams.length];
    System.arraycopy(mainParams, 0, params, 0, mainParams.length);
    System.arraycopy(additionalParams, 0, params,
        mainParams.length, additionalParams.length);
```

```
        return params;
    }

    /**
     * Initialise randomSeed here,
     * if we set it as a preference later it will overwrite this value.
     */
    public void setup() {
        super.setup();
        randomSeed = System.currentTimeMillis();
    }

    public void buildModel() {
        if (recordFirmSizeJoinedLeftStayed) {
            sizeOfFirmJoined = new int[numberEmployees];
            sizeOfFirmLeft = new int[numberEmployees];
            sizeOfFirmStayed = new int[numberEmployees];
        }

        // Set the RePast time period at which to stop.
        this.setStoppingTime(stopAtPeriod);
        // Set the RePast RandomSeed.
        this.setRngSeed(randomSeed);

        firmsList = new ArrayList();

        initializeDataRecorders();

        // Gives compiler error if not initialised.
        Firm firmToJoin = null;

        for (int i = 0; i < numberEmployees; i++) {
            Employee emp = createEmployee(i);
```

```
            agentList.add(emp);
            if (i % initialFirmSize == 0) {
                emp.createAndJoinFirm();
                firmToJoin = emp.firm;
            } else {
                if (firmToJoin == null) {
                    throw new RuntimeException(
                        "firmToJoin not initialised!");
                } else {
                    emp.joinFirm(firmToJoin);
                }
            }
        }

        for (Iterator it = agentList.iterator(); it.hasNext() ; ) {
            ((Employee)it.next()).makeFriends();
        }
        stepDataRecorders();
    }

    /**
     * Used primarily in testing where we don't want to initialise data recorders
     * and where they cause exceptions in test if not switched off.
     */
    public void disableRecorders() {
        recordModelDetail = false;
        recordFirmSizeDistribution = false;
        recordCumulativeFirmSizeDistribution = false;
        recordEmployeeTimeSeries = false;
        recordEmployeeCrossSection = false;
        recordEmployeeCrossSectionCaseStudy = false;
        recordFirmCrossSection = false;
        recordFirmTimeSeries = false;
```

```
        recordFriendshipNetwork = false;
        recordFirmSizeJoinedLeftStayed = false;
        recordFirmLifeStats = false;
        recordFirmBirthStats = false;
        recordFirmJoiningStats = false;
        recordEmployeeDecisionStats = false;
    }

    /**
     * This method should be overwritten in subclass.
     */
    public Employee createEmployee(int i) {
        System.out.println(
            "Overwrite createEmployee() in your model.");
        return new Employee(this, i);
    }

    public void preStep() {
        customTick = getTickCount();
        newFirmsThisPeriod = 0;
        deadFirmsThisPeriod = 0;
        numberStayingCurrentFirm = 0;
        numberCreatingNewFirm = 0;
        numberJoiningFriendsFirm = 0;

        if (iterateSequentially) {
            for (int i = 0; i < agentList.size(); i++) {
                getAgent(i).activate();
                customTick += 1.0/agentList.size();
                if (recordFirmTimeSeries &&
                    recordFirmTimeSeriesDetail && inDetailPeriod())
                {
                    firmTimeSeriesRecorder.record();
```

```
                }
            }
        } else {
            //Randomly select agents for activation.
            for (int i = 0; i < agentList.size(); i++) {
                int j = getNextIntFromTo(0, agentList.size()-1);
                getAgent(j).activate();
                customTick += 1.0/agentList.size();
                if (recordFirmTimeSeries &&
                    recordFirmTimeSeriesDetail && inDetailPeriod())
                {
                    firmTimeSeriesRecorder.record();
                }
            }
        }
    }

    public void step() {
        for (Iterator it = firmsList.iterator(); it.hasNext(); ) {
            ((Firm)it.next()).step();
        }
    }

    public void postStep() {
        stepDataRecorders();
        if (this.getTickCount() % 50 == 0 ||inDetailPeriod()) {
            writeDataRecorders();
        }
    }

    public void atEnd() {
        writeDataRecorders();
    }
```

```
public void destroyFirm(Firm exFirm) {
    firmsList.remove(exFirm);
    if (recordFirmLifeStats) {
        firmLifeStatAge = exFirm.getExactAge();
        firmLifeStatWho = exFirm.getWho();
        firmLifeStatMaxSize = exFirm.getMaxSize();
        firmLifeStatsRecorder.record();
    }
    deadFirmsThisPeriod++;
}

public void addFirm(Firm newFirm) {
    firmsList.add(newFirm);
    if (recordFirmBirthStats) {
        firmBirthStatsRecorder.record();
    }
    newFirmsThisPeriod++;
}

//Accessors

public String getModelName() {
    return this.getClass().getName();
}

/**
 * This method gathers the Subversion code version number which is stored
 * as a parameter and printed out in data files, so that the version of
 * code used to produce a given batch is known.
 */
public String getCodeVersion() {
    int exit = -1;
```

```
        String line = "";
        String resultStr = "";
        Runtime rtime = Runtime.getRuntime();

        try {
            Process child = rtime.exec("svnversion .");

            InputStreamReader inputStreamReader = new
                InputStreamReader(child.getInputStream());
            BufferedReader reader = new
                BufferedReader(inputStreamReader);

            do {
                resultStr += line;
                line = reader.readLine();
            } while (line != null);
            reader.close();

            exit = child.waitFor();

        } catch (Exception ex) {
            resultStr = "Not Detected";
        }
        return resultStr;
    }

    /**
     * Get the agent at position i in the agentList.
     */
    public Employee getAgent(int i){
        return (Employee)agentList.get(i);
    }
```

```
    public int getNumberEmployees() {
        return numberEmployees;
    }

    public void setNumberEmployees(int newNumberEmployees) {
        numberEmployees = newNumberEmployees;
    }

    public int getNumberFriends() {
        return numberFriends;
    }

    public void setNumberFriends(int newNumberFriends) {
        numberFriends = newNumberFriends;
    }

    public int getStopAtPeriod() {
        return stopAtPeriod;
    }

    public void setStopAtPeriod(int argStopAtPeriod) {
        stopAtPeriod = argStopAtPeriod;
    }

    public int getNewFirmsThisPeriod() {
        return newFirmsThisPeriod;
    }

    public int getDeadFirmsThisPeriod() {
        return deadFirmsThisPeriod;
    }

    public int getNumberFirms() {
```

```
        return firmsList.size();
    }

    public void setRandomSeed(long newRandomSeed) {
        randomSeed = newRandomSeed;
    }

    public long getRandomSeed() {
        return randomSeed;
    }

    public double getNumberStayingCurrentFirm() {
        return numberStayingCurrentFirm;
    }

    public double getNumberCreatingNewFirm() {
        return numberCreatingNewFirm;
    }

    public double getNumberJoiningFriendsFirm() {
        return numberJoiningFriendsFirm;
    }

    public double getCustomTick() {
        return customTick;
    }

    public void setInitialFirmSize(int argInitialFirmSize) {
        initialFirmSize = argInitialFirmSize;
    }

    public int getInitialFirmSize() {
        return initialFirmSize;
```

```
    }

    public void setAllowNewFirms(boolean argAllowNewFirms) {
        allowNewFirms = argAllowNewFirms;
    }

    public boolean getAllowNewFirms() {
        return allowNewFirms;
    }

    public void setIterateSequentially(boolean argIterateSequentially)
    {
        iterateSequentially = argIterateSequentially;
    }

    public boolean getIterateSequentially() {
        return iterateSequentially;
    }

    public void addSizeOfFirmJoined(int sizeOfFirm) {
        if (recordFirmSizeJoinedLeftStayed) {
            sizeOfFirmJoined[sizeOfFirm]++;
        }
    }

    public void addSizeOfFirmLeft(int sizeOfFirm) {
        if (recordFirmSizeJoinedLeftStayed) {
            sizeOfFirmLeft[sizeOfFirm]++;
        }
    }

    public void addSizeOfFirmStayed(int sizeOfFirm) {
        if (recordFirmSizeJoinedLeftStayed) {
```

```
            sizeOfFirmStayed[sizeOfFirm]++;
        }
    }

    public double getFirmLifeStatAge() {
        return firmLifeStatAge;
    }

    public double getFirmLifeStatWho() {
        return firmLifeStatWho;
    }

    public double getFirmLifeStatMaxSize() {
        return firmLifeStatMaxSize;
    }

    public void setForceNewSingletonProbability(double
        newForceNewSingletonProbability) {
        forceNewSingletonProbability =
            newForceNewSingletonProbability;
    }

    public double getForceNewSingletonProbability() {
        return forceNewSingletonProbability;
    }

    public boolean inDetailPeriod() {
        return this.getTickCount() > detailStartPeriod &&
            this.getTickCount() < detailStopPeriod;
    }

    public boolean decisionStats() {
        return recordEmployeeDecisionStats && inDetailPeriod();
```

```
}

public void recordDecisionStats() {
    employeeDecisionStatsRecorder.record();
}

public void setDetailStartPeriod(int newDetailStartPeriod) {
    detailStartPeriod = newDetailStartPeriod;
}

public int getDetailStartPeriod() {
    return detailStartPeriod;
}

public void setDetailStopPeriod(int newDetailStopPeriod) {
    detailStopPeriod = newDetailStopPeriod;
}

public int getDetailStopPeriod() {
    return detailStopPeriod;
}

public void setFirmJoiningStatSize(double newFirmJoiningStatSize)
{
    firmJoiningStatSize = newFirmJoiningStatSize;
}

public double getFirmJoiningStatSize() {
    return firmJoiningStatSize;
}

public void setFirmJoiningStatWhoEmployee(
    int newFirmJoiningStatWhoEmployee) {
```

```
        firmJoiningStatWhoEmployee = newFirmJoiningStatWhoEmployee;
    }

    public double getFirmJoiningStatWhoEmployee() {
        return firmJoiningStatWhoEmployee;
    }

    public void setFirmJoiningStatWhoFirm(int newFirmJoiningStatWhoFirm)
    {
        firmJoiningStatWhoFirm = newFirmJoiningStatWhoFirm;
    }

    public double getFirmJoiningStatWhoFirm() {
        return firmJoiningStatWhoFirm;
    }

    public double getEmployeeDecisionStatWho() {
        return employeeDecisionStatWho;
    }
    public double getEmployeeDecisionStatFirmStayWho() {
        return employeeDecisionStatFirmStayWho;
    }
    public double getEmployeeDecisionStatFirmStaySize() {
        return employeeDecisionStatFirmStaySize;
    }
    public double getEmployeeDecisionStatStayUtility() {
        return employeeDecisionStatStayUtility;
    }
    public double getEmployeeDecisionStatFirmJoin1Who() {
        return employeeDecisionStatFirmJoin1Who;
    }
    public double getEmployeeDecisionStatFirmJoin1Size() {
        return employeeDecisionStatFirmJoin1Size;
```

```
    }
    public double getEmployeeDecisionStatFirmJoin1Utility() {
        return employeeDecisionStatFirmJoin1Utility;
    }
    public double getEmployeeDecisionStatFirmJoin2Who() {
        return employeeDecisionStatFirmJoin2Who;
    }
    public double getEmployeeDecisionStatFirmJoin2Size() {
        return employeeDecisionStatFirmJoin2Size;
    }
    public double getEmployeeDecisionStatFirmJoin2Utility() {
        return employeeDecisionStatFirmJoin2Utility;
    }
    public double getEmployeeDecisionStatFirmCreateUtility() {
        return employeeDecisionStatFirmCreateUtility;
    }

    //Data Recorders

    public void initializeDataRecorders() throws
        NullPointerException {
        long filenameTimeStamp = System.currentTimeMillis();

        if (recordModelDetail) {

        recorder = new DataRecorder("./output/"+
            filenameTimeStamp + "_model.txt",
            this, "Random Seed:\t"+ getRngSeed());
        recorder.createNumericDataSource(
            "number_of_firms", firmsList, "size");
        recorder.createNumericDataSource(
            "new_firms", this, "getNewFirmsThisPeriod");
        recorder.createNumericDataSource(
```

```
        "dead_firms", this, "getDeadFirmsThisPeriod");


    recorder.addNumericDataSource("total_employees", new
        TotalNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
          return (double)((Firm)argObject).getSizeForStatCollector();
        }
    });

    recorder.addNumericDataSource("max_firm_size", new
        MaxNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
          return (double)((Firm)argObject).getSizeForStatCollector();
        }
    });

    recorder.addNumericDataSource("max_firm_age", new
        MaxNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((Firm)argObject).getAge();
        }
    });

    recorder.addNumericDataSource("avg_firm_age", new
        AverageNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((Firm)argObject).getAge();
        }
    });

    recorder.addNumericDataSource("avg_firm_size", new
        AverageNumericDataSource(firmsList) {
```

```
        public double getValue(Object argObject) {
            return (double)((Firm)argObject)
            .getSizeForStatCollector();
        }
    });

    recorder.addNumericDataSource("max_utility", new
        MaxNumericDataSource(agentList) {
        public double getValue(Object argObject) {
            return (double)((Employee)argObject).getUtility();
        }
    });

    recorder.addNumericDataSource("min_utility", new
        MinNumericDataSource(agentList) {
        public double getValue(Object argObject) {
            return (double)((Employee)argObject).getUtility();
        }
    });

    recorder.addNumericDataSource("avg_utility", new
        AverageNumericDataSource(agentList) {
        public double getValue(Object argObject) {
            return (double)((Employee)argObject).getUtility();
        }
    });

    recorder.createNumericDataSource("number_staying_current_firm",
        this, "getNumberStayingCurrentFirm");
    recorder.createNumericDataSource("number_creating_new_firm",
        this, "getNumberCreatingNewFirm");
    recorder.createNumericDataSource("number_joining_friends_firm",
        this, "getNumberJoiningFriendsFirm");
```

```
recorder.addNumericDataSource("cohabitationRate", new
    AverageNumericDataSource(agentList) {
    public double getValue(Object argObject) {
        return (double)((Employee)argObject)
        .getCohabitationRate();
    }
});

}

if (recordEmployeeTimeSeries) {
    employeeTimeSeriesRecorder = new DataRecorder("./output/"+
        filenameTimeStamp + "_employee_time_series.txt",
        this, "Random Seed:\t"+ getRngSeed());
}

if (recordEmployeeCrossSection) {
    employeeCrossSectionRecorder = new DataRecorder("./output/"+
        filenameTimeStamp + "_employee_cross_section.txt",
        this, "Random Seed:\t"+ getRngSeed());
}

if (recordEmployeeCrossSectionCaseStudy) {
    employeeCrossSectionCaseStudyRecorder = new
        DataRecorder("./output/"+ filenameTimeStamp
        + "_employee_cross_section_case_study.txt",
        this, "Random Seed:\t"+ getRngSeed());
}

if (recordFirmCrossSection) {
    firmCrossSectionRecorder = new DataRecorder("./output/"+
        filenameTimeStamp + "_firm_cross_section.txt",
```

```
            this, "Random Seed:\t"+ getRngSeed());
        }

        if (recordFirmTimeSeries) {
            firmTimeSeriesRecorder = new DataRecorder("./output/"+
                filenameTimeStamp + "_firm_time_series.txt",
                this, "Random Seed:\t"+ getRngSeed());
            if (recordFirmTimeSeriesDetail) {
                firmTimeSeriesRecorder.createNumericDataSource(
                "custom_tick", this, "getCustomTick");
            }
        }

        if (recordFirmSizeDistribution) {
            firmSizeDistributionRecorder = new DataRecorder("./output/"+
                filenameTimeStamp + "_firm_size_distribution.txt",
                this, "Random Seed:\t"+ getRngSeed());
            firmSizeDistributionRecorder.addObjectDataSource("",
                new DistributionDataSource(
                firmsList, numberEmployees, false, true) {
                public int getValue(Object argObject) {
                    return ((Firm)argObject).getSize();
                }
            });
        }

        if (recordCumulativeFirmSizeDistribution) {
            cumulativeFirmSizeDistributionRecorder = new DataRecorder(
                "./output/"+ filenameTimeStamp +
                "_cum_firm_size_distribution.txt",
                this, "Random Seed:\t"+ getRngSeed());
            cumulativeFirmSizeDistributionRecorder.addObjectDataSource(
                "", new DistributionDataSource(
```

```
                firmsList, numberEmployees, true, true) {
                public int getValue(Object argObject) {
                    return ((Firm)argObject).getSize();
                }
            });
        }

        if (recordFirmSizeJoinedLeftStayed) {
            firmSizeJoinedRecorder = new DataRecorder("./output/"+
            filenameTimeStamp + "_firm_size_joined.txt",
            this, "Random Seed:\t"+ getRngSeed());
            firmSizeJoinedRecorder.addObjectDataSource("", new
                ArrayIntDistributionDataSource(sizeOfFirmJoined,
                numberEmployees, "Size of Firm Joined"));

            firmSizeLeftRecorder = new DataRecorder("./output/"+
                filenameTimeStamp + "_firm_size_left.txt",
                this, "Random Seed:\t"+ getRngSeed());
            firmSizeLeftRecorder.addObjectDataSource("", new
                ArrayIntDistributionDataSource(sizeOfFirmLeft,
                numberEmployees, "Size of Firm Left"));

            firmSizeStayedRecorder = new DataRecorder("./output/"+
                filenameTimeStamp + "_firm_size_stayed.txt",
                this, "Random Seed:\t"+ getRngSeed());
            firmSizeStayedRecorder.addObjectDataSource("", new
                ArrayIntDistributionDataSource(sizeOfFirmStayed,
                numberEmployees, "Size of Firm Stayed"));
        }

        if (recordFriendshipNetwork) {
            friendshipNetworkRecorder = new DataRecorder("./output/"+
                filenameTimeStamp + "_friendship_network.txt",
```

```
            this, "Random Seed:\t"+ getRngSeed());
        friendshipNetworkRecorder.addObjectDataSource("", new
            GraphvizDataSource(agentList, "Friendship Network"));
    }

    if (recordFirmLifeStats) {
        firmLifeStatsRecorder = new DataRecorder("./output/"+
            filenameTimeStamp + "_firm_life_stats.txt",
            this, "Random Seed:\t"+ getRngSeed());
        firmLifeStatsRecorder.createNumericDataSource("age",
            this, "getFirmLifeStatAge");
        firmLifeStatsRecorder.createNumericDataSource("who",
            this, "getFirmLifeStatWho");
        firmLifeStatsRecorder.createNumericDataSource("maxSize",
            this, "getFirmLifeStatMaxSize");
    }

    if (recordFirmBirthStats) {
        firmBirthStatsRecorder = new DataRecorder("./output/"+
            filenameTimeStamp + "_firm_birth_stats.txt",
            this, "Random Seed:\t"+ getRngSeed());
    }

    if (recordFirmJoiningStats) {
        firmJoiningStatsRecorder = new DataRecorder("./output/"+
            filenameTimeStamp + "_firm_joining_stats.txt",
            this, "Random Seed:\t"+ getRngSeed());
        firmJoiningStatsRecorder.createNumericDataSource("size",
            this, "getFirmJoiningStatSize");
        firmJoiningStatsRecorder.createNumericDataSource("employeeWho",
            this, "getFirmJoiningStatWhoEmployee");
        firmJoiningStatsRecorder.createNumericDataSource("firmWho",
            this, "getFirmJoiningStatWhoFirm");
```

```
        }

        if (recordEmployeeDecisionStats) {
            employeeDecisionStatsRecorder = new DataRecorder(
                "./output/"+ filenameTimeStamp +
                "_employee_decision_stats.txt",
                this, "Random Seed:\t"+ getRngSeed());
            employeeDecisionStatsRecorder.createNumericDataSource(
                "customTick", this, "getCustomTick");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "who", this, "getEmployeeDecisionStatWho");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmStayWho", this,
                "getEmployeeDecisionStatFirmStayWho");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmStaySize", this,
                "getEmployeeDecisionStatFirmStaySize");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmStayUtility", this,
                "getEmployeeDecisionStatStayUtility");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmJoin1Who", this,
                "getEmployeeDecisionStatFirmJoin1Who");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmJoin1Size", this,
                "getEmployeeDecisionStatFirmJoin1Size");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmJoin1Utility", this,
                "getEmployeeDecisionStatFirmJoin1Utility");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmJoin2Who", this,
                "getEmployeeDecisionStatFirmJoin2Who");
            employeeDecisionStatsRecorder.createNumericDataSource(
```

```
                "firmJoin2Size", this,
                "getEmployeeDecisionStatFirmJoin2Size");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmJoin2Utility", this,
                "getEmployeeDecisionStatFirmJoin2Utility");
            employeeDecisionStatsRecorder.createNumericDataSource(
                "firmCreateUtility", this,
                "getEmployeeDecisionStatFirmCreateUtility");
        }

    }

    public void stepDataRecorders() {
        if (recordModelDetail) {
            recorder.record();
        }

        if (recordEmployeeTimeSeries) {
            employeeTimeSeriesRecorder.record();
        }

        if (recordEmployeeCrossSection &&
            this.getTickCount() % 100 == 0) {
            employeeCrossSectionRecorder.record();
        }

        if (recordEmployeeCrossSectionCaseStudy &&
            inDetailPeriod()) {
            employeeCrossSectionCaseStudyRecorder.record();
        }

        if (recordFirmTimeSeries) {
            firmTimeSeriesRecorder.record();
```

```
        }

        if (recordFirmCrossSection &&
            this.getTickCount() % 100 == 0) {
            firmCrossSectionRecorder.record();
        }

        if (recordFirmSizeDistribution &&
            this.getTickCount() %
                recordFirmSizeDistributionInterval == 0) {
            firmSizeDistributionRecorder.record();
        }

        if (recordCumulativeFirmSizeDistribution) {
            cumulativeFirmSizeDistributionRecorder.record();
        }

        if (recordFirmSizeJoinedLeftStayed) {
            firmSizeJoinedRecorder.record();
            firmSizeLeftRecorder.record();
            firmSizeStayedRecorder.record();
        }

        if (recordFriendshipNetwork && this.getTickCount() == 1) {
            friendshipNetworkRecorder.record();
        }

    }

    public void writeDataRecorders() {
        if (recordModelDetail) {
            recorder.writeToFile();
        }
```

```
        if (recordEmployeeTimeSeries) {
            employeeTimeSeriesRecorder.writeToFile();
        }

        if (recordEmployeeCrossSection) {
            employeeCrossSectionRecorder.writeToFile();
        }

        if (recordEmployeeCrossSectionCaseStudy) {
            employeeCrossSectionCaseStudyRecorder.writeToFile();
        }

        if (recordFirmTimeSeries) {
            firmTimeSeriesRecorder.writeToFile();
        }

         if (recordFirmCrossSection) {
             firmCrossSectionRecorder.writeToFile();
         }

        if (recordFirmSizeDistribution) {
            firmSizeDistributionRecorder.writeToFile();
        }

        if (recordCumulativeFirmSizeDistribution) {
            cumulativeFirmSizeDistributionRecorder.writeToFile();
        }

        if (recordFirmSizeJoinedLeftStayed) {
            firmSizeJoinedRecorder.writeToFile();
            firmSizeLeftRecorder.writeToFile();
            firmSizeStayedRecorder.writeToFile();
```

```
        }

        if (recordFriendshipNetwork) {
            friendshipNetworkRecorder.writeToFile();
        }

        if (recordFirmLifeStats) {
            firmLifeStatsRecorder.writeToFile();
        }

        if (recordFirmBirthStats) {
            firmBirthStatsRecorder.writeToFile();
        }

        if (recordFirmJoiningStats) {
            firmJoiningStatsRecorder.writeToFile();
        }
    }
}
```

A.1.2 Employee.java

```
package ie.tcd.economics.firms;

import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.analysis.NumericDataSource;

public class Employee {
    protected Firm firm;
    protected Integer[] friends;
    protected SimpleFirmsModel model;
    protected int who;

    protected static final int STAY_CURRENT_FIRM_OPTION = 0;
    protected static final int CREATE_NEW_FIRM_OPTION = -1;
    protected static final int JOIN_EXISTING_FIRM_OPTION = -2;

    public Employee(SimpleFirmsModel newModel, int i) {
        who = i;
        model = newModel;
        dataRecorders();
    }

    public void activate() {
        joinMaximumUtilityFirm();
    }

    /**
     * Optionally override in subclass.
     */
    public double calculateCurrentUtility() {
        return calculateMaximumUtilityAtFirm(firm);
    }
```

```
/**
 * Override in subclass.
 */
public double calculateMaximumUtilityAtFirm(Firm firm) {
    return 0;
}

/**
 * Employees are members of friendship networks. This method initializes
 * that friendship network by creating one-way friendship links to randomly
 * selected employees within the model. The number of unique friends
 * assigned to each employee is a model parameter.
 */
public void makeFriends() {
    if (model.numberFriends > model.numberEmployees - 1) {
        throw new RuntimeException(
        "Must have fewer friends than employees in simulation.");
    }
    friends = new Integer[model.numberFriends];
    for (int i = 0; i < friends.length; i++) {
        Integer friendCandidate;
        boolean newFriend;
        do {
            friendCandidate = new Integer(model.getNextIntFromTo(
                0, model.numberEmployees - 1));
            newFriend = true;
            for (int j = 0; j < i; j++) {
                if (friends[j].equals(friendCandidate)) {
                    newFriend = false;
                    break;
                }
            }
        } while ((friendCandidate.intValue() == who) ||!newFriend);
```

```
            friends[i] = friendCandidate;
        }
    }

    /**
     * Employees join the firm in which they can achieve the highest utility.
     * They examine all available firms (their current firm, firms of all friends,
     * a hypothetical singleton firm they can create), and choose the best
     * option.
     */
    protected void joinMaximumUtilityFirm() {
        double maxUtility = 0;
        double utility = 0;

        int option = STAY_CURRENT_FIRM_OPTION;
        Firm firmToJoin = null;

        if (!forceNewSingleton()) {
            //Utility at the current firm.
            maxUtility = calculateMaximumUtilityAtFirm(firm);

            if (model.decisionStats()) {
                model.employeeDecisionStatFirmStayWho = firm.getWho();
                model.employeeDecisionStatFirmStaySize = firm.getSize();
                model.employeeDecisionStatStayUtility = maxUtility;
            }

            //Utility at a new singleton firm.
            //If we are already a singleton no need to do this.
            if (model.allowNewFirms && firm.getSize() != 1) {
                Firm temporaryFirm = createNewFirm();
                utility = calculateMaximumUtilityAtFirm(temporaryFirm);
                temporaryFirm = null;
```

```
        if (utility > maxUtility) {
            option = CREATE_NEW_FIRM_OPTION;
            maxUtility = utility;
        }

        if (model.decisionStats()) {
            model.employeeDecisionStatFirmCreateUtility =
                utility;
        }
    }

    for (int i = 0; i < friends.length; i++) {
        Firm friendsFirm = getFriend(i).firm;
        if (!friendsFirm.equals(firm)) {
            utility = calculateMaximumUtilityAtFirm(friendsFirm);

            if (utility > maxUtility) {
                option = JOIN_EXISTING_FIRM_OPTION;
                firmToJoin = friendsFirm;
                maxUtility = utility;
            }

            if (model.decisionStats()) {
                switch (i) {
               case 0:
                model.employeeDecisionStatFirmJoin1Who =
                    friendsFirm.getWho();
                model.employeeDecisionStatFirmJoin1Size =
                    friendsFirm.getSize();
                model.employeeDecisionStatFirmJoin1Utility
                    = utility;
               break;
```

```
                    case 1:
                     model.employeeDecisionStatFirmJoin2Who =
                         friendsFirm.getWho();
                     model.employeeDecisionStatFirmJoin2Size =
                         friendsFirm.getSize();
                     model.employeeDecisionStatFirmJoin2Utility
                         = utility;
                    break;
                     }
               }
          }
     }
} else {
     //Force a new singleton firm.
     if (model.allowNewFirms && firm.getSize() != 1) {
          option = CREATE_NEW_FIRM_OPTION;
     }
}

switch (option) {
     case CREATE_NEW_FIRM_OPTION:
          model.addSizeOfFirmLeft(firm.getSize());
          createAndJoinFirm();
          model.numberCreatingNewFirm++;
          model.addSizeOfFirmJoined(1);
          break;

     case JOIN_EXISTING_FIRM_OPTION:
          model.addSizeOfFirmLeft(firm.getSize());
          joinFirm(firmToJoin);
          model.numberJoiningFriendsFirm++;
          model.addSizeOfFirmJoined(firmToJoin.getSize());
          break;
```

```
            case STAY_CURRENT_FIRM_OPTION:
                model.addSizeOfFirmStayed(firm.getSize());
                model.numberStayingCurrentFirm++;
        }

        if (model.decisionStats()) {
            model.employeeDecisionStatWho = who;
            model.employeeDecisionStatsRecorder.record();
        }
    }


    /**
     * Leave current firm, if any, and join the firm passed as argument.
     * @params newFirm Firm to be joined.
     */
    public void joinFirm(Firm newFirm) {
        if (firm != null) {
            if (firm.equals(newFirm)) {
                throw new RuntimeException(
                "Employee trying to join a firm they are already in.");
            }
            firm.loseEmployee(this);
        }
        this.firm = newFirm;
        newFirm.gainEmployee(this);
        if (model.inDetailPeriod()) {
            model.setFirmJoiningStatWhoEmployee(who);
            model.setFirmJoiningStatWhoFirm(newFirm.getWho());
            model.setFirmJoiningStatSize(newFirm.getSize());
            model.firmJoiningStatsRecorder.record();
        }
```

```
    }

    /**
     * Create a new firm and join it.
     */
    public void createAndJoinFirm() {
        Firm newFirm = createNewFirm();
        this.joinFirm(newFirm);
        model.addFirm(newFirm);
    }

    /**
     * Employees are responsible for creating new firms. Hence each subclass
     * of Employee must override createNewFirm to create the corresponding
     * subclass of Firm. This should only be called from createAndJoinFirm().
     *
     * Maybe make private?
     */
    public Firm createNewFirm() {
        return new Firm(model);
    }

    protected boolean forceNewSingleton() {
        return model.getNextDoubleFromTo(0,1) <
            model.forceNewSingletonProbability;
    }

    /**
    * What percentage of my neighbours are in the same firm as me?
    */
    private double cohabitationRate() {
        double total = 0;
        for (int i = 0; i < friends.length; i++) {
```

```
            Firm friendsFirm = getFriend(i).firm;
            if (friendsFirm.equals(firm)) {
                total++;
            }
        }
        return total/(double)friends.length;
    }

    //Accessors

    public double getUtility() {
        return calculateCurrentUtility();
    }

    public Employee getFriend(int i) {
        return model.getAgent(friends[i].intValue());
    }

    public double getWho() {
        return who;
    }

    public double getCohabitationRate() {
        return cohabitationRate();
    }

    //Data Recorders

    public void dataRecorders() {
        if (model.recordEmployeeTimeSeries &&
            who % (model.numberEmployees/20) == 0) {
            model.employeeTimeSeriesRecorder.addNumericDataSource(
                "choice_"+ who, new FirmChoice());
```

```
            model.employeeTimeSeriesRecorder.addNumericDataSource(
                "utility_"+ who, new EmployeeUtility());
        }
    }

    class FirmChoice implements NumericDataSource {
        public double execute() {
            return -999;
        }
    }

    class EmployeeUtility implements NumericDataSource {
        public double execute() {
            return calculateCurrentUtility();
        }
    }
}
```

A.1.3 Firm.java

```
package ie.tcd.economics.firms;

import java.util.ArrayList;

public class Firm {
    protected SimpleFirmsModel model;
    private int age = 0;
    private int size = 0;
    private int maxSize = 0;
    private double bornOnCustomTick;
    protected ArrayList<Employee> employees = new ArrayList();
    protected int who;
    protected static int whoCounter = 0;

    public Firm(SimpleFirmsModel newModel) {
        model = newModel;
        who = whoCounter++;
        bornOnCustomTick = model.customTick;
    }

    public void gainEmployee(Employee newEmployee) {
        if (employees.contains(newEmployee)) {
            throw new RuntimeException(
            "Trying to add employee to a firm they're already in.");
        }
        employees.add(newEmployee);
        if (calculateSize() > maxSize) {
            maxSize = calculateSize();
        }
    }

    public void loseEmployee(Employee exEmployee) {
```

```
        if (!employees.contains(exEmployee)) {
            throw new RuntimeException(
            "Trying to remove employee from a firm they're not in.");
        }
        employees.remove(exEmployee);
        if (employees.size() == 0) {
            model.destroyFirm(this);
        }
    }

    public void step() {
        age++;
        size = calculateSize();
    }

    //Calculators

    private int calculateSize() {
        return employees.size();
    }

    private int calculateSizeExcludingEmployee(Employee emp) {
        if (employees.contains(emp)) {
            return calculateSize() - 1;
        } else {
            return calculateSize();
        }
    }

    //Accessors

    public int getAge() {
        return age;
```

```
    }

    public int getSize() {
        return calculateSize();
    }

    public int getSizeExcludingEmployee(Employee emp) {
        return calculateSizeExcludingEmployee(emp);
    }

    public int getSizeForStatCollector() {
        return size;
    }

    public boolean containsEmployee(Employee emp) {
        return employees.contains(emp);
    }

    public int getMaxSize() {
        return maxSize;
    }

    public int getWho() {
        return who;
    }

    public double getExactAge() {
        return model.customTick - bornOnCustomTick;
    }

}
```

A.2 Data Collection

The classes in this section are concerned with the collection of data during simulation runs and outputting that data to text files. The CollectionSummary class is subclassed by MaxNumericDataSource, MinNumericDataSource, AverageNumericDataSource and TotalNumericDataSource. Only MaxNumericDataSource is shown here as the others are trivially similar. As the names suggest, these classes collect summary data. Generally they will summarise over the list of employees or the list of firms, so for example they might calculate the mean employee utility or the maximum firm output.

The ArrayDataSource class, by contrast, does not summarise data but records it in detail. It also operates on a list such as the list of employees or the list of firms, but it will record every data point rather than aggregating them. The DistributionDataSource and ArrayIntDistributionDataSource classes produce size distribution tables, i.e. Frequency-Count data. The GraphvizDataSource produces output which can be imported into the GraphViz application with minimal processing, this is used to record the friendship network and to subsequently plot it as a network with GraphViz. This is impractical when the number of agents is greater than 100.

A.2.1 CollectionSummary.java

```
package ie.tcd.economics.firms;

import java.util.Collection;
import java.util.Iterator;

public class CollectionSummary{
    Collection objectCollection;

    public CollectionSummary(Collection argObjectCollection) {
        objectCollection = argObjectCollection;
    }

    public double getTotal() {
       double totValue = 0;
```

```
        for (Iterator it = objectCollection.iterator(); it.hasNext()
         ;) {
            double nextValue = getValue(it.next());
            totValue += nextValue;
         }
        return totValue;
    }

    public double getAverage() {
        if (objectCollection.size() == 0) {
            throw new RuntimeException(
            "Trying to take an average with no items in collection.");
        }
        return getTotal()/objectCollection.size();
    }

    public double getMinimum() {
        if (objectCollection.size() == 0) {
            throw new RuntimeException(
            "Trying to calculate minimum with no items in collection.");
        }
        //Set to very large value so we are sure to be less than this.
        double minValue = Double.MAX_VALUE;
        for (Iterator it = objectCollection.iterator(); it.hasNext()
         ;) {
            double nextValue = getValue(it.next());
            if (nextValue < minValue) {
                minValue = nextValue;
            }
        }
        return minValue;
    }
```

```
    public double getMaximum() {
        if (objectCollection.size() == 0) {
            throw new RuntimeException(
            "Trying to calculate maximum with no items in collection.");
        }
        //Set to very small value so we are sure to be greater than this.
        double maxValue = -Double.MAX_VALUE;
        for (Iterator it = objectCollection.iterator(); it.hasNext()
        ;) {
            double nextValue = getValue(it.next());
            if (nextValue > maxValue) {
                maxValue = nextValue;
            }
        }
        return maxValue;
    }

    /**
     * Overwrite this method in subclass.
     * The value you wish to take the min/max/total/average etc. of.
     */
    public double getValue(Object collectionObject) {
        return 0.0;
    }
}
```

A.2.2 MaxNumericDataSource.java

```
package ie.tcd.economics.firms;

import java.util.List;
import uchicago.src.sim.analysis.NumericDataSource;

public class MaxNumericDataSource extends CollectionSummary
    implements NumericDataSource {

    public MaxNumericDataSource(List argObjectList) {
        super(argObjectList);
    }

    public double execute() {
       return getMaximum();
    }

}
```

A.2.3 ArrayDataSource.java

```
package ie.tcd.economics.firms;


import java.util.List;
import java.util.ListIterator;
import uchicago.src.sim.analysis.DataSource;


/**
 * A data collection class designed to output a customisable array of values.
 * Iterates over a list, prints each value separated by a comma.
 */
public class ArrayDataSource implements DataSource {
    List objectList;
    String name;

    public ArrayDataSource(List argObjectList, String argName) {
        name = argName;
        objectList = argObjectList;
    }

    public Object execute() {
        StringBuffer buf = new StringBuffer();

        buf.append("\n"+this.name+"\n");
        ListIterator it = objectList.listIterator();
        buf.append(getValue(it.next()));
        for (;it.hasNext();) {
            buf.append(","+ getValue(it.next()));
        }
        buf.append("\n");
        return buf.toString();
    }
```

```
    /**
     * Overwrite this method in subclass.
     */
    public double getValue(Object listObject) {
        return 0;
    }
}
```

A.2.4 DistributionDataSource.java

```
package ie.tcd.economics.firms;


import java.util.List;
import java.util.ListIterator;
import uchicago.src.sim.analysis.DataSource;


/**
 * Calculates a value distribution (Frequency, Count data) from a collection
 * of raw observations and prints to a file.
 */


public class DistributionDataSource implements DataSource {
    List objectList;
    int maxSize;
    boolean cumulative;
    boolean displaySeparateTable;
    int[] distributionArray;

    public DistributionDataSource(List argObjectList, int argMaxSize,
        boolean argCumulative, boolean argDisplaySeparateTable) {
        this.objectList = argObjectList;
        this.maxSize = argMaxSize;
        this.cumulative = argCumulative;
        this.displaySeparateTable = argDisplaySeparateTable;
    }

    public Object execute() {
        StringBuffer buf = new StringBuffer();

        if (!cumulative ||distributionArray == null) {
            distributionArray = new int[maxSize+1];
        }
```

```
        for (ListIterator it = objectList.listIterator(); it.hasNext()
            ;) {
            distributionArray[getValue(it.next())]++;
        }

        if (displaySeparateTable) {
            buf.append("\n\nFrequency\tCount");
            for (int i = 1; i <= maxSize; i++) {
                if (distributionArray[i]>0) {
                    buf.append("\n"+ i + "\t"+ distributionArray[i]);
                }
            }
            buf.append("\n");
        } else {
            buf.append(distributionArray[0]);
            for (int i = 1; i <= maxSize; i++) {
                buf.append(",");
                buf.append(distributionArray[i]);
            }
        }
        return buf.toString();
    }

    /**
     * Overwrite this method in subclass.
     */
    public int getValue(Object listObject) {
        return 0;
    }
}
```

A.2.5 ArrayIntDistributionDataSource.java

```
package ie.tcd.economics.firms;


import java.util.List;
import java.util.ListIterator;
import uchicago.src.sim.analysis.DataSource;


/**
 * Prints a 1-dimensional array to a table with the array index
 * as Frequency and the stored value as Count.
 *
 * For an example of an array to be printed using this class
 * see SimpleFirmsModel.addSizeOfFirmJoined() and related.
 */
public class ArrayIntDistributionDataSource implements DataSource {
    int[] intArray;
    String name;
    int maxSize;

    public ArrayIntDistributionDataSource(int[] argIntArray,
        int argMaxSize, String argName) {
        name = argName;
        intArray = argIntArray;
        maxSize = argMaxSize;
    }

    public Object execute() {
        StringBuffer buf = new StringBuffer();

        buf.append("\n\nFrequency\tCount");
        for (int i = 1; i <= maxSize; i++) {
            if (intArray[i]>0) {
                buf.append("\n"+ i + "\t"+ intArray[i]);
```

```
            }
        }
        buf.append("\n");

        return buf.toString();
    }
}
```

A.2.6 GraphvizDataSource.java

```
package ie.tcd.economics.firms;


import java.util.List;
import java.util.ListIterator;
import uchicago.src.sim.analysis.DataSource;


public class GraphvizDataSource implements DataSource {
    List objectList;
    String name;

    public GraphvizDataSource(List argObjectList, String argName) {
        name = argName;
        objectList = argObjectList;
    }

    public Object execute() {
        StringBuffer buf = new StringBuffer();
        buf.append("\ndigraph untitled {\n");
        for(ListIterator it = objectList.listIterator(); it.hasNext();){
            Employee thisEmployee = (Employee)it.next();
            for(int i = 0; i < thisEmployee.model.numberFriends ; i++)
            {
                buf.append(thisEmployee.who+" -> "+
                    thisEmployee.friends[i]+"\n");
            }
        }

        buf.append("}\n\n");
        buf.append("---\n");
        return buf.toString();
    }
}
```

A.2.6. GraphvizDataSource.java

A.3 Utilities

A.3.1 BoundedDouble.java

```
package ie.tcd.economics.firms;

/**
 * A convenience wrapper for the double class which ensures that values are kept
 * in an appropriate range, for example (0,1).
 */

public class BoundedDouble {
    private double value;
    final private double MAX_VALUE;
    final private double MIN_VALUE;

    /** Creates a new instance of BoundedDouble */
    public BoundedDouble(double argMinValue, double argMaxValue) {
        if (argMinValue < argMaxValue) {
            MAX_VALUE = argMaxValue;
            MIN_VALUE = argMinValue;

            //set value to an illegal value.
            value = MIN_VALUE - 1;
        } else {
            throw new RuntimeException(
                "Max Value must be structly greater than Min Value.");
        }
    }

    public BoundedDouble(double argMinValue, double argMaxValue,
        double initializeToValue) {
        this(argMinValue, argMaxValue);
        this.setValue(initializeToValue);
```

```
    }

    public double getValue() {
        if (value < MIN_VALUE) {
            throw new RuntimeException(
                "Value has not been initialized.");
        } else {
            return value;
        }
    }

    public void setValue(double argValue) {
        if (argValue < MIN_VALUE) {
            throw new RuntimeException("Value "+ argValue +
                " must be greater than minimum of "+ MIN_VALUE);
        } else if (argValue > MAX_VALUE) {
            throw new RuntimeException("Value "+ argValue +
                " must be less than maximum of "+ MAX_VALUE);
        } else {
            value = argValue;
        }
    }

    public double getMax() {
        return MAX_VALUE;
    }

    public double getMin() {
        return MIN_VALUE;
    }
}
```

A.3.2 Function.java

```
/*
 * (c) 1998-2000 The Brookings Institution. All Rights Reserved
 *
 * Permission to use this software and its documentation for non-commercial
 * purposes and without fee is hereby granted, provided this copyright statement
 * is included. Please contact us for permission for redistribution and other uses.
 *
 * BROOKINGS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
 * OR NON-INFRINGEMENT. BROOKINGS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * Miles Parker
 * mparker@brook.edu
 * http://www.brook.edu/es/dynamics/models/ascape/
 * The Brookings Institution
 * Washington, D.C.
 */

package ie.tcd.economics.firms;

import java.io.Serializable;

/**
 * A one-dimensional function.
 *
```

```
 * @author Miles Parker
 * @version 1.0
 * @since 1.0
 */
public abstract class Function implements Serializable, Cloneable {

    /**
     * Tau "magic" number. approx. 1.61803
     */
    public static final double tau = (1 + Math.sqrt(5)) / 2;

    /**
     * Large end of golden section. approx. .61803
     */
    public static final double sectLarge = tau - 1;

    /**
     * Small end of golden section. approx. .38197
     */
    public static final double sectSmall = 1 - sectLarge;

    /**
     * Desired resolution of maximization.
     */
    public static final double resolution = 0.01;

    /**
     * The first measurement of the current interval.
     * For minor performance reasons, this and other measurements are not
     * initialized in the body of the maximization functions.
     * This can easily be changed if desired.
     * Also, please note that this class is _not_thread safe; in order to make it
     * so, simply initialize the following variables within a contructor (or in the method
```

```java
     * body), make them non-static, and synchronize the methods as appropriate.
     */
protected static double x1 = 0.0;

/**
     * The second measurement of the current interval.
     */
protected static double x2 = 0.0;

/**
     * The third measurement of the current interval.
     */
protected static double x3 = 0.0;

/**
     * The fourth measurement of the current interval.
     */
protected static double x4 = 0.0;

/**
     * The result value for the second measurement.
     */
protected static double f2 = 0.0;

/**
     * The result value for the third measurement.
     */
protected static double f3 = 0.0;

    /**
     * The X axis gap between the first and second measurements
     */
    protected double gap1 = 0;
```

```
/**
 * The X axis gap between the second and third measurements
 * (after one measurement has been dropped, leaving three total.)
 */
protected double gap2 = 0;


/**
 * Maximize the output of this function, assuming function is unimodal,
 * using a golden section search strategy.
 * See Press, Flannery, Teukolsky, Vetterling _Numerical Recipes in *_10.1
 * for general guidelines, but not for specific implementation.
 * @return the optimal input variable
 */
public double maximize() {
 x1 = 0.0;
 x2 = sectSmall;
 x3 = sectLarge;
 x4 = 1.0;
 double gap = 1.0;
 //For exploring number of iterations.
 //int iter = 0;
 while (gap > resolution) {
  f2 = solveFor(x2);
  f3 = solveFor(x3);
  if (f2 < f3) {
   gap1 = x3 - x2;
   gap2 = x4 - x3;
x1 = x2;
   if (gap1 > gap2) {
    x2 = x3 - gap1 * sectSmall;
    f2 = solveFor(x2);
    gap = gap1;
```

```
        }
        else {  //gap[1] <= gap1
         x2 = x3;
         x3 = x2 + gap2 * sectSmall;
         f2 = f3;
         f3 = solveFor(x2);
         gap = gap2;
        }
       }
       else {  //y[3] <= y[0]
        gap1 = x2 - x1;
        gap2 = x3 - x2;
  x4 = x3;
        if (gap1 > gap2) {
         x3 = x2;
         x2 = x3 - gap1 * sectSmall;
         f2 = solveFor(x2);
         gap = gap1;
        }
        else {  //gap[1] <= gap1
         x3 = x2 + gap2 * sectSmall;
         f2 = f3;
         f3 = solveFor(x2);
         gap = gap2;
        }
       }
      }
  return f2 > f3 ?  x2 :  x3;
  }

  /**
   * Solve this (single-variable) function. Override to define your own function.
   * @param x the variable input parameter
```

```
     * @return the output value
     */
    public double solveFor(double x) {
     return x;
    }

    /**
     * Clones this function.
     */
    public Object clone () {
        try {
            Function clone = (Function) super.clone();
            return clone;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }
}
```

A.4 Variable Effort Model

A.4.1 VariableEffortModel.java

```
package ie.tcd.economics.firms;
import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.engine.SimInit;

public class VariableEffortModel extends SimpleFirmsModel {
    private double minTheta = 0;
    private double maxTheta = 1;
    protected boolean useMaxTheta = true;

    private double minLambda = 1;
    private double maxLambda = 1;
    protected boolean useMaxLambda = true;

    protected int firmToWatchOne;
    protected int firmToWatchTwo;

    private double firmLifeStatFoundersTheta;
    private int firmLifeStatFoundersWho;

    private double firmBirthStatFoundersTheta;
    private int firmBirthStatFoundersWho;

    public VariableEffortModel() {
        super();
        allowNewFirms = true;
    }

    public static void main(String[] args) {
        SimInit init = new SimInit();
        init.loadModel(new VariableEffortModel(), null, false);
```

```
}

public String[] getInitParam() {
    String[] params = {"minTheta", "maxTheta", "useMaxTheta",
        "minLambda", "firmToWatchOne", "firmToWatchTwo"};
    return super.additionalInitParams(params);
}

public void buildModel() {
    super.buildModel();
    additionalDataRecorders();
    VariableEffortFirm.firmsToWatch.add(new Integer(
        getFirmToWatchOne()));
    VariableEffortFirm.firmsToWatch.add(new Integer(
        getFirmToWatchTwo()));
}

public Employee createEmployee(int i) {
    return new VariableEffortEmployee(this, i);
}

public void destroyFirm(Firm exFirm) {
    firmLifeStatFoundersTheta =
        ((VariableEffortFirm)exFirm).getFoundersTheta();
    firmLifeStatFoundersWho =
        ((VariableEffortFirm)exFirm).getFoundersWho();
    super.destroyFirm(exFirm);
}

public void addFirm(Firm newFirm) {
    firmBirthStatFoundersTheta =
        ((VariableEffortFirm)newFirm).getFoundersTheta();
    firmBirthStatFoundersWho =
```

```
            ((VariableEffortFirm)newFirm).getFoundersWho();
        super.addFirm(newFirm);
    }

    //Accessors

    public double getMinTheta() {
        return minTheta;
    }

    public void setMinTheta(double argMinTheta) {
        minTheta = argMinTheta;
    }

    public double getMaxTheta() {
        if (useMaxTheta) {
            return maxTheta;
        } else {
            return 1 - minTheta;
        }
    }

    public void setMaxTheta(double argMaxTheta) {
        maxTheta = argMaxTheta;
    }

    public boolean getUseMaxTheta() {
        return useMaxTheta;
    }

    public void setUseMaxTheta(boolean argUseMaxTheta) {
        useMaxTheta = argUseMaxTheta;
    }
```

```
public double getMinLambda() {
    return minLambda;
}

public void setMinLambda(double argMinLambda) {
    minLambda = argMinLambda;
}

public double getMaxLambda() {
    if (useMaxLambda) {
        return maxLambda;
    } else {
        return 1 - minLambda;
    }
}

public void setMaxLambda(double argMaxLambda) {
    maxLambda = argMaxLambda;
}

public int getFirmToWatchOne() {
    return firmToWatchOne;
}

public void setFirmToWatchOne(int argFirmToWatchOne) {
    firmToWatchOne = argFirmToWatchOne;
}

public int getFirmToWatchTwo() {
    return firmToWatchTwo;
}
```

```
public void setFirmToWatchTwo(int argFirmToWatchTwo) {
    firmToWatchTwo = argFirmToWatchTwo;
}

public double getFirmLifeStatFoundersTheta() {
    return firmLifeStatFoundersTheta;
}

public double getFirmLifeStatFoundersWho() {
    return firmLifeStatFoundersWho;
}

public double getFirmBirthStatFoundersTheta() {
    return firmBirthStatFoundersTheta;
}

public double getFirmBirthStatFoundersWho() {
    return firmBirthStatFoundersWho;
}

//Data Recorders

public void additionalDataRecorders() {
    if (recordModelDetail) {
     recorder.addNumericDataSource("max_effort", new
        MaxNumericDataSource(agentList){
         public double getValue(Object argObject) {
             return (double)((VariableEffortEmployee)argObject)
             .getEffort();
         }
     });

     recorder.addNumericDataSource("min_effort", new
```

```
    MinNumericDataSource(agentList) {
     public double getValue(Object argObject) {
         return (double)((VariableEffortEmployee)argObject)
         .getEffort();
     }
});

recorder.addNumericDataSource("avg_effort", new
    AverageNumericDataSource(agentList) {
     public double getValue(Object argObject) {
         return (double)((VariableEffortEmployee)argObject)
         .getEffort();
     }
});

recorder.addNumericDataSource("total_effort", new
    TotalNumericDataSource(agentList) {
     public double getValue(Object argObject) {
         return (double)((VariableEffortEmployee)argObject)
         .getEffort();
     }
});

recorder.addNumericDataSource("max_output", new
    MaxNumericDataSource(firmsList) {
     public double getValue(Object argObject) {
         return (double)((VariableEffortFirm)argObject)
         .getOutput();
     }
});

recorder.addNumericDataSource("min_output", new
    MinNumericDataSource(firmsList) {
```

```
        public double getValue(Object argObject) {
            return (double)((VariableEffortFirm)argObject)
            .getOutput();
        }
    });

    recorder.addNumericDataSource("avg_output", new
       AverageNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((VariableEffortFirm)argObject)
            .getOutput();
        }
    });

    recorder.addNumericDataSource("total_output", new
       TotalNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((VariableEffortFirm)argObject)
            .getOutput();
        }
    });

    recorder.addNumericDataSource("max_efficiency", new
       MaxNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((VariableEffortFirm)argObject)
            .getEfficiency();
        }
    });

    recorder.addNumericDataSource("min_efficiency", new
       MinNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
```

```
            return (double)((VariableEffortFirm)argObject)
            .getEfficiency();
        }
    });

    recorder.addNumericDataSource("avg_efficiency", new
       AverageNumericDataSource(firmsList) {
        public double getValue(Object argObject) {
            return (double)((VariableEffortFirm)argObject)
            .getEfficiency();
        }
    });
    }

    if (recordFirmCrossSection) {
        this.firmCrossSectionRecorder.addObjectDataSource("", new
           ArrayDataSource(firmsList, "Output") {
            public double getValue(Object argObject) {
                return ((VariableEffortFirm)argObject).getOutput();
            }
        });
        this.firmCrossSectionRecorder.addObjectDataSource("", new
           ArrayDataSource(firmsList, "Size") {
            public double getValue(Object argObject) {
                return ((VariableEffortFirm)argObject).getSize();
            }
        });
        this.firmCrossSectionRecorder.addObjectDataSource("", new
           ArrayDataSource(firmsList, "Effort") {
            public double getValue(Object argObject) {
                return ((VariableEffortFirm)argObject)
                .getTotalRealizedEffort();
            }
```

```
        });
    }

    if (recordEmployeeCrossSection) {
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "Who") {
            public double getValue(Object argObject) {
                return ((Employee)argObject).getWho();
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "Theta") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getTheta();
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "Lambda") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getLambda();
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "Effort") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getEffort();
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
           new ArrayDataSource(agentList, "Utility") {
```

```
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getUtility();
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "Firm") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject).firm.who;
            }
        });
        this.employeeCrossSectionRecorder.addObjectDataSource("",
            new ArrayDataSource(agentList, "FirmSize") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject).firm
                .getSize();
            }
        });
    }

    if (recordEmployeeCrossSectionCaseStudy) {
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Who") {
            public double getValue(Object argObject) {
                return ((Employee)argObject).getWho();
            }
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Theta") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getTheta();
            }
```

```
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Lambda") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getLambda();
            }
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Effort") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getEffort();
            }
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Utility") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject)
                .getUtility();
            }
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "Firm") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject).firm.who;
            }
        });
        this.employeeCrossSectionCaseStudyRecorder.addObjectDataSource(
            "", new ArrayDataSource(agentList, "FirmSize") {
            public double getValue(Object argObject) {
                return ((VariableEffortEmployee)argObject).firm
                .getSize();
```

```
                }
            });
        }

        if (recordFirmLifeStats) {
            firmLifeStatsRecorder.createNumericDataSource(
                "foundersTheta", this,
                "getFirmLifeStatFoundersTheta");
            firmLifeStatsRecorder.createNumericDataSource(
                "foundersWho", this,
                "getFirmLifeStatFoundersWho");
        }

        if (recordFirmBirthStats) {
            firmBirthStatsRecorder.createNumericDataSource(
                "foundersTheta", this,
                "getFirmBirthStatFoundersTheta");
            firmBirthStatsRecorder.createNumericDataSource(
                "foundersWho", this,
                "getFirmBirthStatFoundersWho");
        }
    }
}
```

A.4.2 VariableEffortEmployee.java

```
package ie.tcd.economics.firms;

import uchicago.src.sim.analysis.NumericDataSource;


public class VariableEffortEmployee extends Employee {
    private BoundedDouble currentEffort = new BoundedDouble(0,1,0);
    private BoundedDouble theta = new BoundedDouble(0,1);
    private BoundedDouble lambda = new BoundedDouble(0,1);

    public VariableEffortEmployee(
        VariableEffortModel newModel, int i) {
        super(newModel, i);
        theta.setValue(newModel.getNextDoubleFromTo(
            newModel.getMinTheta(),
            newModel.getMaxTheta()));
        lambda.setValue(newModel.getNextDoubleFromTo(
            newModel.getMinLambda(),
            newModel.getMaxLambda()));
        additionalDataRecorders();
    }

    public Firm createNewFirm() {
        return new VariableEffortFirm(model);
    }

    public void activate() {
        setEffort(0);
        super.activate();
        calculateAndSetCurrentEffort();
    }

    //Calculators
```

```
public double calculateMaximumUtilityAtFirm(Firm argFirm) {
    VariableEffortUtilityFunction utilityFunction = new
        VariableEffortUtilityFunction(argFirm, this);
    return utilityFunction.utility(
        utilityFunction.effortForMaximumUtility());
}

public double calculateCurrentUtility() {
    return new VariableEffortUtilityFunction(firm, this)
        .utility(currentEffort.getValue());
}

public double calculateSingletonUtility() {
    VariableEffortUtilityFunction utilityFunction = new
        VariableEffortUtilityFunction(0, 0, getTheta(), getLambda());
    return utilityFunction.utility(
        utilityFunction.effortForMaximumUtility());
}

public double betterOffThanSingleton() {
    if (calculateCurrentUtility() >= calculateSingletonUtility()) {
        // Better off.
        return 1;
    } else {
        // Worse off! We'd be better off on our own.
        return 0;
    }
}

public void calculateAndSetCurrentEffort() {
    setEffort(new VariableEffortUtilityFunction(firm, this)
        .effortForMaximumUtility());
}
```

```
public double calculateFriendsTheta() {
    double total = 0;
    for (int i = 0; i < model.numberFriends; i++) {
        VariableEffortEmployee friend = (VariableEffortEmployee)
            model.getAgent(this.friends[i].intValue());
        total += friend.getTheta();
    }
    return total;
}

//Accessors

public double getEffort() {
    return currentEffort.getValue();
}

public void setEffort(double argEffort) {
    ((VariableEffortFirm)firm)
        .subtractContribution(getRealizedEffort());
    currentEffort.setValue(argEffort);
    ((VariableEffortFirm)firm).addContribution(getRealizedEffort());
}

public double getRealizedEffort() {
    return currentEffort.getValue() * lambda.getValue();
}

public double getTheta() {
    return theta.getValue();
}

public double getLambda() {
```

```
        return lambda.getValue();
    }

    public double getFriendsTheta() {
        return calculateFriendsTheta();
    }

    //Data Recorders

    public void additionalDataRecorders() {
        if (model.recordEmployeeTimeSeries &&
            who % (model.numberEmployees/20) == 0) {
            model.employeeTimeSeriesRecorder.addNumericDataSource(
                "theta_"+who, new EmployeeTheta());
            model.employeeTimeSeriesRecorder.addNumericDataSource(
                "effort_"+who, new EmployeeEffort());
            model.employeeTimeSeriesRecorder.addNumericDataSource(
                "lambda_"+who, new EmployeeLambda());
        }
    }

    class EmployeeTheta implements NumericDataSource {
        public double execute() {
            return getTheta();
        }
    }

    class EmployeeLambda implements NumericDataSource {
        public double execute() {
            return getLambda();
        }
    }
```

```
    class FriendsTheta implements NumericDataSource {
        public double execute() {
            return getFriendsTheta();
        }
    }

    class EmployeeEffort implements NumericDataSource {
        public double execute() {
            return getEffort();
        }
    }

}
```

A.4.3 VariableEffortUtilityFunction.java

```
package ie.tcd.economics.firms;

public class VariableEffortUtilityFunction extends UtilityFunction {
    protected double restOfFirmRealizedEffort;
    protected double restOfFirmSize;
    protected BoundedDouble theta = new BoundedDouble(0,1);
    protected BoundedDouble lambda = new BoundedDouble(0,1);

    public VariableEffortUtilityFunction(
        double argRestOfFirmRealizedEffort,
        double argRestOfFirmSize, double argTheta, double argLambda)
    {
        restOfFirmRealizedEffort = argRestOfFirmRealizedEffort;
        restOfFirmSize = argRestOfFirmSize;
        theta.setValue(argTheta);
        lambda.setValue(argLambda);
    }

    public VariableEffortUtilityFunction(Firm firm,
        VariableEffortEmployee emp) {
        this(((VariableEffortFirm)firm)
            .getTotalRealizedEffortExcludingEmployee(emp),
            ((VariableEffortFirm)firm).getSizeExcludingEmployee(emp),
            emp.getTheta(), emp.getLambda());
    }

    public double effortForMaximumUtility() {
        return this.maximize();
    }

    public double solveFor(double argEffort) {
        BoundedDouble e = new BoundedDouble(0,1);
```

```
        BoundedDouble re = new BoundedDouble(0,1);

        //Assigning to a BoundedDouble automatically tests for proper values.
        e.setValue(argEffort);
        re.setValue(argEffort * lambda.getValue());
        return Math.pow((VariableEffortFirmOutputFunction.getOutput(
            restOfFirmRealizedEffort + re.getValue()) /
                (restOfFirmSize + 1)),
            theta.getValue()) * Math.pow(1 - e.getValue() ,
                1 - theta.getValue());
    }
}
```

A.4.4 VariableEffortFirm.java

```
package ie.tcd.economics.firms;

import java.util.Iterator;
import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.analysis.NumericDataSource;
import java.util.ArrayList;

public class VariableEffortFirm extends Firm {
    private double cachedRealizedEffort = 0;
    private static double defaultTolerance = 0.00000000001;
    protected static ArrayList<Integer> firmsToWatch = new ArrayList();
    private double foundersTheta;
    private int foundersWho;

    public VariableEffortFirm(SimpleFirmsModel newModel) {
        super(newModel);
        if (VariableEffortFirm.firmsToWatch.contains(new Integer(who))){
            additionalDataRecorders();
        }
    }

    public void step() {
        super.step();
        double calculateRealizedEffort =
            calculateTotalRealizedEffort();
        if (Math.abs(calculateRealizedEffort - cachedRealizedEffort)
            < 0.001) {
            cachedRealizedEffort = calculateRealizedEffort;
        }
    }

    public void failOnCachedRealizedEffortDiscrepancy(
```

```
        double tolerance) {
        if (!checkCachedEffortDiscrepancy(tolerance)) {
            throw new RuntimeException("Cached effort "+
                cachedRealizedEffort +
                " out of sync with actual effort "+
                calculateTotalRealizedEffort());
        }
    }

    public void failOnCachedRealizedEffortDiscrepancy() {
        failOnCachedRealizedEffortDiscrepancy(
            VariableEffortFirm.defaultTolerance);
    }

    public void gainEmployee(Employee newEmployee) {
        if (getSize()==0 && getAge()==0) {
            foundersTheta = ((VariableEffortEmployee)newEmployee)
                .getTheta();
            foundersWho = ((VariableEffortEmployee)newEmployee).who;
        }
        super.gainEmployee(newEmployee);
    }

    //Calculators

    private double calculateEfficiency() {
        double totalEffort = getTotalRealizedEffort();
        return VariableEffortFirmOutputFunction
            .getOutput(totalEffort)/totalEffort;
    }

    private double calculateTotalOutput() {
        return VariableEffortFirmOutputFunction
```

```
            .getOutput(getTotalRealizedEffort());
    }

    private double calculateTotalRealizedEffort() {
        double result = 0;
        for (Iterator it = employees.iterator(); it.hasNext(); ) {
            result += ((VariableEffortEmployee) it.next())
                .getRealizedEffort();
        }
        return result;
    }

    private double calculateTotalRealizedEffortExcludingEmployee(
        VariableEffortEmployee emp) {
        double result = 0;
        if (employees.contains(emp)) {
            result = getTotalRealizedEffort() - emp.getRealizedEffort();
            if (getSize() == 1 && result >
                VariableEffortFirm.defaultTolerance) {
                throw new RuntimeException(
                    "Nonzero rest of firm effort in singleton firm.");
            }
        } else {
            result = getTotalRealizedEffort();
        }
        return result;
    }

    private double cachedEffortDiscrepancy() {
        return Math.abs(cachedRealizedEffort -
            calculateTotalRealizedEffort());
    }
```

```
    private double calculateAverageUtility() {
        if (getSize() == 0) {
            return 0;
        } else {
            double result = 0;
            for (Iterator it = employees.iterator(); it.hasNext();)
            {
                result += ((VariableEffortEmployee) it.next())
                    .calculateCurrentUtility();
            }
            return result / getSize();
        }
    }

    private double calculatePercentBetterOffThanSingleon() {
        if (getSize() == 0) {
            return 0;
        } else {
            double result = 0;
            for (Iterator it = employees.iterator(); it.hasNext();)
            {
                result += ((VariableEffortEmployee) it.next())
                    .betterOffThanSingleton();
            }
            return result / getSize();
        }
    }

    private double calculateAverageTheta() {
        if (getSize() == 0) {
            return 0;
        } else {
            double result = 0;
```

```
        for (Iterator it = employees.iterator(); it.hasNext();)
        {
            result += ((VariableEffortEmployee) it.next())
                .getTheta();
        }
        return result / getSize();
    }
}

//Accessors

public double getEfficiency() {
    return calculateEfficiency();
}

public double getOutput() {
    return calculateTotalOutput();
}

public double getTotalRealizedEffort() {
    return cachedRealizedEffort;
}

public double getTotalRealizedEffortExcludingEmployee(
    VariableEffortEmployee emp) {
    return calculateTotalRealizedEffortExcludingEmployee(emp);
}

public void addContribution(double contribution) {
    cachedRealizedEffort += contribution;
}

public void subtractContribution(double contribution) {
```

```
        cachedRealizedEffort -= contribution;
    }

    public boolean checkCachedEffortDiscrepancy(double tolerance) {
        return cachedEffortDiscrepancy() < tolerance;
    }

    public boolean checkCachedEffortDiscrepancy() {
        return checkCachedEffortDiscrepancy(
            VariableEffortFirm.defaultTolerance);
    }

    public double getFoundersTheta() {
        return foundersTheta;
    }

    public int getFoundersWho() {
        return foundersWho;
    }

    //Data Recorders

    public void additionalDataRecorders() {
        if (model.recordFirmTimeSeries) {
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "firm_size_"+ who, new FirmSize());
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "total_effort_"+ who, new TotalEffort());
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "average_employee_utility_"+ who, new
                AverageUtility());
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "average_employee_theta_"+ who, new AverageTheta());
```

```
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "percent_better_off_than_singleton_"+ who, new
                PercentBetterOffThanSingleton());
        }
    }

    class FirmSize implements NumericDataSource {
        public double execute() {
            try {
                return VariableEffortFirm.this.getSize();
            } catch (NullPointerException ex) {
                return 0;
            }
        }
    }

    class TotalEffort implements NumericDataSource {
        public double execute() {
            try {
                return VariableEffortFirm.this.getTotalRealizedEffort();
            } catch (NullPointerException ex) {
                return 0;
            }
        }
    }

    class AverageUtility implements NumericDataSource {
        public double execute() {
            try {
                return VariableEffortFirm.this.calculateAverageUtility();
            } catch (NullPointerException ex) {
                return 0;
            }
```

```
        }
    }

    class AverageTheta implements NumericDataSource {
        public double execute() {
            try {
                return VariableEffortFirm.this.calculateAverageTheta();
            } catch (NullPointerException ex) {
                return 0;
            }
        }
    }

    class PercentBetterOffThanSingleton implements NumericDataSource {
        public double execute() {
            try {
                return VariableEffortFirm.this.
                    calculatePercentBetterOffThanSingleon();
            } catch (NullPointerException ex) {
                return 0;
            }
        }
    }
}
```

A.4.5 VariableEffortFirmOutputFunction.java

```
package ie.tcd.economics.firms;

public class VariableEffortFirmOutputFunction {
    final static protected double A = 1;
    final static protected double B = 1;
    final static protected int EXP = 2;

    public static double getOutput(double effort) {
        if (effort < 0 && Math.abs(effort) > 0.0000000000001) {
            throw new RuntimeException(
        "Calling VariableEffortFirmOutputFunction.getOutput"
        + " with negative effort.");
        }
        return A * effort + B * Math.pow(effort, EXP);
    }
}
```

A.5 Exogenous Birth Model

A.5.1 ExogenousBirthModel.java

```
package ie.tcd.economics.firms;

public class ExogenousBirthModel extends SimpleFirmsModel {

    public ExogenousBirthModel() {
        super();
    }

    public void buildModel() {
        super.buildModel();
        additionalDataRecorders();
    }

    public Employee createEmployee(int i) {
        return new ExogenousBirthEmployee(this, i);
    }

    public void additionalDataRecorders() {

        if (recordModelDetail) {
            recorder.addNumericDataSource("firms_available", new
               AverageNumericDataSource(agentList) {
                public double getValue(Object argObject) {
                    return (double)((ExogenousBirthEmployee)argObject).
                        getCountOfAvailableFirms();
                }
            });
        }

        if (recordFirmCrossSection) {
```

```
            this.firmCrossSectionRecorder.addObjectDataSource("", new
                ArrayDataSource(firmsList, "Who") {
                public double getValue(Object argObject) {
                    return ((ExogenousBirthFirm)argObject).who;
                }
            });
            this.firmCrossSectionRecorder.addObjectDataSource("", new
                ArrayDataSource(firmsList, "Size") {
                public double getValue(Object argObject) {
                    return ((ExogenousBirthFirm)argObject).getSize();
                }
            });

        }

        if (recordEmployeeCrossSection) {
            this.employeeCrossSectionRecorder.addObjectDataSource("",
                new ArrayDataSource(agentList, "Who") {
                public double getValue(Object argObject) {
                    return ((ExogenousBirthEmployee)argObject).who;
                }
            });
            this.employeeCrossSectionRecorder.addObjectDataSource("",
                new ArrayDataSource(agentList,
                    "CountOfAvailableFirms") {
                public double getValue(Object argObject) {
                    return ((ExogenousBirthEmployee)argObject).
                        getCountOfAvailableFirms();
                }
            });
        }
    }
}
```

A.5.1. ExogenousBirthModel.java

A.5.2 ExogenousBirthEmployee.java

```
package ie.tcd.economics.firms;

import java.util.HashSet;
import java.util.Iterator;
import uchicago.src.sim.analysis.NumericDataSource;

public class ExogenousBirthEmployee extends Employee {
    private double randomDouble;
    private int countOfAvailableFirms;
    private double combinedSizeOfAvailableFirms;

    public ExogenousBirthEmployee(SimpleFirmsModel newModel, int i){
        super(newModel, i);
    }

    public Firm createNewFirm() {
        return new ExogenousBirthFirm(model);
    }

    // This one is too different to follow the standard model,
    // so just overwrite joinMaximumUtilityFirm.
    protected void joinMaximumUtilityFirm() {
        int option = STAY_CURRENT_FIRM_OPTION;
        Firm firmToJoin = null;

        if (!forceNewSingleton()) {

            HashSet availableFirms = new HashSet();

            CollectionSummary firmSizeSummary = new
                CollectionSummary(availableFirms) {
              public double getValue(Object collectionObject) {
```

```
            return ((Firm)collectionObject).getSize();
        }
    };

    availableFirms.add(firm);

    for (int i = 0; i < friends.length; i++) {
        availableFirms.add(getFriend(i).firm);
    }

    countOfAvailableFirms = availableFirms.size();

    double combinedSizeOfFirms = firmSizeSummary.getTotal();
    combinedSizeOfAvailableFirms = combinedSizeOfFirms;

    if (model.allowNewFirms && firm.getSize() != 1) {
        //Think about a singleton firm.
        combinedSizeOfFirms++;
    }

    double random = model.getNextDoubleFromTo(0,1);
    double accum = 0;

    for (Iterator it = availableFirms.iterator(); it.hasNext();)
    {
        Firm thisFirm = (Firm)it.next();
        accum += (double)thisFirm.getSize();
        if (accum/combinedSizeOfFirms > random) {
            if (!thisFirm.equals(firm)) {
                option = JOIN_EXISTING_FIRM_OPTION;
                firmToJoin = thisFirm;
            }
            break;
```

```
                }
            }

            if (model.allowNewFirms && firm.getSize() != 1) {
                if (accum/combinedSizeOfFirms < random) {
                    option = CREATE_NEW_FIRM_OPTION;
                }
            }

        } else {
            //Force a new singleton firm.
            if (model.allowNewFirms && firm.getSize() != 1) {
                option = CREATE_NEW_FIRM_OPTION;
            }
        }

        switch (option) {
            case CREATE_NEW_FIRM_OPTION:
                model.addSizeOfFirmLeft(firm.getSize());
                createAndJoinFirm();
                model.numberCreatingNewFirm++;
                model.addSizeOfFirmJoined(1);
                break;

            case JOIN_EXISTING_FIRM_OPTION:
                model.addSizeOfFirmLeft(firm.getSize());
                joinFirm(firmToJoin);
                model.numberJoiningFriendsFirm++;
                model.addSizeOfFirmJoined(firmToJoin.getSize());
                break;

            case STAY_CURRENT_FIRM_OPTION:
                model.addSizeOfFirmStayed(firm.getSize());
```

```
            model.numberStayingCurrentFirm++;
        }
    }

    // Accessors
    public double getCountOfAvailableFirms() {
        return countOfAvailableFirms;
    }

    public double getCombinedSizeOfAvailableFirms() {
        return combinedSizeOfAvailableFirms;
    }

    // Data Recorders
    class CountOfAvailableFirms implements NumericDataSource {
        public double execute() {
            return getCountOfAvailableFirms();
        }
    }

    class CombinedSizeOfAvailableFirms implements
        NumericDataSource {
        public double execute() {
            return getCombinedSizeOfAvailableFirms();
        }
    }
}
```

A.5.3 ExogenousBirthFirm.java

```
package ie.tcd.economics.firms;

import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.analysis.NumericDataSource;
import java.util.ArrayList;

// This subsclass of Firm isn't needed for behaviour, just to add individual firm observers.
public class ExogenousBirthFirm extends Firm {
    protected static ArrayList<Integer> firmsToWatch = new ArrayList();

    public ExogenousBirthFirm(SimpleFirmsModel newModel) {
        super(newModel);
        if (ExogenousBirthFirm.firmsToWatch.contains(new Integer(who))){
            additionalDataRecorders();
        }
    }

    public void additionalDataRecorders() {
        if (model.recordFirmTimeSeries) {
            model.firmTimeSeriesRecorder.addNumericDataSource(
                "firm_size_"+ who, new FirmSize());
        }
    }

    class FirmSize implements NumericDataSource {
        public double execute() {
            try {
                return ExogenousBirthFirm.this.getSize();
            } catch (NullPointerException ex) {
                return 0;
            }
        }
```

```
    }
}
```

A.5.3. ExogenousBirthFirm.java

A.6 Cost Curve Model

A.6.1 CostCurveModel.java

```
package ie.tcd.economics.firms;

import java.util.Iterator;
import uchicago.src.sim.engine.SimInit;
import uchicago.src.sim.util.Random;

public class CostCurveModel extends SimpleFirmsModel {

    private double gamma = 0.5;
    private double phi = 0;
    private double nu = 0;
    private double beta = 0;

    public CostCurveModel() {
        super();
    }

    public static void main(String[] args) {
        SimInit init = new SimInit();
        init.loadModel(new CostCurveModel(), null, false);
    }

    public String[] getInitParam() {
        String[] params = {"gamma", "phi", "nu", "beta"};
        return super.additionalInitParams(params);
    }

    public Employee createEmployee(int i) {
        return new CostCurveEmployee(this, i);
    }
```

```
public void buildModel() {
    super.buildModel();
    additionalDataRecorders();
}

public void step() {
    super.step();
    // Set up Firm data recorders after 1st step.
    /*if (this.getTickCount() == 1) {
        for (Iterator it = firmsList.iterator(); it.hasNext(); ) {
            CostCurveFirm firm = (CostCurveFirm)it.next();
            if (firm.getSize() > 0) {
                firm.additionalDataRecorders();
            }
        }
    }*/
}

// Accessors

public double getGamma() {
    return gamma;
}

public void setGamma(double argGamma) {
    gamma = argGamma;
}

public double getPhi() {
    return phi;
}
```

```
public void setPhi(double argPhi) {
    phi = argPhi;
}

public double getNu() {
    return nu;
}

public void setNu(double argNu) {
    nu = argNu;
}

public double getBeta() {
    return beta;
}

public void setBeta(double argBeta) {
    beta = argBeta;
}

// Data Recorders

public void additionalDataRecorders() {
    if (recordModelDetail) {
        recorder.addNumericDataSource("min_unit_cost", new
            MinNumericDataSource(firmsList) {
             public double getValue(Object argObject) {
                 return (double)((CostCurveFirm)argObject)
                     .getUnitCost();
             }
        });

        recorder.addNumericDataSource("max_unit_cost", new
```

```
                MaxNumericDataSource(firmsList) {
                 public double getValue(Object argObject) {
                     return (double)((CostCurveFirm)argObject)
                         .getUnitCost();
                 }
            });
        }
    }
}
```

A.6.2 CostCurveEmployee.java

```
package ie.tcd.economics.firms;


public class CostCurveEmployee extends Employee {


    public CostCurveEmployee(SimpleFirmsModel newModel, int i){
        super(newModel, i);
    }


    public Firm createNewFirm() {
        return new CostCurveFirm(model);
    }


    public double calculateCurrentUtility() {
        return calculateMaximumUtilityAtFirm((CostCurveFirm)firm);
    }


    public double calculateMaximumUtilityAtFirm(
        Firm firmForUtility) {
        if (firmForUtility.containsEmployee(this)) {
            return ((CostCurveFirm)firmForUtility).getPerCapitaIncome();
        } else {
            return CostCurveFirmCostFunction.calculatePerCapitaIncome(
                (CostCurveFirm)firmForUtility,
                ((CostCurveFirm)firmForUtility).getMarketSharePlusOne(),
                firmForUtility.getSize() + 1);
        }
    }
}
```

A.6.3 CostCurveFirm.java

```
package ie.tcd.economics.firms;


import java.lang.Math;
import uchicago.src.sim.analysis.DataRecorder;
import uchicago.src.sim.analysis.NumericDataSource;


public class CostCurveFirm extends Firm {


    protected double unitCost;


    public CostCurveFirm(SimpleFirmsModel newModel) {
        super(newModel);
        double gamma = ((CostCurveModel)model).getGamma();
        //Random unitCost assigned to each firm at initialization.
        unitCost = model.getNextDoubleFromTo(0.5 - gamma, 0.5 + gamma);
        additionalDataRecorders();
    }


    // Calculators
    private double calculatePerCapitaIncome() {
        return CostCurveFirmCostFunction.calculatePerCapitaIncome(this,
            calculateMarketShare(), getSize());
    }


    private double calculateMarketShare() {
        return (double)getSize() / (double)model.numberEmployees;
    }


    private double calculateMarketSharePlusOne() {
        return ((double)getSize() + 1.0) / (double)model.numberEmployees;
    }
```

```
// Accessors

public double getPerCapitaIncome() {
    return calculatePerCapitaIncome();
}

public double getMarketShare() {
    return calculateMarketShare();
}

public double getMarketSharePlusOne() {
    return calculateMarketSharePlusOne();
}

public double getUnitCost() {
    return unitCost;
}

//Data Recorders

public void additionalDataRecorders() {
    if (model.recordFirmTimeSeries) {
        model.firmTimeSeriesRecorder.addNumericDataSource(
            "market_share_"+hashCode(), new MarketShare());
        model.firmTimeSeriesRecorder.addNumericDataSource(
            "unit_cost_"+hashCode(), new UnitCost());
    }
}

class MarketShare implements NumericDataSource {
    public double execute() {
        return CostCurveFirm.this.getMarketShare();
    }
```

```
    }

    class UnitCost implements NumericDataSource {
        public double execute() {
            return CostCurveFirm.this.getUnitCost();
        }
    }
}
```

A.6.4 CostCurveFirmCostFunction.java

```
package ie.tcd.economics.firms;

public class CostCurveFirmCostFunction {



    public static double calculateUnitCost(CostCurveFirm firm,
        double marketShare) {
        double phi = ((CostCurveModel)firm.model).getPhi();
        double nu = ((CostCurveModel)firm.model).getNu();
        double beta = ((CostCurveModel)firm.model).getBeta();

        if (marketShare < 0) {
            throw new RuntimeException(
            "marketShare Cannot Be Less Than 0");
        } else if (marketShare > 1) {
            throw new RuntimeException(
            "marketShare Cannot Be Greater Than 1");
        }

        return firm.unitCost * (1 + phi * (2 * nu - marketShare) *
         marketShare * (1 - beta * marketShare));
    }

    public static double calculateTotalCost(
        CostCurveFirm firm, double marketShare, int firmSize) {
        return calculateUnitCost(firm, marketShare) * firmSize;
    }

    public static double calculateTotalIncome(
        CostCurveFirm firm, double marketShare, int firmSize) {
        return firmSize - calculateTotalCost(
            firm, marketShare, firmSize);
```

```
    }

    public static double calculatePerCapitaIncome(
        CostCurveFirm firm, double marketShare, int firmSize) {
        return calculateTotalIncome(
            firm, marketShare, firmSize) / firmSize;
    }
}
```