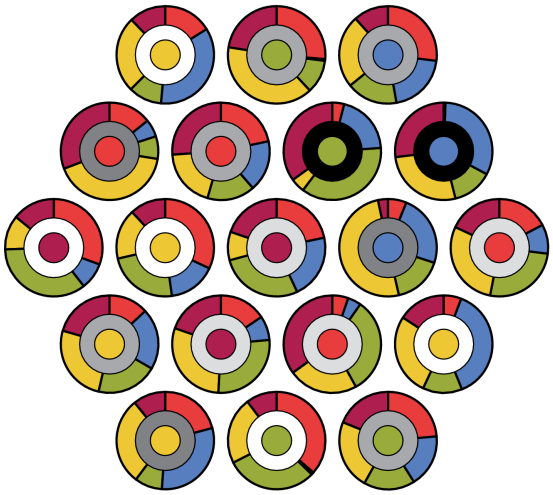
UNIVERSITY OF GOTHENBURG



An Extensible and Scalable Agent-Based Simulation of Barter Economics

Master of Science Thesis

Pelle Evensen
Mait Märdin

University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2009

An Extensible and Scalable Agent-Based Simulation of Barter Economics
Pelle Evensen & Mait Märdin

Examiner: Sibylle Schupp

Chalmers University of Technology
Department of Computer Science and Engineering
SE–412 96 Göteborg
Sweden
Telephone +46 (0)31–772 1000

Cover:
Relative private price visualisation for bartering agents. See Appendix A.2.7, page 67.

Department of Computer Science and Engineering
Göteborg, Sweden March 2009

An Extensible and Scalable Agent-Based Simulation of Barter Economics

Pelle Evensen, Mait Märdin

March 24, 2009

Abstract

This thesis project studies a simulation of decentralised bilateral exchange economics, in which prices are private information and trading decisions are left to individual agents. We set to re-engineer the model devised by Herbert Gintis and take his Delphi version as the basis for providing a new, portable barter economics simulation tool in Java. By introducing some extension points for new agent and market behaviours, we provide simple means to implement variations on the original model. In particular, our system could be used to study the emergent properties of heterogeneous agent behaviours.

Through the addition of some default visualisation models we provide means for an improved intuitive understanding of the interaction between individual agents.

The multi-agent simulation library MASON is used as the underlying simulation platform. The results of running the software with various parameters are compared to the results from the original version to confirm the convergence of the two programs.

Acknowledgements

Foremost we thank our supervisor Sibylle Schupp, for always being very kind and encouraging.

Sara Landolsi and Johan Tykesson for statistics help, Anders Moberg for some proofreading and suggestions for chapter 5.

Märt Kalmo for being our opponent. And last but definitely not least, Kaija & Hanna for putting up with us during periods of intense work.

Contents

List of Figures

Chapter 1

Introduction

Simulation is a young and rapidly growing field, useful in many different disciplines of social sciences. Economics is one of those for which simulation is very well suited—the models tend to be complex, non-linear systems that are intractable using other approaches such as mathematical modelling. In this thesis, we study the simulation of a particular economic model—*barter economy*. It is a simple economic model where agents exchange goods without money or other real-life factors such as firms, taxes, capital or material. Despite the simplicity, the dynamics of barter economies is not completely understood.

The particular model we study in this project was formulated by Herbert Gintis in [Gin06]. A proof of concept implementation in the Delphi programming language is provided from his home page.

The interesting result of Gintis' work is that in a decentralised economy, where trading agents have neither money nor prices as public information and with little central control, a system of approximately equilibrium prices emerges in the long run.

We aim to reproduce the original results and functionality by reimplementing the model in Java. We also provide means to study related models by way of extending the program by changing agent as well as market behaviour.

Chapter 2

Background

2.1 Simulation as a way of making science

One of the first uses of computers in a large-scale simulation was during World War II to model the process of nuclear detonation [Met87]. Ever since, the number of applications for computer simulation has been growing—it is now used to gain insight into the operation of natural systems in physics, chemistry, biology, economics, psychology, social sciences and possibly in various other disciplines. All of this has been made possible due to the rapid growth of computing power.

The increasing use of simulations raises an important question: what is the value of simulation as a way of making science? Robert Axelrod [Axe03] tries to answer this question by comparing simulation to the two standard methods of doing science: induction and deduction. Induction is a form of reasoning that makes generalisations based on individual instances. In social sciences, the examples of using induction could be the analysis of opinion surveys and macro-economic data. Deduction, on the other hand, is reasoning which uses deductive arguments to move from given statements (premises) to conclusions, which must be true if the premises are true. Amy Greenwald [Gre97] brings an example of deductive reasoning in classical game theory—if all the players are rational, and if they all know that they all are rational, and so on, then they all know that all the others play best responses, and as a result, they all play best responses to those best responses, which brings us to an equilibrium where no player has anything to gain by changing his or her own strategy. This is called Nash equilibrium and the discovery of this was reached by deduction [Nas50]. So, where does simulation belong? Axelrod thinks that simulation is a third way of doing science, as it combines elements from both standard methods. Like deduction, it starts with a set of explicit assumptions and then generates data that can be analysed inductively. Simulation does not prove any theorems like deduction and the simulated data does not come from direct measurement of the real world as is the case for typical induction. Rather, the data comes from a rigorously specified

set of rules. Axelrod concludes that while induction can be used to find patterns in data, and deduction can be used to find consequences of assumptions, simulation modelling can be used to aid intuition.

In 1969, Thomas Schelling used simulation to show how racial segregation happens even if individuals have a small preference for the skin colour of their neighbours [Sch69]. Schelling did not use any computers for this simulation. In his later book [Sch78], he even pointed out what is needed to replicate the results:

> "Some vivid dynamics can be generated by any reader with a half-hour to spare, a roll of pennies and a roll of dimes, a tabletop, a large sheet of paper, a spirit of scientific inquiry, or, lacking that spirit, a fondness for games."

Placing the pennies and dimes in different patterns on the "board" and then moving them one by one if they had too many neighbours of different colour were all there was to it. Despite the simplicity of the simulation, it nevertheless fulfilled the main purpose of aiding intuition—anyone could easily understand the theory by replicating the simulation. Of course, computers can simulate this much faster and we can carry out the same process hundreds of times per second, but this adds little value once we have understood the theory by doing it on paper. The paper method is possible because the model is so simple and no heavy calculations are necessary. However, if we have an economic model with several parameters and significantly more individuals acting, the only option is computer simulation. No matter whether we use a computer or not, the main value of simulation is to better understand the operation or the behaviour of a system.

2.1.1 Importance of replication

Just as important as it is to understand some phenomenon of a complex system, is sharing the insights with others. Axelrod [Axe03] brings out several difficulties that arise when sharing the results of a computer simulation. One of the main things he is concerned with is whether the shared results of a simulation are reproducible. Schelling did an excellent job in this regard, his work is easy to replicate. Unfortunately, this is often not the case for computer simulations. The models are usually sensitive to many, small details and describing them all would not fit in an article, making it hard for others to replicate or even understand the results. So, it is very important to find other means of providing the documentation and source code of the computer simulation together with the interpretation of the results.

Once all the details of a simulation are made available, it is also very important that someone tries to replicate it. According to Axelrod, this is virtually never done:

> "New simulations are produced all the time, but rarely does any one stop to replicate the results of any one else's simulation model." [Axe03]

He even calls replication one of the hallmarks of cumulative science and emphasises that it is needed to confirm whether the claimed results of a given simulation are reliable in the sense that they can be reproduced by someone starting from scratch. The basis for this suggestion is that it is easy to make programming errors, especially for those with little programming experience. This, in turn, could lead to mistaken results or misrepresentation of what was actually simulated. Or, there could be errors in analysing or reporting the results.

2.2 Agent-Based Computational Economics

At the intersection of economics and computation, lies a fairly new field called agent-based computational economics (ACE)—the computational study of economies modelled as evolving systems of autonomous interacting agents [Tes03]. The reason why economists are discovering the possibilities of agent-based modelling is that a certain class of economic problems is not solvable with mathematical models [Axt00]. The reason why these ideas have not been put into practice for centuries is that agent-based modelling was not feasible until computer hardware became powerful enough to carry out those computation-intensive simulations. But now, for these mathematically intractable problems, agents come in very handy. Tesfatsion [Tes03] gives a precise definition of an "agent" in that context—it is a bundle of data and behavioural methods, representing an entity constituting part of a computationally constructed world. For example, an agent can represent individuals (e.g., consumers, producers), social groupings (e.g., families, firms), institutions (e.g., markets, regulatory systems), biological entities (e.g., crops, livestock) or physical entities (e.g., infrastructure, weather). By creating a bunch of these simple agents and making them interact with each other, we can model a complex system. A defining property for complex systems formulated by Vicsek [Vic02] is that the laws describing the behaviour of a complex system are qualitatively different from those that govern its units. The "father" of agent-based modelling, Thomas Schelling, referred to the existence of such systems in economics in his classic paper "Models of Segregation" [Sch69]:

> "Economists are familiar with systems that lead to aggregate results that the individual neither intends nor needs to be aware of, the results sometimes having no recognisable counterpart at the level of the individual."

For example, he names the creation of money by a commercial banking system or the way that saving decisions cause depressions or inflation. Vicsek summarises the benefits of agent-based approach to complex systems:

> "By directly modelling a system made of many units, one can observe, manipulate and understand the behaviour of the whole system much

> better than before. In this sense, a computer is a tool that improves not our sight (as does the microscope or telescope), but rather our insight into mechanisms within complex systems." [Vic02]

ACE research has four objectives [Tes06]. Firstly, to empirically understand why particular global regularities have evolved and persisted, despite the absence of centralised planning and control. Those global regularities are the large-scale effects of complex systems, indirect results of interacting individuals. Thus, the aim is to generate those global regularities within agent-based worlds. Epstein [Eps06] calls this a "generative" approach to science, the main question being how decentralised local interactions of heterogeneous autonomous agents generate the given regularity.

The second objective is normative understanding. Here the ultimate question is how agent-based models can be used as laboratories for the discovery of good economic designs. Again, an agent-based world is constructed, but this time the aim is to assess how efficient, fair and orderly the outcomes of a specific economic design are.

The third objective is qualitative insight and theory generation, with the main question of how economic systems can be more fully understood through a systematic examination of their potential dynamical behaviours under different initial conditions. Tesfatsion [Tes06] reasons that such understanding would help to clarify not only why certain outcomes have regularly been observed but also why others have not.

Finally, the objective also pursued by this thesis project, is methodological advancement. Researchers with this objective in mind try to find the best methods and tools to undertake the rigorous study of economic systems. Tesfatsion lists the needs of ACE researchers that the methodology should meet:

- Model structural, institutional, and behavioural characteristics of economic systems.
- Formulate interesting theoretical propositions about their models.
- Evaluate the logical validity of these propositions by means of carefully crafted experimental designs.
- Condense and report information from their experiments in a clear and compelling manner.
- Test their experimentally-generated theories against real-world data.

In this thesis, we are more interested in the advancement of practical tools of programming, visualisation and validation than in the advancement of methodological principles.

2.2.1 Advantages of agent-based approach

To further see the benefits of ACE, Robert Axtell brings out the advantages of agent-based computational modelling over conventional mathematical theorising [Axt00].

Firstly, in agent-based computational models it is easy to limit rationality of the agents. The agents do not need to act rationally, one can experiment with different kinds of agents and just see what happens. In contrast, mathematical models assume rational agents, just like physicists have the ideal gas and the perfect fluid. Secondly, it is easy to make agents heterogeneous in agent-based models. It is a matter of instantiating the agent population with different initial states or behaviour strategies. Thirdly, Axtell points out that by "solving" the model merely by executing it, we are left with an entire dynamical history of the process under study. Therefore, it is possible to not only concentrate on the possible equilibria of the model, but one can also study the dynamics of the process. As a final advantage, he argues that in most social processes either physical space or social networks matter, which are difficult to account for mathematically. In agent-based models however, it is quite easy to have the agent interactions mediated by space or networks or both.

Together with all these advantages, Axtell points out a significant disadvantage that agent-based modelling methodology has when compared to mathematical modelling. Namely, a single run of an agent model does not give us any information about the robustness of the results. He raises a more formal question:

> "Given that agent model A yields result R, how much change in A is necessary in order for R to no longer obtain?" [Axt00]

The truth is that we can only answer this by running the model with systematically varied initial conditions or parameters and then assess the robustness of results. This of course limits the size of the parameter space that we can check for robustness in a reasonable time. In mathematical economics, the question of parameter robustness is often formally resolvable.

2.2.2 Construction of agent-based models

Tesfatsion [Tes06] compares the ACE methodology to a culture-dish approach in biology—an ACE modeller first computationally constructs an economic world, populates it with multiple interacting agents and then steps back and just observes the development of that small world over time. The most important part of constructing such a world is specifying the agent. An agent typically has data attributes (e.g., type of agent, info about other agents, utility function) and behavioural methods (e.g., market protocol, private pricing strategy or learning algorithm). That is the groundwork of agent-based models, what is left is the glue code to make the agents

interact in some regular manner and at the same time advance the model in time. The effort in this part can be greatly reduced by using some existing multi-agent system framework. There are many choices for this, and in the next section we will take a closer look at one of them. Eventually, every ACE model should have the property of being dynamically complete, meaning that it must be able to develop over time solely on the basis of agent interactions, without further interventions from the modeller [Tes06].

2.3 MASON multi-agent simulation toolkit

Most of the agent-based models need some boilerplate code that is common among all of them. Examples include a good random number generator, synchronisation of agent interactions, visualisation and a graphical user interface for controlling the simulation. This is a fair amount of work if starting from scratch, but in most cases unnecessary because a large variety of frameworks (or toolkits or platforms) exist for multi-agent systems, offering the described basic functionality and often more. One of those frameworks, used in this project, is MASON[1]—Multi-Agent Simulator Of Neighbourhoods (or Networks). This free and open-source general purpose simulation toolkit is a joint effort of George Mason University's Computer Science Department and the George Mason University Center for Social Complexity. Analysing the pros and cons of every existing framework would be enough work for another thesis project, but the important criteria talking in favour of MASON are the Java programming language (making it multi-platform), high performance and thorough documentation.

A not so recent review [RLJ05] of agent-based simulation platforms found MASON to be the fastest among four other popular platforms—Swarm and Java Swarm[2], Repast[3] and NetLogo[4]. To date, over 50 platforms[5] with similar goals exist, which means that most of them remain undiscovered for this project.

2.3.1 The architecture of MASON

The motivation behind starting the MASON project in the first place was the need for a general purpose simulation toolkit that would not be tied to any specific domain [LCRPS04]. Other critical needs were speed, the ability to migrate a simulation run from platform to platform and therefore also platform independence. The authors of MASON argue that at the time when MASON development started, the existing systems did not meet these needs well. They either tied the model to the GUI too

[1] `http://cs.gmu.edu/~eclab/projects/mason/`
[2] `http://www.swarm.org`
[3] `http://repast.sourceforge.net`
[4] `http://ccl.northwestern.edu/netlogo/`
[5] `http://en.wikipedia.org/wiki/ABM_Software_Comparison`

closely, or could not guarantee platform-independent results, or were slow because they were written in an interpreted language. The architectural choices of MASON are to fix these problems.

MASON is written in Java in order to take advantage of its portability and strict semantics for libraries and math operations to guarantee duplicable results. Another useful feature is object serialisation, which enables saving a simulation state to disk and restoring it later. From the architectural viewpoint, the toolkit is modular and layered. The bottom layer consists of utility data structures, such as custom implementations of `Bag` and `Heap`. Next comes the model layer. This is the core of MASON and has all the functionality to create simulations with command line interfaces. This layer includes a single base class for the simulated model (`sim.engine.SimState`), backed by functionality for scheduling agents and also a high-quality pseudo-random number generator (see 6.2). MASON employs a specific usage of the term agent—it is a computational entity which may be scheduled to perform some action and possibly manipulate the environment [LCRPS04]. So, the agents are scheduled to take action, or "step forward" in MASON's terms, and the only requirement for an agent is to implement a `Steppable` interface with a single method that is called by MASON according to the schedule.

The model layer is completely independent of the visualisation layer. Nevertheless, attaching visualisations and a GUI for simulation control is straightforward. For these purposes, another base class called `GUIState` is provided and very little knowledge of the Java Swing GUI framework is required. The serialisation of the model (`SimState`) to or from disk also happens through this class. For the visualisation purposes, there are several classes to support drawing various 2D and 3D representations of the model.

From a programmer's perspective, MASON is low level and requires Java programming skills to be able to construct one's own agent-based model. The only design concept enforced by MASON is its view of an agent as something that steps forward in a series of discrete events. But there is a positive side of its generality—the domains for which MASON is suitable ranges from robotics, machine learning and artificial intelligence to multi-agent models of social systems [LCRPS04].

2.4 Gintis' Barter Economy

One of the important questions in economics has for a long time been how to match the demand and supply of all goods in a market of perfect competition, so that there is neither excess demand nor supply. Or from another point of view, how to find the market-clearing prices that would result in this match. The study of this problem has its own branch in theoretical economics called *General Equilibrium Theory*, but the first man to address these issues was the French economist Leon Walras (1834–

1910). He proposed a dynamic process called *tâtonnement* (or groping) by which general equilibria might be reached.

The central part of this process is an (Walrasian) auctioneer who calls out prices of goods after which agents register how much of each good they would like to offer (supply) or purchase (demand) at the given price. No transactions or production take place at disequilibrium prices. Instead, prices are lowered for goods with excess supply and raised for goods with excess demand. Eventually, this process will give rise to general equilibria.

Herbert Gintis demonstrates a different, agent-based approach [Gin06]. He constructs a simple economic model called *barter economy*, where agents just produce, exchange, and consume a number of goods in many consecutive periods. No other realistic factors such as money, firms, capital, or material are included. The major difference from Walras' model is that the described barter economy is completely decentralised—no top-down control exists such as the Walrasian auctioneer with the perfect information. Gintis sums up his approach:

> "Rather than using analytically tractable but empirically implausible adjustment mechanisms and informational assumptions (such as Walrasian tâtonnement and prices as public information), we treat the economy as complex system in which agents have extremely limited information, there is no aggregate price-adjustment mechanism institution, and out of equilibrium exchange occurs in every period." [Gin06]

So, Gintis sets to extend the empirical understanding of the dynamics of barter economy that would lead to an equilibrium.

2.4.1 Overview of the process

The following process devised by Gintis is carried out in each period to evolve the economy over time. It begins with a synchronised *production* phase—each agent starts with an empty inventory of goods and then produces a fixed amount of a single good. The production phase is followed by an unsynchronised *exchange-consumption-production* phase. Here the agents first seek exchange partners and then try to agree on the amounts of exchanged goods according to their strategies. A strategy for an agent is a price vector for the various goods it produces or consumes. Agents only give away a quantity of their own production good and only if the value of what they receive in exchange is at least as great as the value of what they give away, according to their private price vector. After a successful trade, an agent consumes an optimal consumption bundle and produces more of his production good if his inventory becomes empty after the consumption.

The final phase is *reproduction-mutation*, which only happens after a certain number of periods (for example every 10th period). In this phase, a fraction of

low-scoring agents imitate the strategies (the private prices) of high-scoring agents and with a small probability, each strategy undergoes a small mutation. This phase can also be seen as the learning phase.

Repeating this process for a long enough time (on the order of 150 000 periods), Gintis showed that the prices converge approximately to the market-clearing values and thus an equilibrium is reached.

Chapter 3

Analysis of the implementations

In this chapter, we look at two different implementations of the barter economy model. First, we give an overview of Gintis' original implementation constructed from scratch in the Delphi programming language. Then, we study an alternative implementation of the same model built in Java, which we have implemented ourselves using MASON.

3.1 Original implementation

The original implementation can be viewed as a proof of concept. It is written in Delphi, a further development of Object Pascal, which enables rapid construction of GUI applications on Windows platform. Although Delphi has object-oriented language features such as encapsulation, polymorphism and inheritance, this implementation does not take advantage of those.

The core of the program is fitted into a single source file with nearly 1000 lines of code. Most importantly, a clear distinction of what constitutes an agent has been made. Fig. 3.1 shows the definition of the Agent class devised by Gintis. Everything in this class has public access, even the private price vector. Thus, the idea of missing public information is not directly projected to the code, as any agent has access to the prices of any other agent. But this is a design issue and good care has been taken to ensure that no agent reads the prices of another agent unless they really need to—that is when they are scoring low and need to imitate someone else's strategy. One could argue that avoiding encapsulation like this would ease the programming effort as everything is at hand when needed. But, at the same time, the lack of encapsulation increases the coupling between different parts of the program and results in hard to follow "spaghetti" code.

Another thing to notice is the four similar methods of `Trade` vs. `CommonPriceTrade`, `Eat` vs. `CommonPriceEat`, `Lambda` vs. `CommonPriceLambda` and `SetDemandAndSupply` vs. `SetCommonPriceDemandAndSupply`. In fact, there is very little dif-

```
Agent = class(TObject)
   Index, Produces : Integer;
   Price, Inventory, Buy, ExchangeFor : Array of Double;
   ProduceAmount, Score : Double;
   Constructor Init(Produces, Index : Integer);
   Procedure Copy(AgentIdx : Integer);
   Procedure CopyAndMutate(AgentIdx : Integer);
   Procedure Eat;
   Procedure CommonPriceEat(EPrices : Array of Double);
   Function Trade(B : Agent) : Boolean;
   Function CommonPriceTrade(B : Agent; EPrices : Array of Double) : Boolean;
   Function Lambda : Real;
   Function CommonPriceLambda(EPrices : Array of Double) : Real;
   Procedure SetDemandAndSupply;
   Procedure SetCommonPriceDemandAndSupply(EPrices : Array of Double);
end;
```

Figure 3.1: Class definition for Agent in the original implementation of the barter economy

ference between the two variants and CCFinder[1], a token based clone detector, suggests that they are all duplicates. In the worst case, when it comes to comparing `Trade` to `CommonPriceTrade`, the size of an exact clone is over 50 lines long. The only difference between a standard method and a `CommonPrice*` variant is that the latter operates on a price vector passed as a parameter rather than on agent's own private prices.

To sum up, the code has not been written with replication in mind. Lack of comments makes it even more difficult to extract the important details about the model. Nevertheless, the implementation serves its purpose and confirms the results presented in [Gin06].

3.1.1 Deviations from the paper

This section illustrates the need for reproducing simulation models—to detect programming errors that might have affected the drawn conclusions. The original implementation has several deviations from what was described in the paper, or features that obviously have not been intentional. Most of them have no effect on the price convergence property of the model, but we will nevertheless look at them to provide fixes in the re-implementation.

Firstly, a rather significant bug is introduced when calculating the demand vector for an agent. Every agent wants to consume n goods in fixed proportions $(o_1, \ldots, o_n)$. If $(x_1, \ldots, x_n)$ is the inventory of an agent, then the utility function is defined as in Eq. 3.1.

$$u(x_1, \ldots, x_n) = \min_{0<j\leq n} \frac{x_j}{o_j} \tag{3.1}$$

[1] `http://www.ccfinder.net/`

It is not wise for agents to consume the whole amount of one particular good and only a small fraction of some other good. To maximise the score, agents should try to acquire equal proportions of all n goods. Gintis calculates the optimum inventory (or demand) according to Eq. 3.2, where x^*_{ij} denotes the optimum inventory for good j of agent i and λ^* is the proportion that would result in the highest utility.

$$x^*_{ij} = \lambda^* o_j \tag{3.2}$$

$$\lambda^* = \frac{\sum_j p_{ij} x_{ij}}{\sum_j p_{ij} o_j} \tag{3.3}$$

The calculation of λ^* is shown in Eq. 3.3. The numerator of this fraction is the income constraint for an agent—the value of all goods in the inventory according to an agent's private prices. The denominator is the value that an agent ultimately wants to consume, according to the private prices again. This is how the λ^* calculation is presented in the paper and what seems reasonable. In Gintis' implementation however, λ^* is calculated as in Eq. 3.4.

$$\lambda^* = \frac{\sum_j p_{ij} x_{ij}}{\sum_j p_{ij} x_{ij}} = 1 \tag{3.4}$$

So the question is what are the consequences to the results. Could this discrepancy lead to different convergence behaviour? All agents will on average score lower because they waste everything they produce to buy a single or a few other goods instead of getting a little bit of everything and thus a better score. The global effect on the economy is shown in plots (e) and (f) of Fig. 3.2. $\mathcal{J}_l$ denotes our Java implementation with the λ^* calculation bug fixed and $\mathcal{G}_k$ is the original implementation. The plot (e) compares the means of the average relative prices[2] in a 3-good[3] economy. We can see how different the means are between $\mathcal{J}_l$ and $\mathcal{G}_k$ versions by comparing them to the means in plot (a), where the Java version $\mathcal{J}$ has all the same bugs as in $\mathcal{G}_k$.

The same goes for the plots (f) and (b), except that they show the variance of the average relative prices between different runs of the simulation.

[2] An average relative price for a good shows how much the price of the good differs from the equilibrium price on the average (among all the agents). So, a value of -0.3 at some point means that the average price is 30% lower from the equilibrium price at that point. The mean of those prices just indicates that we have several runs of the same simulation.

[3] The price of one particular good is taken as the price unit for all the other goods. Thus, one good always has a constant equilibrium price in the charts and is not shown.

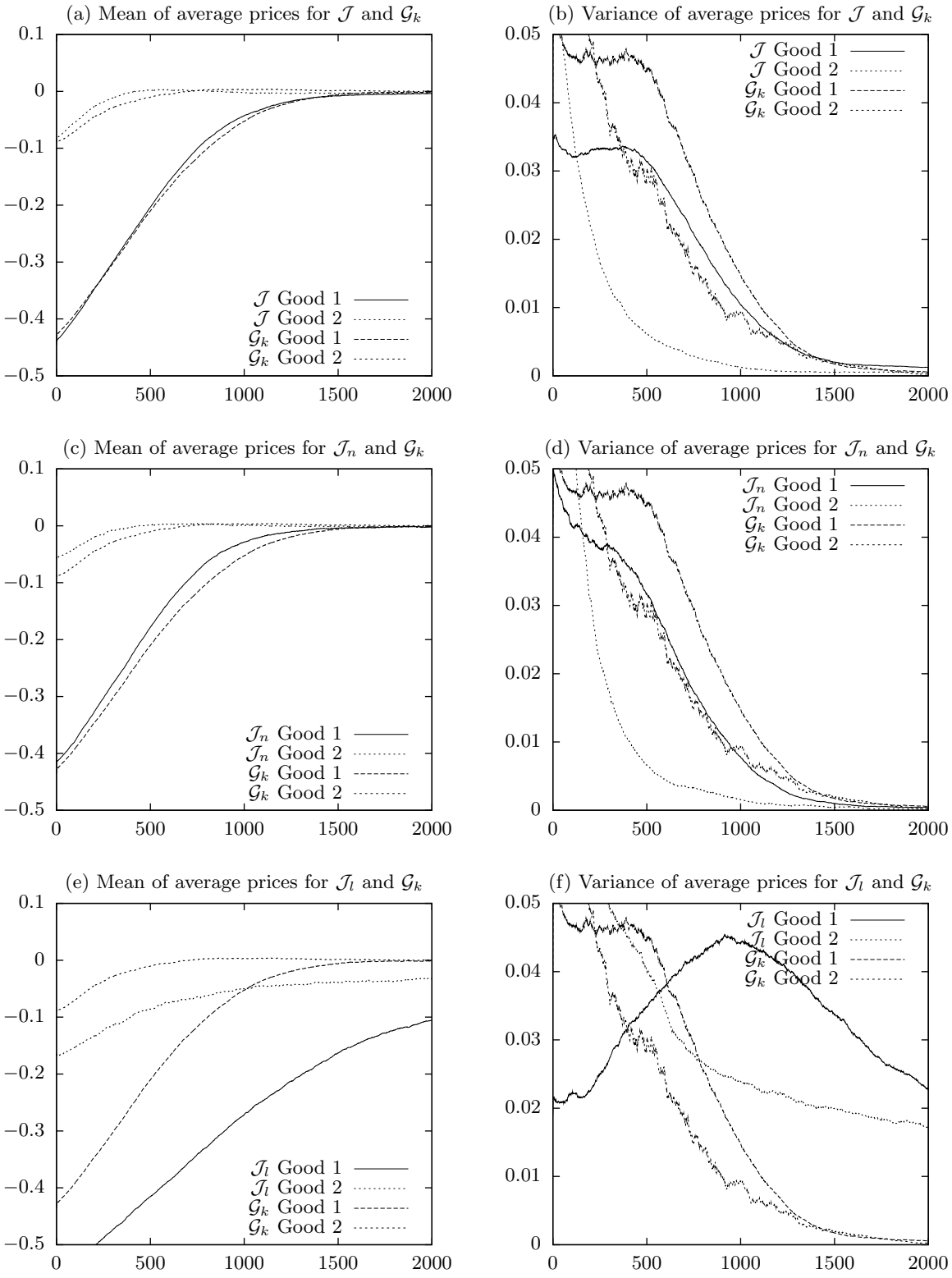


Figure 3.2: Our implementations $\mathcal{J}$, $\mathcal{J}_l$ & $\mathcal{J}_n$ with different bugs fixed compared to the original implementation $\mathcal{G}_k$.

Another discrepancy from the paper is how agents calculate the amount of their produce-good that they are willing to exchange for other goods. Gintis calls the trade initiating agent an "offerer" and defines the following procedure to determine trade amounts:

> "When Offerer i producing good g encounters agent j producing $h \neq g$, he uses Eq. 3.2 and Eq. 3.3 to determine an amount $x_{ih} > 0$ of good h he will accept in exchange for an amount $x_{ig} \equiv p_{ig}x_{ig}/p_{ih}$ of his production good g."

Obviously the indices of $x_{ig} \equiv p_{ig}x_{ig}/p_{ih}$ have gone wrong here—from agent i's point of view, the amount of good g it gives for good h is determined by finding the value of what it would receive, i.e., $p_{ih}x_{ih}$, divided by the price of its production good p_{ig}. So, the correct equation would be $x_{ig} \equiv p_{ih}x_{ih}/p_{ig}$ and this is also how the implementation looks like. But even though, this invariant is not preserved throughout the simulation. At the start of every period, x_{ig} is calculated correctly as described above for every possible good $h \neq g$. After successful trading however, Gintis makes a shortcut and adjusts the amounts for both agents (i and j) as shown in Eq. 3.5.

$$x_g \leftarrow x_g - givenAmount \tag{3.5}$$

It proves to be correct for the offerer (agent i), because it gets to choose the trade conditions and the amount it gives away will always reflect its prices. But for the responder (agent j), this method gives wrong amounts because it accepts a trade if $p_{jg}x_{ig} \geq p_{jh}x_{ih}$; that is, when it values what it receives in trade at least as much as what it gives up. In case the received value is strictly greater, the adjustment of x_{jg} by Eq. 3.5 will go wrong—agent j gives up less than it is willing to give up and thus, next time it buys the same good, it gives up more than its private prices would allow. The correct way would be to recalculate x_{ig}, every time x_{ih} changes, from $x_{ig} \equiv p_{ih}x_{ih}/p_{ig}$.

The third bug that affects the simulation flow is from the reversed order of two important events. After each trading period, Gintis first resets the demand (x_h) and supply (x_g) for all agents. Then, if it happens to be a reproduce period, lower scoring agents get a chance to copy, or "imitate", the prices of better scoring agents. After each such period, a bunch of agents who just got new price vectors, will perform trades according to the old price vector, because the demand and supply will not be recalculated until the next period. This also means trades with negative profit, as did the previously described bug. The global effects of those two are not as significant as with the bad λ^* calculation. The plots (c) and (d) of Fig. 3.2 illustrate the differences. $\mathcal{J}_n$ denotes our Java implementation with the the two bugs fixed and $\mathcal{G}_k$ is the original implementation. We can see that the means in (c) are not

very different from the means in (a), where the $\mathcal{J}$ version has all the bugs. The same goes for the variances in plots (d) and (b).

Then there are a couple of minor bugs that have little or no effect on simulation process. First, if agents are allowed to shift from producing one good to another, giving the parameter $producerShiftRate$ a value 0.01 does not mean that 1% of all agents are going to shift their production good, but rather $0.01 \times totalAgents \times numGoods$ agents will shift[4]. It is hard to tell whether this behaviour was intentional. Other bugs include a slight miscalculation of the standard deviation for consumer and producer prices[5] of a particular good; and crashing the program at runtime when dividing the agents unequally between different production goods[6]. The latter sometimes causes a call for a negative random number which in turn raises an exception in the Delphi `Random` function.

We discuss one more conceptual difference between the paper and the implementation in 4.3.3, where it is more natural to explain.

3.2 Generalisation of the barter economy

The goal of agent-based modelling is not to provide as accurate representation of some real world system as possible, but rather to enrich our understanding of them. Creating an all-in-one general model does not take us closer to that goal, as it becomes harder to grasp "what is causing what" if the parameter space grows too large. Axelrod [Axe03] calls for adhering to the KISS principle, which stands for the army slogan "keep it simple, stupid." He explains:

> "The KISS principle is vital because of the character of the research community. Both the researcher and the audience have limited cognitive ability. When a surprising result occurs, it is very helpful to be confident that one can understand everything that went into the model."

Does generalising the barter economy model mean abandoning the KISS principle? If we think of generalisation as of adding other realistic factors to the same model, then this definitely is a trade-off for simplicity. Such realistic factors could be the agents consuming and producing an unlimited number of goods, or employing a money good which is not consumed but only used in transactions, or introducing other types of agents like firms that hire regular agents to produce goods. But as Axelrod puts it, the complexity of agent-based modelling should be in the simulated results, not in the assumptions of the model.

Another approach to generalisation is from the methodological point of view—could we provide a general enough toolkit that allows modelling the barter economy

[4] In the `ProducerShift` procedure.
[5] The `CalculateConsumerPriceStdDev` and `CalculateProducerPriceStdDev` functions.
[6] At the start of the main function `Button1Click`.

as it is, plus other roughly similar models from the same domain? Building such a domain specific layer on top of MASON would require input from people with economic background and still there would be a question of what is the common part of all such economic models, if anything at all. It is not unlikely that the best abstraction is close to what MASON provides. To illustrate the point, we study another model devised by Gintis, called *GenEqui*.

3.2.1 The GenEqui project

The GenEqui project is a follow-up of the work on the barter economy. In the corresponding paper [Gin07], Gintis studies the same kind of issues as in [Gin06]. The aim is to once again extend the empirical understanding of the dynamical properties of the Walrasian general equilibrium model and to find an alternative to the tâtonnement process.

Although the goals of the two models are the same, the underlying agent-based models differ substantially. GenEqui introduces firms and a Monetary Authority who creates money by giving out loans to firms and by paying unemployment insurance. The simple agents are called workers and they look for jobs in firms to get paid, for the earned money they will buy goods produced by those firms. The core properties of the model are just like in the barter model—the private prices of agents and also the private demand and supply conditions for the firms.

In a sense, the GenEqui model is as close to the barter model as possible. They both evolve towards market clearing prices by minimising the public information and using imitation as the learning mechanism. One would expect that they have much in common and a good abstraction can be made for both of them. But when it comes to the implementation, there is not much sharing between them. The agents in the barter economy have almost nothing in common with the workers in GenEqui, or even less with the firms. We used CCFinder again to see if there is any copy-paste code between the two projects, but no; Gintis found that it is better to start from scratch than to build on top of the barter economy.

The barter economy model was not easy to reproduce, but the complexity of the GenEqui model is bigger by an order of magnitude, as is the number of details hidden in uncommented code. All that leaves the GenEqui project out of scope for this thesis project.

3.3 New implementation

At this point, we have developed some rough guidelines for the new implementation. Firstly, it should not be much more general than the original barter economy but rather more flexible—the user should be able to extend it without having to study the whole source code. Secondly, to give some guarantees on how the program

works, it is important to take encapsulation at its highest and make it difficult for the user to accidentally change the behaviour of the model in undesired ways and thus misinterpret the results. Thirdly, MASON will be used as the underlying simulation platform; and last but not least, we set to fix the mismatches between the code and the description of the model and take Gintis' paper as the reference rather than the implementation.

3.3.1 Adapting to MASON's architecture

The architecture of the new implementation is, to a large extent, driven by MASON's architecture—a schedule of agents and a clear distinction between the simulated model and the GUI. The central part of every model is the simulation state, a class inherited from MASON's `SimState` (see Fig. 3.3). This is where the model is initially set up by creating and scheduling the agents. To be precise, anything that implements the `Steppable` interface can be scheduled. The GUI layer that attaches itself to the simulation state is optional, `SimState` and also our `BarterEconomy` are unaware of it.

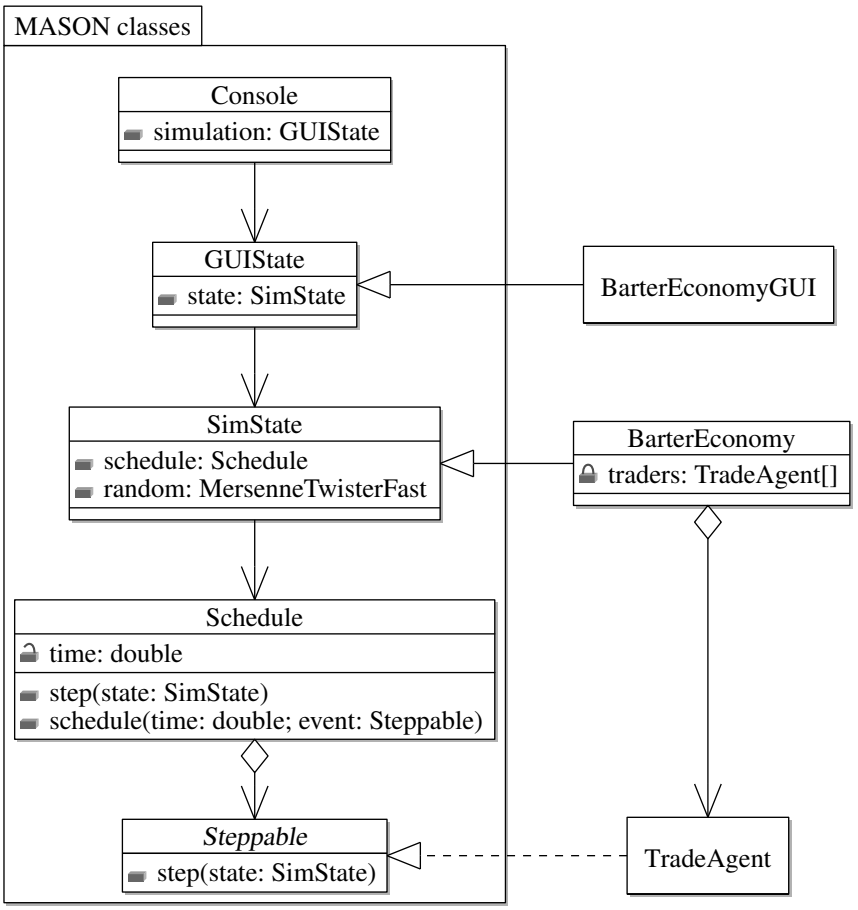


Figure 3.3: The class structure for the new implementation using MASON

Once the model is set up, it is advanced by calling the `step` method in `Schedule`. This method increases an internal time ticker of the schedule and invokes the individual `Steppable`'s that were scheduled for this particular timestamp. If a GUI is used to start the simulation, then it is `GUIState` that takes care of these top level calls to `Schedule`, otherwise it is up to the user to write a loop for this purpose.

The described architecture matches well the *Model-View-Controller* (MVC) pattern. `SimState` and its supporting classes represent the model part, whereas `GUIState` together with various charting and visualisation functionality constitutes the view part. MASON's `Console` class defines the way the user interface reacts to user input and thus acts as a controller. It contains all the functionality for simulation control (starting, stopping, pausing) and also an interface to modify the model parameters.

3.3.2 Ensuring local correctness

Before comparing the global behaviour of our implementation to that of the reference implementation, it is helpful to be sure that our program really does what we think it does. If we can confirm our assumptions about the low-level behaviour of the program and still get a different global behaviour, then the assumptions themselves need to be revised.

The focus is on the `TradeAgent` class, as it defines and mutates the state of the simulation model. To ensure its correctness, we establish some class invariants as well as pre- and postconditions for the key methods. We implement these checks by using the `assert` statement in Java. Another consideration was *The Java Modelling Language*[7], which uses Java annotation comments for specifying various checks, but unfortunately the supporting tools do not handle Java versions above 1.4 yet.

Figure 3.4: Instance variables in the `TradeAgent` class

The state of an agent is defined by the instance variables shown in Fig. 3.4. The `produceGood` field defines the good that a particular agent produces and `produce-`

[7] `http://www.eecs.ucf.edu/~leavens/JML/`

`Amount` is the amount it can produce at a time. The type of `produceGood` is `int`, which means that a good is represented just by an integer. Thus, the private prices of an agent can be expressed as an array of doubles, where the array index also denotes the good number. The same mapping between goods and array indices is used for all the fields with `double[]` as the type. From this we know that the length of every such array always equals to the number of goods in the model. We can define it as a class invariant for the `TradeAgent` class.

However, breaking the described invariant would be a major bug and most likely would crash the program. It is more desirable to detect errors that silently affect the behaviour of the model. One such invariant that is broken in the original implementation was described in 3.1.1. The amounts that an agent is willing to give in exchange for any other good are pre-calculated and stored in the `exchangeFor` array. As these amounts always reflect the price vector of the agent, the following assertion should always hold for every good `g`:

```
assert Math.abs(exchangeFor[g] − price[g] * demand[g] / price[produceGood]) < 0.01;
```

Figure 3.5: Checking a class invariant with the `assert` statement

Another invariant is that the `score` of an agent can not be negative. The same also holds for the values in `consume[]` (how much of each good an agent consumes), `price[]`, `demand[]`, `exchangeFor[]` and `inventory[]`.

All these invariants are established in the constructor of the `TradeAgent` class and if any of the assertions on these invariants fails later on, there must be a programming error.

Additionally, we use assertions to ensure that no resources are lost or created during a trade as shown in Fig. 3.6; or to check that the trade conditions have not been changed after both sides have adjusted the exchanged amounts to be compatible with their inventory (the ratio of the amounts has to be the same).

```
Double before = null, after = null;
assert (before = responder.getInventory(produceGood) + inventory[produceGood]) != null;
... // trading between this and responder
assert (after = responder.getInventory(produceGood) + inventory[produceGood]) != null;
assert Math.abs(before − after) < 0.02;
```

Figure 3.6: Comparing the total amount of a traded good before and after the trade

By default, the assertions in Java are not enabled and thus have no performance penalty at run-time.

3.3.3 Serialisation

MASON supports *checkpointing*, that is, saving the simulation state to disk and restoring it later on. This is particularly useful when the simulation runs take a long time—one might want to fork the simulation at some point and continue with different parameters for different branches from there on.

The checkpointing is built on the Java *object serialisation* API and concerns only the model layer. Writing the simulation state to disk starts from the `BarterEconomy` class, and every object referenced from there on must be made serialisable by implementing the `java.io.Serializable` interface somewhere in its class hierarchy.

In fact, there is more to serialisation than just implementing the named interface. One issue that Bloch [Blo08] points out, is that when a class implements `Serializable`, its byte-stream encoding (or serialised form) becomes part of its exported API. To make the serialised form of the simulation concise and compatible with the newer versions of the classes, we want to serialise the minimum number of fields that lets us restore the global state of the simulation. For example in the `TradeAgent` class, we can avoid serialising the `myProxy` field of type `TradeAgentProxy` (by using the `transient` keyword). It is a restricted view on the `TradeAgent` class (see 4.2), which is easy to reconstruct after reading the rest of the object from a byte stream as shown in Fig. 3.7.

```
private transient TradeAgentProxy myProxy; // No need to serialise this

private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        myProxy = new TradeAgentProxy(this);

        if (!checkInvariants()) {
                throw new InvalidObjectException("TradeAgent invariants broken!");
        }
}
```

Figure 3.7: Restoring an instance of `TradeAgent` class from its serialised form

We must also permit that the `readObject` method, which constructs the object from a byte stream, becomes another public constructor for the class. Thus, we need to ensure that the invariants of the class still hold after reading an object from a byte stream (Fig. 3.7).

When it comes to serialising the user provided classes (the strategy classes, see 4.3), we decided to extend our interfaces with the `Serializable` interface rather than letting the individual classes decide on implementing it. This guarantees the default serialisation protocol for all the user-provided classes and the user does not need to know about the serialisation framework.

3.3.4 Thread safety

Not much effort has been put into making the classes thread-safe as the simulation process is sequential. The only exception is the `BarterParams` class, which has to handle concurrent reads and writes from the main thread and the Swing GUI thread. The class is a container for all the model parameters and provides just the accessor methods.

An easy way to synchronise the class is to make all the methods `synchronized` and thus use the intrinsic lock of the `BarterParams` class. We can avoid acquiring the lock when reading/writing `int` and `boolean` parameters as those operations are atomic.

A more fine-grained (non)solution would have been to introduce a separate lock for every parameter field, so that different parameters could be read and written concurrently. But at some point we need to clone the whole object and thus also need all the locks. Acquiring the locks one by one is a possible source for deadlocks.

Chapter 4

Extensibility

4.1 Possible uses for the application

We are speculating that this application could be used by people who are studying or are conducting research in economics rather than in computer science. As such, we want to provide for easy and *safe* ways to extend the application by way of simple, yet flexible interfaces.

Generalising over different market models could not be done in a way that would allow for orthogonal extensions—at least not for those extensions we had in mind. The more general the market model, the less behaviour is pre-defined and the more is left to custom extensions. Those extensions then become dependent on each other. For example, it is hard to describe the agent behaviour for accepting trades if we do not even know what constitutes an agent and a trade. Depending on the other extensions, it could be a work contract between a worker and a firm, but also the bartering of two goods as in the barter economy.

This leads us to thinking of the "market rules" and "trader behaviour" just within the barter economy. A user should be able to change the behaviour of the agents, either for all or for some fraction so that agents with different behaviours could cooperate in the same simulation. Gintis' original formulation thus becomes a special case that is implemented as the default behaviour.

4.2 Safety/correctness aspects of extensions

If we regard the whole simulation as a game and look at each user of the system as a player, how can we ensure that no player can cheat? Cheating in this context would mean that a player (for its agents) can gain information it is not supposed to have or that it can affect any agents by other means than passing data back to the class that called the strategy.

The market itself may need access to the agents that any strategy used in the

agents should not have. We solve this problem by introducing a *protective proxy* [GHJV95] for the `TradeAgent`. The purpose of the proxy is to provide a more restrictive interface to the `TradeAgent` class. In particular, no mutators are accessible and no references to the fields in the agent can be accessed. In this case, the proxy is not expected to ever revoke the permissions of the accessing class. If the `TradeAgent` class is extended, the proxy does *not* change automatically; thus the strategies can not, accidentally or intentionally, break any rules that were in place before the change. This could maybe be regarded as a special case of the *Facet pattern*, where we have a fixed facet accessible for the agent strategies and one for the market.

4.3 Extending agents on the individual level

The task of controlling that new behaviours are orthogonal to each other is simplified by not letting the `TradeAgent` class be sub-classed. By explicitly prohibiting inheritance we can tightly control what extensions are allowed.

There could be two different goals when inheriting from some class;

> "One can view inheritance as a private decision of the designer to "reuse" code because it is useful to do so; it should be possible to easily change such a decision. Alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child class is merely specialising or refining the parent class." [Sny86]

The first goal is clearly not appropriate here; we do not expect the agent class to be useful outside this project. The second goal could be relevant but if the class is extended by inheritance we run into the "fragile base class problem" [MS97]. In our application the problem would manifest itself as follows;

- If methods are overridden, there can be no guarantee that the child still acts according to the market rules. To be able to guarantee that the agent plays by the rules, it should no longer have [write] access to its own data. Another solution to ensure that the agent abide by the market rules would be that all transactions would have to be verified by some signature or checksum that easily can be verified but not "forged" by the agent. This would be outside the scope of this project and also slow down the simulation. Even if the child is originally well behaved, later additions to the base class can render it unsafe.

4.3.1 Barter strategies

Axelrod discusses some aspects of adaptive vs. rational/optimising strategies in [Axe03]. The trading behaviour in Gintis' original model could be described as

purely adaptive; the agents do not take history into account, nor do they try to maximise their score by actively adjusting their price levels. By providing an interface that lets the agent have access to slightly more information compared to the original program, a new set of adaptive *and* rational behaviours could be created. In particular, agents having different behaviours could co-exist in the same market. The barter strategy interface and an implementation of Gintis' original barter rule can be found in Fig. 4.1.

```
public interface BarterStrategy {
    public boolean acceptOffer(TradeAgentProxy me, int offersGood,
                               double offersAmount, double requestsAmount);
}

public class OriginalBarterStrategy implements BarterStrategy {
    @Override
    public boolean acceptOffer(TradeAgentProxy myself, int offersGood,
                               double offersAmount, double requestsAmount) {
        return !(myself.getDemand(offeredGood) == 0 ||
                 myself.getExchangeFor(offeredGood) == 0 ||
                 myself.getInventory(myself.getProduceGood()) == 0 ||
                 myself.getPrice(offeredGood) * offeredAmount <
                 myself.getPrice(myself.getProduceGood()) * requestedAmount);
    }
}
```

Figure 4.1: Barter strategy interface and our implementation of a function equivalent to Gintis' implementation.

4.3.2 Improvement strategies

In the original model, agents improve by being chosen as the worst performing in a pair. The worse agent then copies the prices from the better performing agent and possibly adjusts the prices up or down by a fixed factor. We have not generalised this to let the user implement new ways for the market to handle selection.

The improvement strategy is implemented on the agent side, letting the agent have full control over what to do when it is being selected for improvement. Fig. 4.2 shows the improvement strategy interface and our implementation of Gintis' original improvement rule (agent side).

The barter- and improvement strategies can be implemented in the same class if the need for a strongly optimising agent should arise.

4.3.3 Replacement/mutation strategies

The reproduction-mutation phase as described in section 3.3 in [Gin06] (from here on called $\mathcal{R}_{3.3}$) is very dissimilar to the one used in the original Delphi program (from here on called $\mathcal{R}_D$). There was an implementation (called `GetNextGeneration` in

```
public interface ImprovementStrategy extends Observer {
    public double[] improve(TradeAgentProxy betterAgent, TradeAgentProxy myself);
}

public class CopyAndMutateImprovementStrategy implements ImprovementStrategy {
    private int numGoods;
    private double mutationRate;
    private double mutationDelta;

    ...

    @Override
    public double[] improve(TradeAgentProxy betterAgent, TradeAgentProxy myself) {
        double[] newPrices = new double[numGoods];
        double[] priceMutation = getMutationVector();

        for (int good = 0; good < numGoods; good++) {
            newPrices[good] = betterAgent.getPrice(good) * priceMutation[good];
        }

        return newPrices;
    }

    ...
}
```

Figure 4.2: Improvement strategy interface and our implementation of a function equivalent to Gintis' `CopyAndMutate`.

`barterp.pas`, from here on called $\mathcal{R}_{3D}$) in the Delphi source of something that looked somewhat congruent to $\mathcal{R}_{3.3}$, but it was not used.

When we enabled $\mathcal{R}_{3D}$ in the original program, it failed to converge and got stuck at some particular average price instead of oscillating around some (possibly equilibrium) price. We do not know what the rationale for replacing the $\mathcal{R}_{3D}$ algorithm was, except that the implementation is broken. It sounds like a reasonable algorithm that could have some real-life correspondence; since it takes the entire population into account, weaker agents are more likely to copy the prices from the strongest agents *in a global sense*. On the other hand, $\mathcal{R}_D$ is conceptually simple and also as asymptotically fast as could be possible. Copying will be linear time in the number of goods, g. If k is the number of agents to replace, we can not expect a better asymptotic complexity than $O(gk)$.

The default behaviour of our Java program ($\mathcal{J}$) is to use the same algorithm as is in Gintis' program ($\mathcal{G}$) by default. Fig. 4.3 shows the strategy interface and our implementation (from here on called $\mathcal{R}_{DJ}$) of Gintis' Delphi version of the reproduction/mutation phase.

We also provide an implementation that we feel is a reasonable interpretation of $\mathcal{R}_{3.3}$ called `OriginalReplacementStrategy` (from here on $\mathcal{R}_{3J}$). $\mathcal{R}_{3J}$ should be

expected to be much slower than $\mathcal{R}_{DJ}/\mathcal{R}_D$, especially as replacement rates approach 0.5.

Not knowing how interesting $\mathcal{R}_{3.3}$ is[1] we have not provided a *fast* implementation of it in $\mathcal{R}_{3J}$. It should be possible to make an implementation of $\mathcal{R}_{3.3}$ having complexity in either $O(gn)$ or $O(gk \log k)$, g being the number of goods, n being the number of agents per good and k being the number of agents to replace. In comparison, $\mathcal{R}_{DJ}$ has complexity $O(gk)$ and we guess that $\mathcal{R}_{3J}$ has *average time* complexity on the order of $O(gn \log n)$. In the interest of time, we have not made a proper analysis of the complexity of our implementation of $\mathcal{R}_{3J}$. It should at least *almost surely*[2] terminate assuming a perfect stream of random numbers.

```
public interface ReplacementStrategy {
    public void getNextGeneration(List<TradeAgent> producers, BarterParams params,
                                  MersenneTwisterFast random);
}

public class DelphiReplacementStrategy implements ReplacementStrategy {
    @Override
    public void getNextGeneration(List<TradeAgent> producers, BarterParams params,
                                  MersenneTwisterFast random) {
        long replacements = Math.max(1, Math.round(producers.size() *
                                                   params.getReplacementRate()));

        for (int i = 0; i < replacements; i++) {
            int j = random.nextInt(producers.size());
            int k = random.nextInt(producers.size());

            if (producers.get(j).getScore() > producers.get(k).getScore()) {
                producers.get(k).improve(producers.get(j));
            } else {
                producers.get(j).improve(producers.get(k));
            }
        }
    }
}
```

Figure 4.3: Replacement/mutation strategy interface and our implementation of a strategy ($\mathcal{R}_{DJ}$) equivalent to $\mathcal{R}_D$.

[1] After all, it was not used in Gintis' original program

[2] `http://en.wikipedia.org/wiki/Almost_surely`

Chapter 5

Scalability

In this chapter, we study the scalability of our program with regard to model parameters. In particular, it is interesting to see how increasing the number of goods or agents influences the simulation performance. To support the results of time measurements, we first analyse the asymptotic time complexity of our program.

5.1 Asymptotic time complexity

When we started the work on this project we were quite surprised by how slow the simulation was even for relatively few agents and goods, such as the parameters Gintis originally used.

By analysing the time complexity of the model as described in [Gin06] we can see that we should not expect the simulation to ever be very fast due to *cubic* time complexity in the number of goods used.

We do not try to analyse the complexity with regards to memory use, cache behaviour, I/O, etc., but restrict ourselves to analysing the *asymptotic worst time complexity* of each function used in the original model.

Note that the pseudo-code given is a slight simplification; we have for most functions only included the parts that we need to facilitate time complexity analysis.

5.1.1 Definitions

Notation for execution time

$T(\text{FUNCTION}())$ is the time FUNCTION() takes to execute. $O(T(\text{FUNCTION}()))$ is the asymptotic time complexity for the execution of FUNCTION(). $O(T(a.b - a.c))$ is the time complexity for executing lines $a.b$ to $a.c$, inclusive.

Assumptions about complexity of individual operations

We assume that all functions for generating random numbers take constant time. Although this is not quite correct for RANDOMINT(), we deem it sufficiently close to constant to let us simplify the analysis (see C.2). Assignments, array element read/writes, application of binary/unary arithmetic and logical operators, memory allocation/calling an empty constructor, etc., are also assumed to be in $O(1)$.

5.2 Analysis of the barter-algorithm

The complete algorithm is shown in Alg. 5. To make the analysis as simple as possible, we show the complexity for the innermost functions and loops first, letting us eventually calculate the complexity for the full program as the last step.

Algorithm 1 Barter initialization pseudo-code

Parameters: g : number of goods, n : number of agents per good

```
1:  function INIT(g, n)
2:      for each i ∈ {1, ..., g} do
3:          for each j ∈ {1, ..., n} do
4:              a ← new agent.
5:              ▷ Set a's type/production good to i. ◁
6:              a_produces ← i
7:              for each k ∈ {1, ..., g} do
8:                  ▷ RANDOM() returns a random uniform value on [0, 1). ◁
9:                  a_price_k ← RANDOM()
10:             end for
11:             A_i ← A_i ∪ {a} ▷ Add a to A_i ◁
12:         end for
13:     end for

14:     return A
15: end function
```

5.2.1 Complexity of init

We start with the initialisation function, INIT(), as shown in Alg. 1. INIT() is called from MAIN(), Alg. 5.

Lines 1.1 to 1.1 execute a constant time function and an assignment g times for a complexity of $O(g)$.

Lines 1.1 to 1.1 execute n times. The creation and initialisation of a new agent (line 1.1) takes some constant plus $O(g)$ time for initialising the prices. Thus the loop 1.1 to 1.1 takes time $O(n)(O(g)+O(T(1.1-1.1))) = O(n)((O(g)+O(g))) \subseteq O(ng)$.

The outermost loop, lines 1.1 to 1.1, is executed g times giving final complexity

$$O(T(1.1-1.1))O(ng) \subseteq O(g)O(ng) \subseteq O(ng^2) \tag{5.1}$$

Algorithm 2 Barter improvement pseudo-code

Parameters: h : "well-performing" agent

```
 1: function IMPROVE(h)
 2:     ▷ Create new agent a having h's attributes. ◁
 3:     a ← h
 4:     for each i ∈ {1, ..., g} do
 5:         if RANDOM() < mutationRate then
 6:             if RANDOMBOOLEAN() then
 7:                 a_price_i ← a_price_i Δ_m
 8:             else
 9:                 a_price_i ← a_price_i Δ_m^-1
10:             end if
11:         end if
12:     end for

13:     return a
14: end function
```

5.2.2 Complexity of improve

IMPROVE(), Alg. 2, is called from REPRODUCEANDMUTATE(), Alg. 3.

The assignment in 2.3 is in $O(g)$, copying all prices from h to a. Each step in the loop 2.4 to 2.12 is in $O(1)$, loop g times, for a complexity of $O(g)$. The total complexity of IMPROVE() is thus

$$O(g) + O(g) \subseteq O(g) \tag{5.2}$$

5.2.3 Complexity of reproduceAndMutate

REPRODUCEANDMUTATE(), Alg. 3, is called from MAIN(), Alg. 5.

Lines 3.8 to 3.12 are all constant time operations with the exception of the call to IMPROVE(), being in $O(g)$.

Assuming $replacementRate \leq 1$, k on line 3.2 can be at most n. The first inner loop body, 3.6–3.13, is executed k times, $k \leq n$. All statements in the loop are constant time except the call to IMPROVE(), giving the first inner loop a complexity of $O(k)O(g) \subseteq O(kg) \subseteq O(ng)$.

The second inner loop body, 3.14–3.16, is executed n times and in each step calling a function in $O(1)$ for a complexity of $O(n)$.

Algorithm 3 Barter reproduction & mutation pseudo-code. Note that this is *not* equivalent to the Reproduction-Mutation phase described in section 3.3 of [Gin06]. This algorithm was taken from Gintis' Delphi version.

Parameters: A : list of lists of agents, g : number of goods, n : number of agents per good

```
 1: function REPRODUCEANDMUTATE(A, g, n)
 2:     k ← max(1, ⌊n × replacementRate⌋)
 3:     for each i ∈ {1, ..., g} do
 4:         for each j ∈ {1, ..., k} do
 5:             ▷ RANDOMINT(k) returns a random integer on {0, ..., k − 1}. ◁
 6:             a ← RANDOMINT(n) + 1
 7:             b ← RANDOMINT(n) + 1
 8:             if score(A_{g,a}) < score(A_{g,b}) then
 9:                 A_{g,a} ← IMPROVE(A_{g,b}) ▷ Copy/mutate from A_{g,b}'s prices. ◁
10:             else
11:                 A_{g,b} ← IMPROVE(A_{g,a}) ▷ Copy/mutate from A_{g,a}'s prices. ◁
12:             end if
13:         end for

14:         for each i ∈ {1, ..., n} do
15:             Reset score for A_{g,i}
16:         end for
17:     end for
18: end function
```

The outer loop body, 3.3–3.17, executes g times. The loop body has complexity $O(ng) + O(n)$ for a total of

$$O(g)(O(ng) + O(ng)) \subseteq O(ng^2) \tag{5.3}$$

5.2.4 Complexity of runPeriod

RUNPERIOD(), Alg. 4, is called from MAIN().

The loop body for 4.2–4.5 sets the inventory for each good, $O(g)$, and then sets the inventory of the production good, $O(1)$. This is repeated ng times (the total number of agents in the system) for a complexity of $O(g)O(ng) \subseteq O(ng^2)$.

Lines 4.19–4.23 are in $O(1)$ except the two calls to CONSUME(), each being in $O(g)$ for complexity $2O(g) \subseteq O(g)$. $O(T(4.19 - 4.23))$ is $O(g)$ or $O(1)$ depending on whether bartering was successful or not. It could also be claimed that it is $O(g)$ but we need to put it this way for analysing the complexity of the enclosing loop.

The enclosing loop, 4.15–4.25, is executed at most m times. The loop could terminate in two ways:

Algorithm 4 Barter pseudo-code for running one period

Parameters: A : list of lists of agents, g : number of goods, n : number of agents per good, m : maximum number of barter attempts

```
1: function runPeriod(A, g, n, m)
2:     for each a ∈ A_{1,...,g} do ▷ For each producer of each good ◁
3:         Set the inventory of a to zero for each good.
4:         Set the inventory of a's production good to g.
5:     end for

6:     p ← random permutation of {1, ..., g}.
7:     for each i ∈ {1, ..., g} do
8:         o ← random permutation of {1, ..., n}.
9:         for each j ∈ {1, ..., n} do
10:            for each k ∈ {1, ..., g} do
11:                if k ≠ i then ▷ Barter if the current agent is not of type p_k ◁
12:                    barterSucceeded ← false, t ← 0
13:                    ▷ The current agent is A_{p_i,o_j}. ◁
14:                    repeat
15:                        ▷ Choose a random agent to trade with. ◁
16:                        r ← randomInt(n) + 1
17:                        ▷ Let agent o_j of type p_i barter with agent r of type p_k ◁
18:                        barterSucceeded ← tryBarter(A_{p_i,o_j}, A_{p_k,r})
19:                        if barterSucceeded then
20:                            Exchange goods between A_{p_i,o_j} and A_{p_k,r}.
21:                            consume(A_{p_i,o_j})
22:                            consume(A_{p_k,r})
23:                        end if
24:                        t ← t + 1
25:                    until t ≥ m ∨ barterSucceeded
26:                end if
27:            end for
28:        end for
29:    end for
30: end function
```

- The loop terminates with m failed attempts, having complexity $O(m)$.
- The loop terminates since bartering succeeded. Calling CONSUME() is in $O(g)$.

Thus the worst case would be that the agent succeeds with bartering after $m-1$ attempts for complexity $O(m-1)+O(g) \subseteq O(m+g)$.

The inner good-loop body, 4.12–4.25, executes $g-1$ times due to the fact that the agent does not trade with agents of the same type (line 4.11). Thus the loop 4.10–4.27 has complexity $O(g)(O(m+g)) \subseteq O(g(m+g))$.

The agent-loop body, 4.10–4.27, executes n times for a complexity of $O(n)O(g(m+g)) \subseteq O(ng(m+g))$.

Creating a random permutation on $\{1,\ldots,n\}$ is in $O(n)$. Thus the loop body 4.8–4.28 is in $O(n)+O(ng(m+g)) \subseteq O(ng(m+g))$.

The outer good-loop body, 4.8–4.28, is executed g times and the body has complexity $O(ng(m+g))$ for a total complexity of $O(g)O(ng(m+g)) \subseteq O(ng^2(m+g))$.

Final complexity of RUNPERIOD() is

$$\begin{aligned} &O(T(4.2-4.5)+O(T(4.6))+O(T(4.7-4.29)) \subseteq \\ &O(ng^2)+O(g)+O(ng^2(m+g)) \subseteq O(ng^2(m+g)) \end{aligned} \tag{5.4}$$

Algorithm 5 Barter main program pseudo-code

Parameters: g : number of goods, n : number of agents per good, m : maximum number of barter attempts, r : periods between reproduction/mutation, p : number of periods to run

```
1: function MAIN(g, n, m, r)
2:     A ← INIT(g, n)
3:     for each i ∈ {1, ..., p} do
4:         RUNPERIOD(A, g, n, m)
5:         if i ≡ 0 (mod r) then
6:             REPRODUCEANDMUTATE(A, g, n)
7:         end if
8:     end for
9: end function
```

5.2.5 Complexity of main

MAIN() is the outermost level of the algorithm, shown in Alg. 5.

Initialisation is in $O(ng^2)$, running the program for one period is in $O(ng^2(m+g))$. Reproduction/mutation is in $O(ng^2)$, executing every rth period. Assuming the worst case for r, $r=1$, REPRODUCEANDMUTATE() will be called every period making $O(r) \subseteq O(1)$. The complexity for MAIN() is

Parameter	Value	Parameter	Value
agents	100	reproduce period	10
replacement rate	0.05	mutation rate	0.1
$\Delta_{mutation}$	0.95	max tries	5
goods	6	consume	1, 2, 3, 4, 5, 6

Table 5.1: Default parameters for timing tests of $\mathcal{J}$ and $\mathcal{G}$

$$\begin{aligned} O(T(\textsc{init}())) + O(p)O(T(\textsc{runPeriod}())) + \\ O(p)O(T(\textsc{reproduceAndMutate}()) \subseteq \\ O(ng^2) + O(png^2(m+g)) + O(png^2) \subseteq \\ O(ng^2(1+p(m+g+1))) \subseteq \\ O(png^2(m+g)) \end{aligned} \tag{5.5}$$

5.3 Testing the performance of different parameter sets

The asymptotic worst case behaviour may not be a good description of the actual running time. In Fig. 5.1 we show time measurements for Gintis' original parameter set, Table 5.1, varied over number of goods. As we can see by fitting the polynomial $y = ax^3 + bx^2 + cx + d$, x being the number of goods and y being time taken in seconds, a is typically much smaller than b. This observation implies that the actual running time is closer to cng^2. With n being the number of agents per good and g being the number of goods, for $g \geq 13$, the polynomial $f(n, g) = 100n(0.005g^3 + 0.077g^2 - 0.17x + 2.4)$ gave us the correct time within $\{-13, +6\}$ percent. The cubic coefficient (0.005) shows to be more than an order of magnitude smaller than the square coefficient (0.077) in practise.

5.3.1 Test environment

The timing tests were run on a dual dual core 2 GHz AMD Opteron 270 machine with 4 GB of RAM. We used the Java 1.6.0_07 JVM/JRE with the "`-server`" option for all tests. For some of the tests the program used slightly more than 100% CPU, implying that it for some duration was running on more than one core. The measured times are CPU-seconds, not real time.

5.4 Possible gains from concurrency

The loop 4.9–4.28 of RUNPERIOD() should be possible to parallelise completely. The dependencies are strictly one way, i.e. exactly one type of producer is acting as offerers and exactly one type is acting as responder. The responders should simply block if they are already within the barter code block. This is easily facilitated by Java's *synchronized* mechanism.

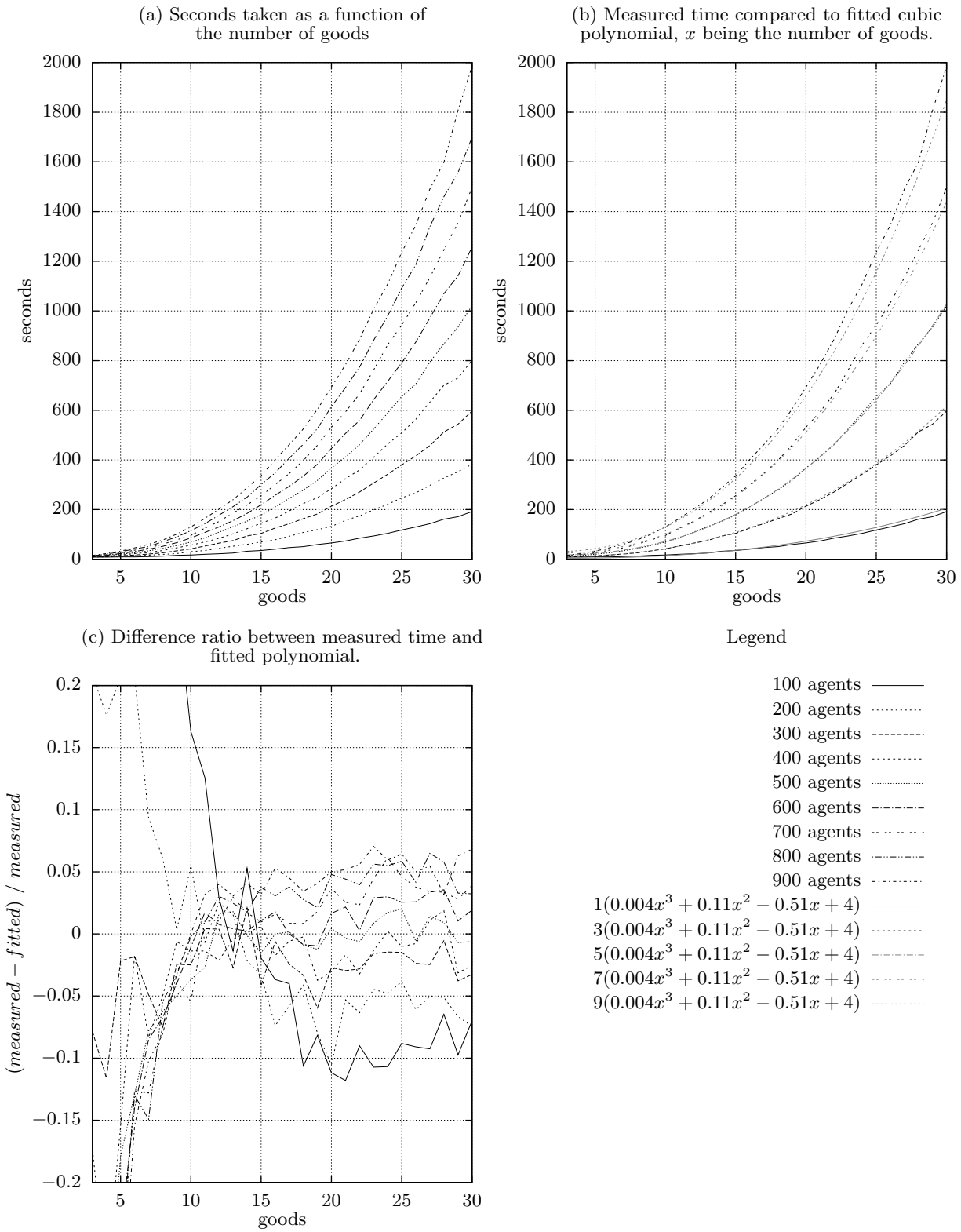


Figure 5.1: Time taken in seconds for running our program for 1000 periods, using the default parameters and varying the number of goods.

Chapter 6

Convergence

Here we examine how "close" we can say that our program is compared to Gintis' original implementation. One of the bigger problems is to decide on a notion of similarity for pseudo-random processes.

6.1 Global testing

> "... program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." [Dij72]

We need to gather evidence that Gintis' Delphi program, $\mathcal{G}$, and our Java version, $\mathcal{J}$, behave the same on a global scale. For any re-implementation where the architecture or structure has been changed, there will be other mechanisms than unit tests only needed.

One problem with writing a program for simulation is that one typically does not know what the results are expected to be. In our case, the expected output should be *similar* to what $\mathcal{G}$ is producing.

In [Axe03] Axelrod gives three levels of replication:

1. "Numerical identity" in which the results are reproduced precisely.
2. "Distributional equivalence" in which the results can not be distinguished statistically.
3. "Relational equivalence" in which the qualitative relationships among the variables are reproduced.

Since we could not (with a reasonable amount of work) completely control each and every mechanism used in the platforms for the two implementations, such as how floating point arithmetic is performed, we did not strive for "numerical identity" but only for "distributional equivalence". Hence we say that the programs are similar when they are distributionally equivalent.

This in turn begs the question what statistics to use to distinguish data sets; what statistics capture the essential features of the model? Not knowing what the essential features are, we will have to construct tests that make as few assumptions about relevant features as possible.

6.2 Random number generation

Since many of the decisions taken in Gintis' model are randomised, we wanted to make sure that any particular structure[1] of the Delphi pseudo-random number generator (PRNG) did not affect the convergence properties of the simulations in ways that should not be expected from a proper random sequence. We call the original Delphi program, using Delphi's built-in PRNG "$\mathcal{G}_o$".

See Appendix C for some analysis and tests of the Delphi PRNG as well as a description of the generator we replaced it with in Delphi. We call the Delphi version with the original PRNG replaced by the KISS99 PRNG "$\mathcal{G}_k$".

With the number of runs and tests we have done with $\mathcal{G}_o$ and $\mathcal{G}_k$, we can not conclusively say that the output of $\mathcal{G}_o$ differs from $\mathcal{G}_k$ in any significant way. We will still use the altered program, $\mathcal{G}_k$, as our reference.

MASON uses the Mersenne Twister MT19937 [MN98], a well known good generator with few known flaws [LS07] and a very long period ($> 10^{6001}$).

6.3 Our definition of similarity for price convergence

Defining a necessary condition for passing the test is easy; when partitioning the data from different runs from the reference process, each partition element should be regarded as a "pass" with regards to the other element. Output data from another process which we want to compare should (for some statistic) behave to the reference data as the reference data does to itself. This is our definition of "distributional equivalence".

Since we can not define what is considered *sufficient*, we will regard a process as a "pass" if it just satisfies the *necessary* condition of producing data that is no more different to the reference than the reference is to itself.

6.3.1 Similarity statistics

What data should be similar?

For some process, we sample average good prices relative to a reference good at uniformly distributed times t. We denote one such list of prices as $P_{X,s,S,g}$, X being

[1] For some types of random number generators, in particular linear congruential generators, some tuples of short lengths such as 2, 3 or 4 occur with much too non-uniform frequencies. See [Mar68] and [MZ93].

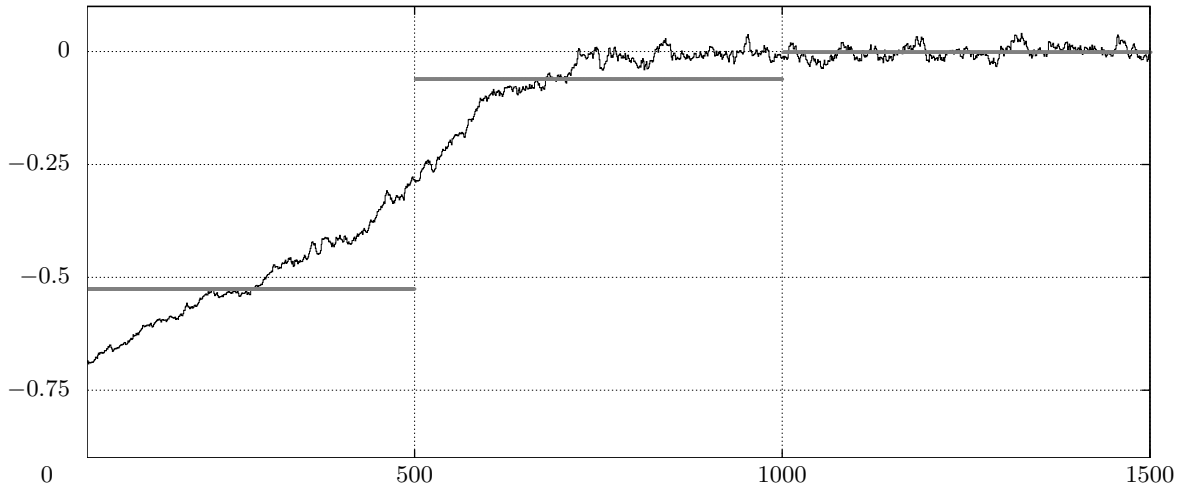


Figure 6.1: Time series and averages for 3 time slices of a good in a test run.

the name of the program, s being the random number generator seed, S being the starting conditions (parameter set excluding the seed) for the run and g being the index for the good. For our evaluation/description, we only use every 200th sample and run each process for at least as many periods as to expect convergence for at least one good.

Assuming we have some reference data $P_{R,s_{\dots},S,g}$ for fixed S but for a number of distinct s, we can compare sets of runs, $P_{R,s_I,S,g}$ to $P_{R,s_J,S,g}, I \cap J \equiv \emptyset, |I| \approx |J|$ and see how similar we should expect them to be.

Unfortunately, comparing the reference to itself is not feasible since we typically have a limited set of reference data. We have from testing seen that the way we partition the reference greatly influences the test results. Furthermore, to ensure that the samples really are independent, we have to make sure that we only take *one sample* from each run $r, P_{R,r,S,g}$; in practice, we will not be able to compare the entire time series for one good but will have to set for a few reference points. We create these reference points by segmenting a time series into a few slices of equal length and then compute the averages, see Fig. 6.1.

Testing for distribution equality

For the subsequent statistical tests we have used MATLAB[2]. We need to test the hypothesis that the distribution of prices at some period and for some good are the same. We start by testing if the populations are normal by way of the Lilliefors test[3], implemented in MATLAB as `lillietest`. For the samples we have taken, the Lilliefors test rejects the null-hypothesis, that the population is normal, at the 99.9% level.

[2]See `http://www.mathworks.com/`

[3]`http://en.wikipedia.org/wiki/Lilliefors_test`

Since we do not think the populations are normal, we need to use a non-parametric test; a test for similarity that does not have any a priori knowledge with regards to the distribution.

One such test that can be used for continuous populations is the *Kolmogorov-Smirnov test* (KS-test). The *one-sample* KS-test gives a probability that a sample comes from a statistical distribution with known parameters (given as a CDF). Since we do not know what distribution to expect (and thus even less so, the parameters), we will use the *two-sample* KS-test.

The two-sample KS-test takes two samples, computes their empirical [cumulative] distribution functions (ECDFs) and based on the maximum distance between the two "functions", gives a probability that they are sampled from the same distribution. A practical description of the two-sample KS-test can be found in [She00]. See Fig. 6.2 for some examples of ECDFs and corresponding KS-test probabilities. The two-sample KS-test is implemented as `kstest2` in MATLAB.

For each time-slice s_t we should expect two samples from the population of runs for a fixed good to have the same distribution for the two programs. For each parameter set, we apply Algorithm 6. If *all* goods fail the KS-test at the 10^{-5} level, we reject the null-hypothesis (same CDF) and regard the programs as different.

Choice of H_0

It is not clear that there is one uniquely perceived (dis)similarity between two programs and thus not clear that one particular null-hypothesis is superior to another. The choice of distributional equivalence (in the KS-test sense) may be the most stringent but a weaker hypothesis might also do.

Two weaker null-hypotheses that could be of interest is "same median" and "same mean":

- $H_0 \iff$ *populations have same median.* The *Wilcoxon rank-sum test* is a robust test (insensitive to outliers) giving a probability estimate that two populations have the same median.
- $H_0 \iff$ *populations have same mean.* An interval estimate of the difference of means, $\mu_1 - \mu_2$, indicates that the means are the same when the interval contains zero.

To illustrate the difference in sensitivity to differences in distributions, we have created Fig. 6.3 and Table 6.1. Each table entry describes the probability that the null-hypothesis holds for the KS-test (same distribution) and the rank-sum test (same median).

As can be seen in Table 6.1, rows 2a, 2b, columns 1a, 1b, the rank-sum test is more sensitive than the KS-test to differences in medians. The KS-test is more

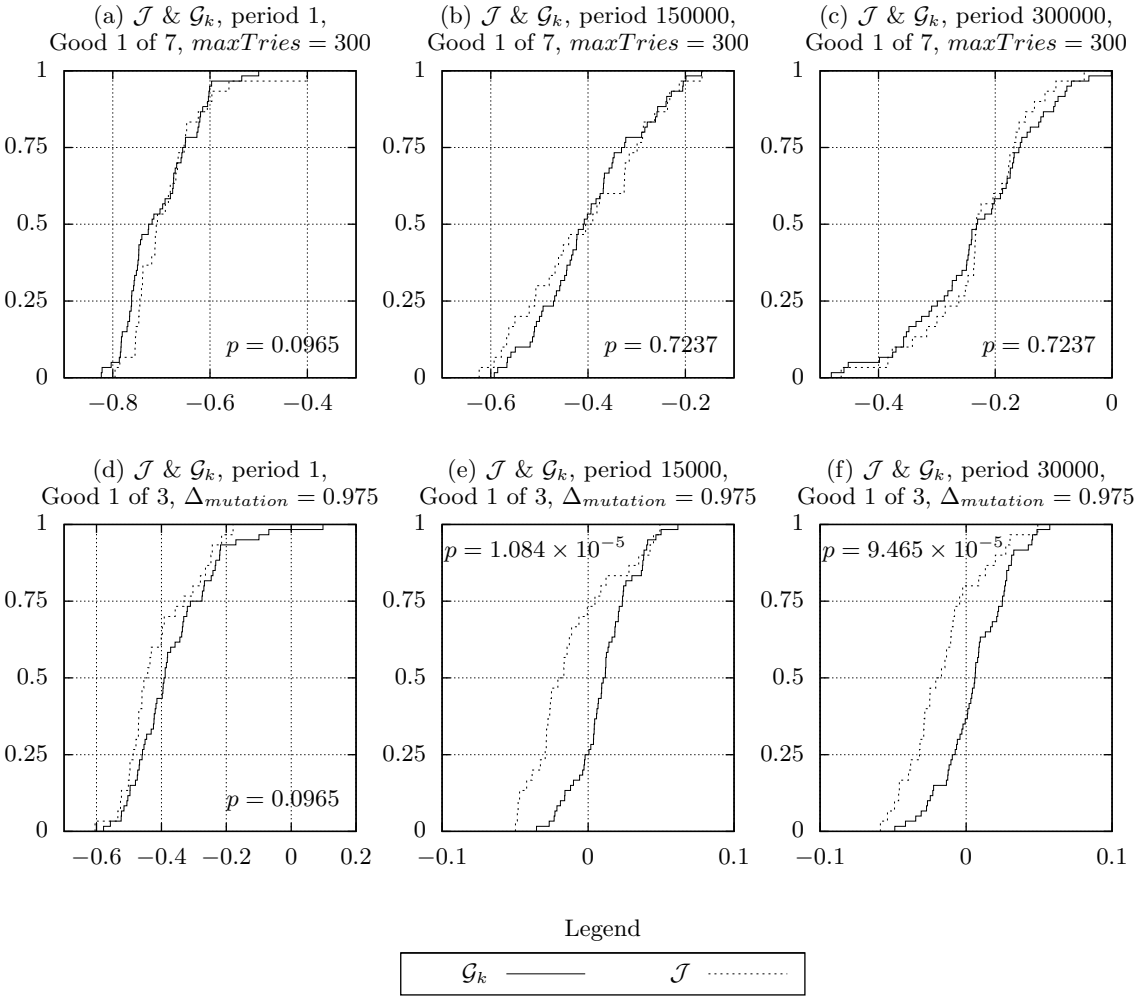


Figure 6.2: Empirical CDFs and KS-test probabilities.

sensitive to differences of variance and distribution *shape* as can be seen when comparing 1b to 3b or, more spectacularly, 3b to 4b. The slight difference between the tests when comparing 4a to 4b stems from the fact that even though the means and variances are the same for these populations, the medians are not; median for 4a is -0.116 and median for 4b is -0.076.

Algorithm 6 Similarity test for time series

Parameters: x_{ref} : reference time series, x_{cmp} : time series to compare, s : time slices, t : test repetitions

Require: $rows(x_{ref}) = rows(x_{cmp})$

Require: $s \leq \min(cols(x_1), cols(x_2))$.

▷ The time series data are three-dimensional arrays indexed as $s_{i,j,k}$ where i is the good index, j is the time index and k is the test run index. ◁

1: **function** MULTITIMESERIESTEST(x_{ref}, x_{cmp}, s, t)
2: **for each** $i \in \{1, \ldots, t\}$ **do**

▷ Partition each set of test data into s partition elements of $\approx$ equal size. Each partition is over the test runs meaning that each partition element will contain *all* good prices over time from some subset of test runs. ◁

3: $r_{ref,i} \leftarrow$ a random partition of runs from x_{ref}.
4: $r_{cmp,i} \leftarrow$ a random partition of runs from x_{cmp}.

▷ Now $r_{ref,i,j,k}$ will be the price of good i at time j for test run k (out of $testruns(r_{ref})/s$). ◁

5: **end for**
6: **for each** $j \in \{1, \ldots, goods\}$ **do**
7: **for each** $k \in \{1, \ldots, s\}$ **do**
8: **for each** $i \in \{1, \ldots, t\}$ **do**
9: **for each** $l \in \{1, \ldots, testruns(r_{ref,i})\}$ **do**

▷ Use disjoint subsets of p (partition elements) for each time slice to ensure that the test of time slice s_x will not influence the test of slice s_{x+1}. ◁

10: $m_{ref,i,l} \leftarrow$ mean of good j for time slice k for element l of $r_{ref,i}$.
11: **end for**
12: **for each** $l \in \{1, \ldots, testruns(r_{cmp,i})\}$ **do**
13: $m_{cmp,i,l} \leftarrow$ mean of good j for slice k for element l of $r_{cmp,i}$.
14: **end for**

▷ Compute the KS-probability that the samples m_{ref} and m_{cmp} come from the same distribution. ◁

15: $p_i \leftarrow$ KSTEST2(m_{ref}, m_{cmp})
16: **end for**

▷ Let score for good j at slice k be the median KS-probability of the t tests. ◁

17: $P_{j,k} \leftarrow median(p)$
18: **end for**
19: **end for**

20: **return** P
21: **end function**

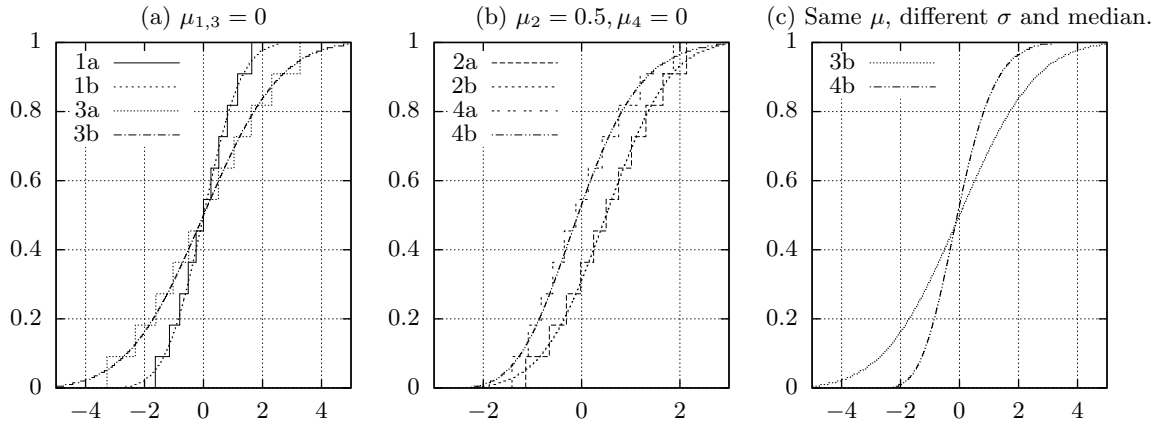


Figure 6.3: Empirical CDFs for different sized samples and parameters.

	Test	1a	1b	2a	2b	3a	3b	4a
1b	KS	1.00000	—	—	—	—	—	—
	rank-sum	1.00000	—	—	—	—	—	—
2a	KS	0.98517	0.54975	—	—	—	—	—
	rank-sum	0.32464	0.13000	—	—	—	—	—
2b	KS	0.54975	0.00081	1.00000	—	—	—	—
	rank-sum	0.13000	$< 10^{-6}$	1.00000	—	—	—	—
3a	KS	0.98517	0.64977	0.73584	0.25911	—	—	—
	rank-sum	1.00000	1.00000	0.59936	0.35197	—	—	—
3b	KS	0.69378	0.00748	0.29082	$< 10^{-6}$	1.00000	—	—
	rank-sum	1.00000	1.00000	0.32634	0.00246	1.00000	—	—
4a	KS	1.00000	0.99998	0.73584	0.34630	0.73584	0.49799	—
	rank-sum	0.94764	0.89708	0.26429	0.10187	1.00000	0.97370	—
4b	KS	1.00000	0.99961	0.41489	0.00006	0.56488	0.00259	1.00000
	rank-sum	0.92927	0.73164	0.10619	$< 10^{-7}$	0.98988	0.93210	0.97370

Table 6.1: Comparing populations with the KS-test and the rank-sum test.

6.3.2 Actual testing procedure

Any statistical testing of a system as complex as the barter model will at best be superficial; the complete parameter space is immensely large. By choosing a default parameter set (Table 6.2) and vary one parameter at a time we yet hope to get a representative image of the programs' behaviours.

For each parameter set we have chosen, $\mathcal{G}_k$ and $\mathcal{J}$ has been run at least 60 times each for a number of periods at which we think they have converged for most goods.

Parameter	Value	Parameter	Value
agents	100	reproduce period	10
replacement rate	0.5	mutation rate	0.4
$\Delta_{mutation}$	0.995	max tries	5
goods	3	consume	1, 2, 3

Table 6.2: Default parameters for convergence tests of $\mathcal{J}$ and $\mathcal{G}_k$. If more than three goods are used, the consumption rate of good g equals the good index. E.g. consume for 5 goods would be 1, 2, 3, 4, 5.

Test results

In Table 6.3 we show the results of our tests. The "Parameter" column lists what parameters, if any, were changed compared to the default parameters in Table 6.2. KS_A is the mean KS-probability for the time slices we sample, KS_t is the median KS-probability for slice t of 3 over 50 attempts.

We say that $\mathcal{J}$ passes the test for a parameter set if $KS_A \geq 0.05$ (marked with a ✓) *and does not* have "catastrophic failures" for both goods. We call a KS-probability less than 0.001 a catastrophic failure and mark it with a ✗. If any time-segment has a catastrophic failure, the convergence for that good is marked as a fail. If the test fails for all goods, either by having at least one catastrophic failure for all goods or all KS_A being less than 0.05, we mark the result as a ☠.

The $u_1 - u_2$ column gives a 99% confidence interval for the mean difference between the mean of the last 10% of the periods for a good. If the means of the populations really are the same, the interval should contain zero.

$|\mathcal{G}_k|$ and $|\mathcal{J}|$ denote how many runs of $\mathcal{G}_k$ and $\mathcal{J}$ were used for the test. With 3 time slices, the number of samples for each KS-test is $\frac{1}{3}\{|\mathcal{G}_k|, |\mathcal{J}|\}$.

A brief interpretation of Table 6.3

For most parameter sets we have chosen, $\mathcal{J}$ passes the KS-test as well as either having a mean difference interval that contains zero or being very close (± 0.2) to containing zero.

Unfortunately, there are four parameter sets that do fail the KS-test.

For one of the sets, the "Default", the difference in means is very small ([+0.08, +0.34], [+0.53, +0.82]). We think that the convergence points for $\mathcal{G}_k$ and $\mathcal{J}$ might be the same. A few symmetric outliers will make the KS-test fail (higher variance) and a few assymmetric outliers will make the rank-sum test fail (see Figure D.5.1), even if the means really are the same.

For the three other bad cases, $\Delta_{mutation} = 0.975$, $repr.Period = 400$ and $repl.Rate = 0.75$, we can not tell where the discrepancies come from. In the in-

terest of time, we at some point had to give up looking for the reason(s) to be able to finish the report.

One of the points in the programs we have given extra attention to is the permutation generation (see C.1.2) but we are confident that the permutation generattion code in both programs behave identically in a statistical sense.

One possibility that we have not examined thoroughly is round-off errors. Even though we have made sure that the *same precision* floating point arithmetic is used in $\mathcal{G}_k$ and $\mathcal{J}$, exceptional events such as division by zero or handling of NaN may be handled differently. Making a deeper study of the differences (or similarities) between Java 6 and Delphi 7 with regards to floating of arithmetic seems like a daunting task.

6.4 Level of replication

By observing Table 6.3 we can see that we for some parameter sets fall short of our goal to reach "distributional equivalence". Based on Table 6.3 and the plots in Appendix D, we can however say that we do reach "relational equivalence"; if the parameter changes did not have the same general effects in both programs, we would see $\mu_1 - \mu_2$ interval estimates that were off by 10% or more.

Parameter	G	KS_μ	KS_1	KS_2	KS_3	$\mu_1 - \mu_2$	Figures	$\lvert\mathcal{G}_k\rvert$	$\lvert\mathcal{J}\rvert$	Result
10 agents	1	0.433	0.610	0.457	0.233	$[-1.14, +0.33]$	D.2 a–d	60	119	✓
	2	0.480	0.610	0.457	0.372	$[+0.27, +0.51]$				✓
1000 agents	1	0.267	0.128	0.129	0.544	$[-0.33, +0.58]$	D.2 e–h	96	130	✓
	2	0.233	0.513	0.183	0.004	$[-0.00, +0.41]$				✓
Default	1	0.347	0.502	0.540	✗	$[+0.08, +0.34]$	D.5 l	100	130	☠
	2	0.203	0.505	0.103	✗	$[+0.53, +0.82]$				
$maxTries = 300$	1	0.599	0.529	0.719	0.547	$[-3.92, +3.20]$	D.5 k	60	129	✓
	2	0.595	0.548	0.704	0.533	$[-0.73, -0.05]$				✓
$maxTries = 300$	1	0.467	0.278	0.654	0.468	$[-0.31, +3.51]$	D.5 j	60	114	✓
	2	0.592	0.620	0.687	0.468	$[-0.07, +0.80]$				✓
	3	0.466	0.679	0.389	0.331	$[+0.11, +0.80]$				✓
$maxTries = 300$	1	0.588	0.689	0.466	0.608	$[-4.52, +5.47]$	D.1 e–h	60	128	✓
	2	0.378	0.350	0.584	0.201	$[-2.72, -1.69]$				✓
	3	0.543	0.644	0.544	0.441	$[-0.96, +1.08]$				✓
	4	0.229	0.523	0.129	0.035	$[-2.48, -0.37]$				✓
$maxTries = 300$	1	0.440	0.457	0.327	0.535	$[-7.69, +11.5]$	D.1 a–d	60	119	✓
	2	0.523	0.457	0.457	0.656	$[+0.33, +3.20]$				✓
	3	0.480	0.457	0.457	0.527	$[-0.22, +1.44]$				✓
	4	0.370	0.148	0.392	0.570	$[-2.15, +0.92]$				✓
	5	0.491	0.457	0.457	0.558	$[-2.30, +1.65]$				✓
	6	0.273	0.327	0.121	0.370	$[-0.17, +5.38]$				✓
$\Delta_{mut.} = 0.75$	1	0.348	0.362	0.369	0.313	$[-1.77, +5.15]$	D.3 a–d	60	114	✓
	2	0.637	0.604	0.654	0.654	$[-3.01, +4.25]$				✓
$\Delta_{mut.} = 0.975$	1	0.114	0.342	✗	✗	$[+1.67, +2.36]$	D.5 g	100	129	☠
	2	0.088	0.264	✗	✗	$[+1.86, +2.69]$				
$\Delta_{mut.} = 0.9975$	1	0.437	0.584	0.648	0.080	$[-4.55, -1.19]$	D.3 e–h	60	129	✓
	2	0.367	0.547	0.551	0.002	$[+0.38, +0.79]$				✓
$mut.Rate = 0.01$	1	0.508	0.610	0.457	0.457	$[-6.01, +7.20]$	D.5 a	60	120	✓
	2	0.559	0.610	0.457	0.610	$[-1.76, -1.16]$				✓
$mut.Rate = 0.1$	1	0.598	0.654	0.563	0.579	$[-5.71, +9.65]$	D.5 b	60	114	✓
	2	0.511	0.470	0.375	0.687	$[-0.04, +0.22]$				✓
$mut.Rate = 1$	1	0.448	0.537	0.622	0.185	$[+0.43, +0.78]$	D.5 c	60	130	✓
	2	0.290	0.487	0.380	0.004	$[+0.80, +1.21]$				✓
$maxTries = 3$	1	0.678	0.704	0.777	0.554	$[-8.32, +2.69]$	D.4 a–d	60	115	✓
	2	0.339	0.483	0.403	0.132	$[+0.25, +0.38]$				✓
$maxTries = 30$	1	0.290	0.250	0.332	0.289	$[+0.04, +0.23]$	D.4 e–h	60	117	✓
	2	0.192	0.511	0.062	0.001	$[+0.27, +0.50]$				✓
$repr.Period = 100$	1	0.578	0.671	0.663	0.401	$[-2.00, +1.69]$	D.5 h	60	130	✓
	2	0.517	0.626	0.487	0.440	$[+0.24, +0.55]$				✓
$repr.Period = 400$	1	0.318	0.511	0.442	✗	$[-0.78, -0.15]$	D.5 i	95	130	☠
	2	0.123	0.296	0.074	✗	$[+0.41, +1.37]$				
$repl.Rate = 0.01$	1	0.434	0.295	0.506	0.502	$[-12.8, +7.48]$	D.5 d	60	123	✓
	2	0.658	0.644	0.675	0.656	$[-0.32, -0.01]$				✓
$repl.Rate = 0.1$	1	0.488	0.399	0.527	0.539	$[-4.06, +7.35]$	D.5 e	60	118	✓
	2	0.459	0.635	0.515	0.227	$[+0.36, +1.07]$				✓
$repl.Rate = 0.75$	1	0.161	0.403	0.079	✗	$[+0.39, +0.61]$	D.5 f	100	130	☠
	2	0.182	0.532	0.014	✗	$[+0.57, +0.87]$				

Table 6.3: KS-tests and mean difference intervals comparing $\mathcal{G}_k$ to $\mathcal{J}$. The "Parameter" column lists what parameters, if any, were changed compared to the default parameters in Table 6.2.

Chapter 7

Further work

Throughout the thesis work, we have been prioritising various features of the program to fit in the time limit. In this chapter, we point out the features that were left out, but might be useful further developments.

7.1 Parallelisation

The asymptotic time complexity of the model proved to be quite bad. To speed up the simulation, it could help to perform certain simulation steps concurrently. We identified the parts of the program that could benefit from parallelisation in 5.4.

7.2 Different ways of outputting data

To analyse the dynamics of a simulation run in other ways than just looking at different charts, it is desirable to have the data outputting as an extension point. The user can then direct and format the output in a custom way.

7.3 Dynamically loadable extensions

The way users can experiment with their own strategies is to write a new Java main class and inject custom strategies programmatically as described in Appendix B. It would be more convenient to load the strategy classes through the GUI.

7.4 Visualisation of different strategies

The visualisation of the agents should reflect the differences in used strategies. For example, if a group of agents uses a different bartering strategy, one should be able to identify these agents without clicking and inspecting each agent individually.

Chapter 8

Conclusions

We have developed an extensible and portable simulation tool for barter economics, with the focus on replicating Gintis' original barter economy model. We have shown the convergence between his proof of concept implementation and our program, although the task proved to be much more challenging than we anticipated. There are two sides to convergence—making the models match in the first place, but at the same time, convincing ourselves that they really do match, or the opposite if they do not. Neither of these problems is trivial for the agent-based models involving randomness.

It is easy to overlook seemingly insignificant implementation details. Examples include the random number generation method, rounding of the floating-point values or the order of simulated events. These details are hard to notice, but they are important contributors to the global model behaviour.

Once we have a model to compare to the original, it becomes necessary to formulate what it means for the two programs to converge. The difficulty here lies in defining a strong enough non-parametric statistic to say that we can not tell the original apart from the remake. At the same time, the statistic should match the essence of what is relevant in the program.

The discrepancies between the description and the original implementation of the barter economy confirm the importance of replication. Although we were able to reproduce the main property of the model, the emergence of equilibrium prices, we also found programming errors that affect the way this property is reached.

Besides the efforts in replicating Gintis' work, we have taken the model one step further by providing means for visualising individual agents and extending the model with different agent or market behaviours. Those additions improve the intuitive understanding of the model and make it possible to study the emergent properties of heterogeneous agent behaviours. MASON proved to be a suitable platform for these purposes—it does not place restrictions on the types of models that can be simulated, and provides easy means for creating a GUI and visualising agents.

If we were to do a similar project over, we would first try to achieve the numerical identity to the reference program. Then change the design step by step while observing that the numerical identity is preserved. This approach could save us from statistically testing for convergence.

Bibliography

[Axe03] Robert Axelrod. Advancing the Art of Simulation in the Social Sciences. *Japanese Journal for Management Information System*, 12(3):3–16, December 2003.

[Axt00] Robert Axtell. Why Agents? On the Varied Motivations for Agent Computing in the Social Sciences. *Center on Social and Economic Dynamics, Brookings Institution, Working Paper No. 17*, November 2000.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15:859–866, 1972.

[EM08] Pelle Evensen and Mait Märdin. Delphi PRNG? *Usenet posting to alt.comp.lang.borland-delphi*, 2008. `http://groups.google.se/group/alt.comp.lang.borland-delphi/msg/0fca42741d00276d`.

[Eps06] Joshua M. Epstein. *Generative Social Science: Studies in Agent-Based Computational Modeling*, chapter 1. Princeton University Press, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, pages 208–210. Addison-Wesley, 1995.

[Gin06] Herbert Gintis. The Emergence of a Price System from Decentralized Bilateral Exchange. *Contributions to Theoretical Economics*, 6(1):1302–1302, 2006.

[Gin07] Herbert Gintis. The Dynamics of General Equilibrium. *Economic Journal*, 117(523):1280–1309, October 2007.

[Gre97] Amy Greenwald. Modern Game Theory: Deduction vs. Induction. Technical report, New York University, January 1997.

[Knu97] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, pages 10–15. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[LCRPS04] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. MASON: A New Multi-Agent Simulation Toolkit. In *Proceedings of the 2004 Swarmfest Workshop*, 2004.

[LS07] Pierre L'Ecuyer and Richard J. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–22:40, 2007.

[LSCK02] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An objected-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, Nov–Dec 2002.

[Mar68] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25–28, 1968.

[Mar99] George Marsaglia. Random numbers for C: End, at last? *Usenet posting to sci.stat.math*, 1999. `http://groups.google.com/group/sci.stat.math/msg/b555f463a2959bb7`.

[Met87] Nicholas Metropolis. The Beginning of the Monte Carlo Method. *Los Alamos Science*, 15:125–130, 1987.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[MS97] Leonid Mikhajlov and Emil Sekerinski. The Fragile Base Class Problem and Its Solution. Technical Report 117, Turku Centre for Computer Science, 1997.

[MZ93] George Marsaglia and Arif Zaman. Monkey tests for random number generators. *Computers and Mathematics with Applications*, 26(9):1–10, 1993.

[Nas50] John Nash. Equilibrium Points in N-Person Games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

[RLJ05] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based Simulation Platforms: Review and Development Recommendations. *SIMULATION*, 82(9):609–623, September 2005.

[Sch69] Thomas C. Schelling. Models of Segregation. *The American Economic Review*, 59(2):488–493, 1969.

[Sch78] Thomas C. Schelling. *Micromotives and Macrobehavior*, chapter 4. Norton, 1978.

[She00] David J. Sheskin. *The Handbook of Parametric and Nonparametric Statistical Procedures*, pages 319–329. Chapman & Hall/CRC, second edition, 2000.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA*, pages 38–45, 1986.

[Tes03] Leigh Tesfatsion. Agent-Based Computational Economics. *Iowa State University Economics Working Paper No. 1, USA*, August 2003.

[Tes06] Leigh Tesfatsion. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. In *Handbook of Computational Economics*, volume 2, chapter 16, pages 831–880. North Holland, June 2006.

[Vic02] Tamás Vicsek. Complexity: The Bigger Picture. *Nature*, 418(6894):131, July 2002.

Appendix A

User documentation

A.1 Installation instructions

Requirements: *Java Runtime Environment version 6.*

The program can be obtained from `http://www.evensen.org/barter`. To unpack (assuming a UNIX-like system):

```
tar xvf barterEconomy.tar.gz
```

This will create a `barterEconomy` directory with all the required libraries in it. Now the program can be run by:

```
java -jar barterEconomy/barterEconomy.jar
```

A.2 Using the program

A.2.1 The About tab

When the program is first started, the ABOUT tab is displayed in the MASON console. It contains the description of the simulation model (Fig. A.1).

A.2.2 The Model and Displays tabs

Before starting a simulation run, it is possible to change the model parameters in the MODEL tab or activate/deactivate various displays in the DISPLAYS tab (Fig. A.2). To change the number of goods, the CONSUME array has to be changed in the MODEL tab.

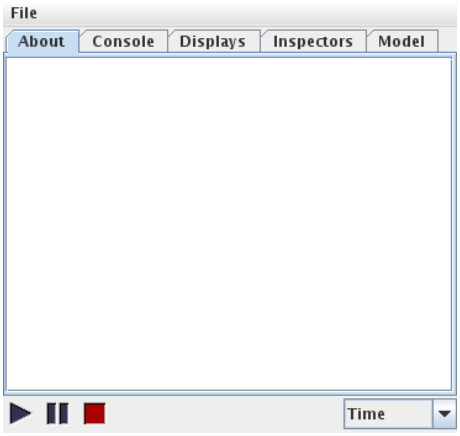


Figure A.1: The ABOUT tab with the model description in the MASON console

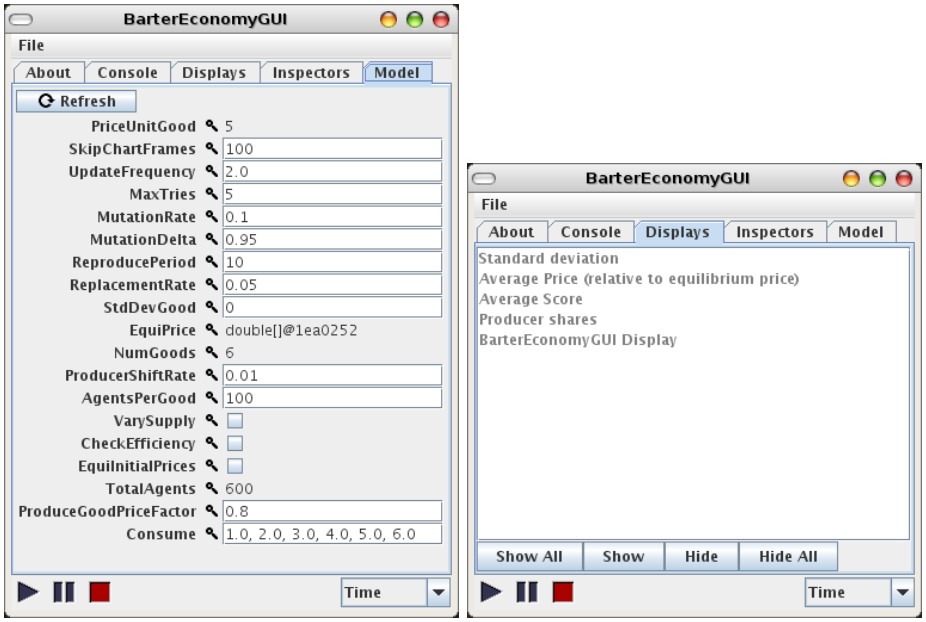


Figure A.2: The MODEL and DISPLAYS tab

A.2.3 The Average Score chart

Under the DISPLAYS tab, it is possible to activate four different charts. The AVERAGE SCORE chart displays time series for the average scores of agents grouped by their production good, but also the total average score among all agents. In a 6-good economy, there will be 7 series as shown in Fig. A.3.

A.2.4 The Average Price chart

The AVERAGE PRICE chart displays time series for the average prices of the goods relative to the equilibrium prices. As one good is taken as the price unit, its price will always be constant ($y = 0$) in the chart. Thus, in a 6-good economy, we get 5

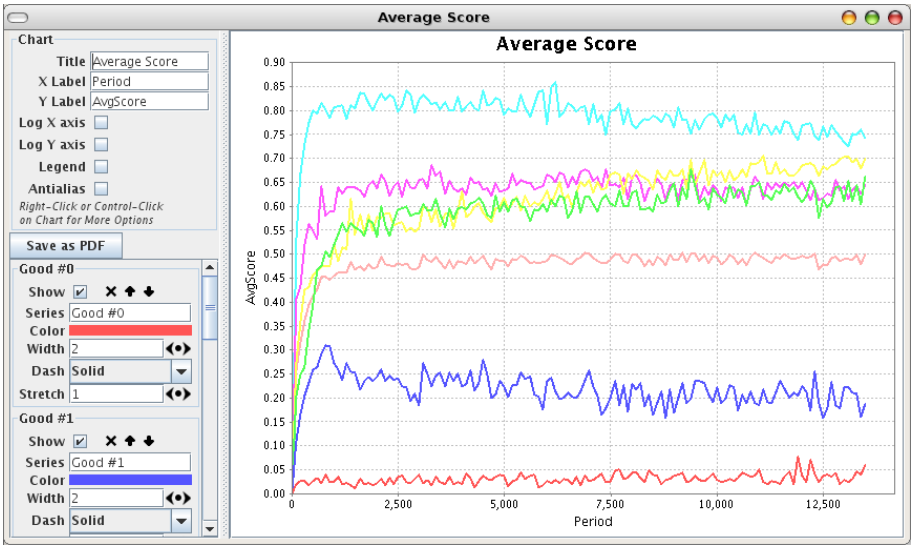


Figure A.3: The AVERAGE SCORE chart

prices that vary over time (Fig. A.4).

Figure A.4: The AVERAGE PRICE chart

A.2.5 The Producer Shares chart

The PRODUCER SHARES chart is interesting if the VARYSUPPLY option is turned on under the MODEL tab. It shows the share that the producers of a particular good have. If there is an equal number of producers for each good it just displays

a number of constant series equal to the number of goods in the model. Fig. A.5 shows what happens when varySupply is turned on mid-run.

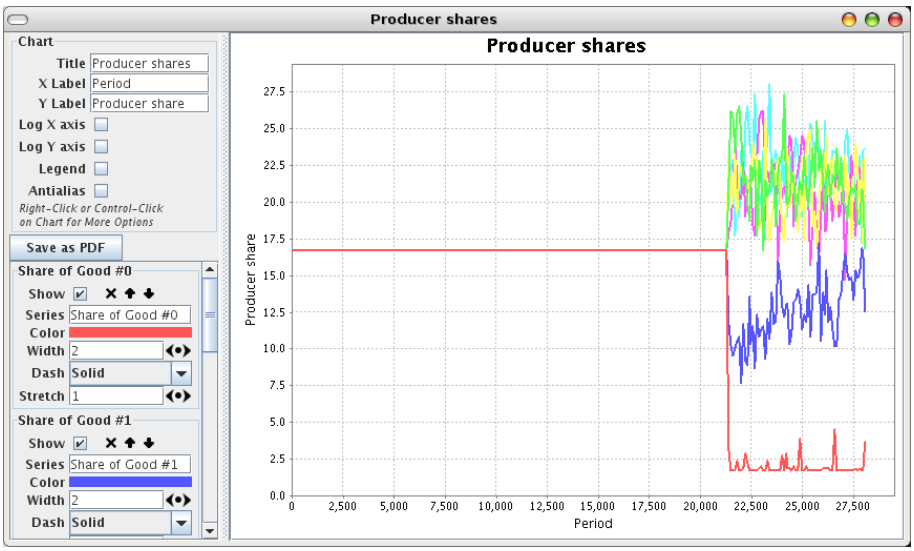


Figure A.5: The Producer Shares chart

A.2.6 The Standard Deviation chart

The Standard Deviation chart displays three different standard deviation series: *consumer price*, *producer price*, and *mean price* (Fig. A.6). All these are calculated for one good only, which can be specified in the Model tab with the *stdDevGood* parameter. The standard deviation of *consumer price* considers the prices from all the agents except the producers of the good. The standard deviation of *producer price* takes into account only the prices from the producers themselves. Finally, the standard deviation of *mean price* considers the mean prices of the good for last 100 periods.

A.2.7 Visualising the agents

To get a better understanding of the economy at the agent level, the Agent Visualisation can be activated under the Displays tab. This brings up a display similar to the one in Fig. A.7. Any active display will slow down the simulation significantly, but this one is the worst (with respect to performance) and *should not* be enabled if simulation speed is important. Since there typically is no need to see what happens in each round, setting the Skip field in the Agent Visualisation to a value of about 100, lessens the impact the display has on the overall speed.

In this visualisation, each agent is portrayed as a circle. This circle in turn is split up to provide various kinds of information about the agent. The innermost

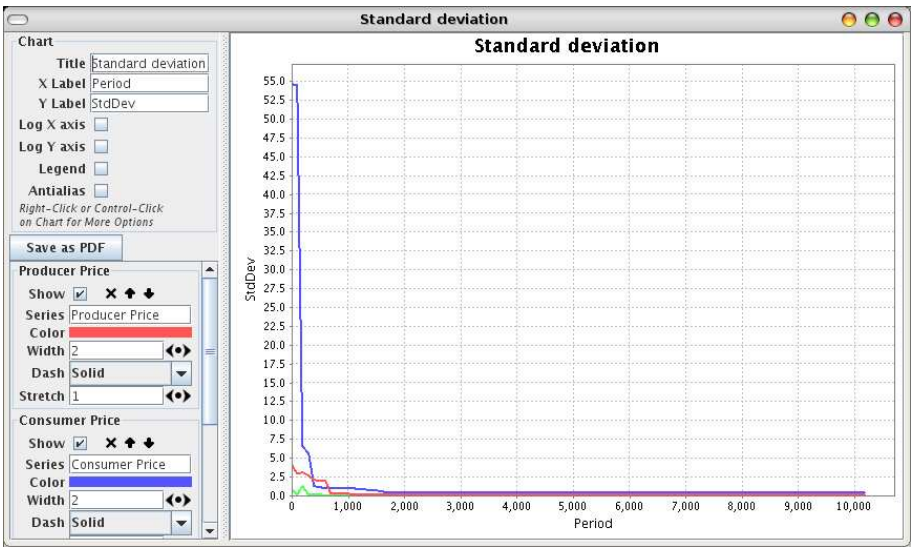


Figure A.6: The Standard Deviation chart

Figure A.7: Visualised agents

circle reflects the production good of the agent. The next layer shows the score—the darker it is, the higher the score for the agent (see Fig. A.8). The outermost pie chart shows the price ideas of the agent (larger share means higher price for that good).

A.2.8 Inspecting an agent

To get even more detailed information about an individual agent, one has to first select it by double-clicking it. This marks the agent and opens up the Inspectors tab in the MASON console as shown in Fig. A.9.

Various fields from the agent class are shown in the Inspectors tab, and by

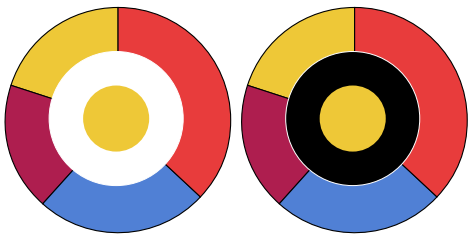


Figure A.8: A low scoring agent (on the left) vs. a high scoring agent

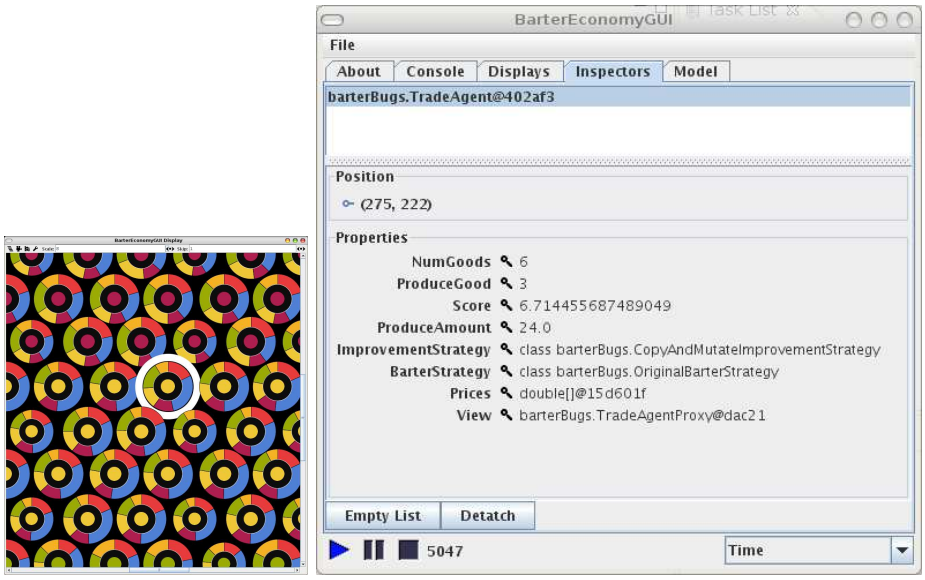


Figure A.9: Selecting an agent for inspection

clicking on the magnifying class it is possible to see even further details about composite types—for example the price array (see Fig. A.10), updated in real time.

Figure A.10: Inspecting the prices of an individual agent

Appendix B

Developer documentation

The goal of this Appendix is to instruct the user in extending the model.

B.1 The strategy interfaces

The first step in extending the model is to write an implementation of one of the three strategy interfaces. All three of those interfaces (`BarterStrategy`, `ImprovementStrategy` and `ReplacementStrategy`) are in the `barter` package and contain one method each as shown in Fig. B.1.

```
public interface BarterStrategy {
    public boolean acceptOffer(TradeAgentProxy me, int offersGood,
                               double offersAmount, double requestsAmount);
}

public interface ImprovementStrategy {
    public double[] improve(TradeAgentProxy betterAgent, TradeAgentProxy myself);
}

public interface ReplacementStrategy {
    public void getNextGeneration(List<TradeAgent> producers, BarterParams params,
                                  MersenneTwisterFast random);
}
```

Figure B.1: Interface methods of the strategies.

A correct implementation of the `acceptOffer` method in `BarterStrategy` just returns a `boolean` value, indicating whether the agent accepts the offer specified by input parameters.

The `improve` method in `ImprovementStrategy` is expected to return an improved price vector based on the information about two agents: itself and a better scoring agent. The length of the improved price vector should be the same as it is for the two agents.

The `getNextGeneration` method in `ReplacementStrategy` is a bit different as it expects the user to pick pairs of agents from the `producers` list and call the improve method on them as follows:

```
producers.get(i).improve(producers.get(j), params, random);
```

Figure B.2: Telling the `i`-th agent to improve its prices based on `j`-th agent.

B.2 Running the model with custom strategies

Once the strategy implementations are complete, they can be put in use by creating a new main class as shown in Fig. B.3.

```
1  import barter.BarterEconomy;
2  import barter.BarterParams;
3  import barter.CopyAndMutateImprovementStrategy;
4  import barter.DelphiReplacementStrategy;
5  import barter.GenerousBarterStrategy;
6  import barter.OriginalBarterStrategy;
7  import barter.display.BarterEconomyGUI;
8
9  public class Main {
10         public static void main(String[] args) {
11                 BarterEconomy eco = new BarterEconomy();
12                 eco.setParams(new BarterParams());
13
14                 eco.addBarterStrategy(OriginalBarterStrategy.class, 0.5);
15                 eco.addBarterStrategy(GenerousBarterStrategy.class, 0.5);
16
17                 eco.setImprovementStrategy(CopyAndMutateImprovementStrategy.class);
18                 eco.setReplacementStrategy(DelphiReplacementStrategy.class);
19
20                 BarterEconomyGUI gui = new BarterEconomyGUI(eco);
21                 gui.create();
22         }
23 }
```

Figure B.3: Initialising the model with custom strategies.

The code between lines 12-18 is optional, default values exist for these fields in the `BarterEconomy` class. Using a custom `ImprovementStrategy` or `ReplacementStrategy` is as easy as specifying the new class for the economy (lines 17 and 18).

Initialising the model with custom barter strategies is a bit trickier. To be able to initialise agents with different barter strategies, there is an additional parameter for the `addBarterStrategy` method which determines the shares of different strategies. When the simulation is started, those shares are summed up and the number of

agents using a particular strategy is determined based on its share. In the above example, half of the agents use `OriginalBarterStrategy` and the other half uses `GenerousBarterStrategy`.

Finally, the GUI is brought up with the last two lines (20 and 21).

Appendix C

Delphi pseudo-random number generation

C.1 Flaws in the original Delphi V7 generator

Delphi V7 uses the most significant bits from the linear congruential generator (LCG)[1] $x_i = ax_{i-1} + 1 \mod 2^{32}, a = 134775813$. When clocking the generator for a number $[0, q)$, the state is updated and the output is $\lfloor \frac{qx_i}{2^{32}} \rfloor$.

Our biggest concern with the Delphi PRNG is that the period is very short; 2^{32}. Within one single simulation run it would be possible to exhaust the period. A run of the barter model uses at least $periods \times agentsPerGood \times goods^2$ random numbers, assuming every agent is successful in its first barter attempt. Another problem is that different runs, using different seeds, would use substreams that are not disjoint [LSCK02] with high probability.

C.1.1 Testing the generator

One comprehensive test suite for testing (pseudo) random number generators is L'Ecuyer and Simard's "TestU01", [LS07]. In the terminology of Marsaglia (and later L'Ecuyer), such a test suite is named a "battery".

To assess how closely the Delphi PRNG models a random stream of numbers, we run the batteries "SmallCrush" and "Crush" on various output combinations of the Delphi PRNG.

Since many tests in the TestU01 batteries use 32 bit numbers, we run the tests not only with the complete 32 bit outputs from the Delphi PRNG but also with outputs that are concatenations of several smaller words. In the original application, most calls to the PRNG are with small multipliers; the largest multiplier used is the number of agents per group. Table C.1 shows the concatenations we used.

[1] See [Knu97] for a comprehensive treatment of linear congruential generators.

By running the "TestU01" [LS07] test suite, we have found that the Delphi generator fails on most tests as can be seen in Tables C.3, C.4 and C.5. A failure consists of a goodness-of-fit that is either too far or too close to the expected distribution. Table C.2 contains a legend for the symbols used in the tests. The most spectacular failures are the ones containing an ε; the chance of a stream generating these by chance should be less than 10^{-300}.

For all tests conducted with TestU01, version 1.2.1 was used.

C.1.2 Practical significance of poor properties of the Delphi PRNG

Whether the poor test scores from TestU01 is a practical rather than theoretical issue for the original simulation we do not know. One of the algorithms that relies heavily on randomization is the permutation generation for good order and agent order. We have constructed two simple tests that are necessary (but not sufficient) to pass to determine whether permutations are at least identically distributed (i.d.), if not independently identically distributed (i.i.d).

Permutation index test

Any permutation of n unique elements can be indexed by an integer on $[0, n!)$. We call this a *permutation index*. We generate a sequence of k permutations, take their permutation indices and construct a histogram from them. Assuming they are i.i.d., the histogram counts should be Poisson-distributed with $\lambda = \frac{k}{n!}$. We then compute a goodness-of-fit score (by way of the one-sample KS-test) to get the probability p that the histogram really is Poisson-distributed.

When conducting tests for $n = \{3, 4, 5, 6, 7, 8, 9\}$, we did not find any indication that the histograms are *not* Poisson-distributed.

Element position test

For any permutation of $\{1, \ldots, n\}$ we measure the frequency of element m ending up in position m'. We generate k permutations and build a histogram ($n \times n$ matrix) from every $\langle m, m' \rangle$ pair. Any column and any row of the matrix should now be Poisson-distributed with $\lambda = \frac{k}{n}$. We then compute a goodness-of-fit for all rows and columns, obtaining $2n$ probabilities. Had these probabilities been independent, we could have done a KS-test for uniformity over $[0, 1]$. Since they are not, we can just say that the permutation generator passes the test unless we get values $p < \varepsilon$ or $p > 1 - \varepsilon$ for some sufficiently small epsilon, say $\varepsilon = 10^{-15}$.

From this test, we have not been able to conclude that the element positions are anomalous. The tests were conducted for $n = \{50, 100, 200\}$.

A stronger test (that we have not conducted) would be to generate many matrices, testing that any particular row/column probability is uniform over test runs.

"Generator"	Structure
D_{32}	All bits, 1×32
D_{16}	Concatenation of two 16 bit outputs, 2×16
D_8	Concatenation of four 8 bit outputs, 4×8
D_7	Concatenation of four 7 bit outputs and one 4 bit output, $4 \times 7 \| \| 4$
D_6	Concatenation of five 6 bit outputs and one 2 bit output, $5 \times 6 \| \| 2$

Table C.1: To generate 32-bit words to test, a concatenation of several outputs from the PRNG may be needed.

Symbol	p-value
$\downarrow$	$p < 10^{-15}$
$\Downarrow$	$p < 10^{-50}$
$\varepsilon^{\Downarrow}$	$p < 10^{-300}$
$\uparrow$	$p > 1 - 10^{-15}$
$\Uparrow$	$p > 1 - 10^{-50}$
$\varepsilon^{\Uparrow}$	$p > 1 - 10^{-300}$

Table C.2: A p-value close to 0 implies that the test gives rise to a (empirical) distribution *too far* from what should be expected from a random stream. A value close to 1 implies the test gives a distribution *too close* to the expected distribution. An empty slot means that the generator passed the test.

C.2 KISS generator in Delphi

Even though we have not been able to show that the Delphi PRNG leads to any anomalous results in the original application, in the light of the test results from TestU01 it seems prudent to replace it with a better generator. One generator that passes "Crush" as well as "BigCrush" ([LS07]) of TestU01 is "KISS99" [Mar99]. KISS99 is easy to implement and has a fairly long period, in our implementation $> 6.5 \times 10^{36}$.

We also introduced a rejection sampling algorithm (borrowed from `nextInt(int)` in `java.util.Random`). Thus the average number of calls to the PRNG is 2 in the worst case (for an interval of $[0, 2^{31} + 1)$) and exactly one call in the best case (for an interval of $[0, 2^n)$).

For our implementation of KISS99 in Delphi, see [EM08].

Test		p-value				
	name	D_{32}	D_{16}	D_8	D_7	D_6
1	BirthdaySpacings	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
2	Collision	$\uparrow$	$\uparrow$	$\uparrow$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
3	Gap	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
4	SimpPoker	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
5	CouponCollector	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
7	WeightDistrib	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
8	MatrixRank	$\varepsilon^{\Downarrow}$				
9	HammingIndep	$\varepsilon^{\Downarrow}$				
10	RandomWalk1 H	$\varepsilon^{\Downarrow}$				
10	RandomWalk1 M	$\varepsilon^{\Downarrow}$				
10	RandomWalk1 R	$\varepsilon^{\Downarrow}$				
10	RandomWalk1 C	$\varepsilon^{\Downarrow}$				

Table C.3: Test results from running TestU01's "SmallCrush" on the Delphi system generator.

Test		p-value				
	name	D_{32}	D_{16}	D_8	D_7	D_6
1	SerialOver, $t = 2$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
2	SerialOver, $t = 4$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
3	CollisionOver, $t = 2$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
4	CollisionOver, $t = 2$	$\varepsilon^{\Downarrow}$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
5	CollisionOver, $t = 4$	$\uparrow$	$\uparrow$	$\uparrow$		$\uparrow$
6	CollisionOver, $t = 4$	$\varepsilon^{\Downarrow}$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
7	CollisionOver, $t = 8$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
8	CollisionOver, $t = 8$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\uparrow$	$\uparrow$	$\uparrow$
9	CollisionOver, $t = 20$	$\uparrow$	$\Downarrow$	$\uparrow$	$\varepsilon^{\Uparrow}$	$\uparrow$
10	CollisionOver, $t = 20$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\uparrow$		$\downarrow$
11	BirthdaySpacings, $t = 2$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
12	BirthdaySpacings, $t = 3$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
13	BirthdaySpacings, $t = 4$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
14+15	BirthdaySpacings, $t = 7$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
16+17	BirthdaySpacings, $t = 8$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
18	ClosePairs NP, $t = 2$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\downarrow$	$\Downarrow$
18	ClosePairs mNP, $t = 2$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
18	ClosePairs mNP1, $t = 2$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
18	ClosePairs mNP2, $t = 2$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\Downarrow$	$\Downarrow$
18	ClosePairs NJumps, $t = 2$	$\uparrow$	$\uparrow$	$\uparrow$	$\Downarrow$	$\uparrow$
19	ClosePairs NP, $t = 3$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\downarrow$
19	ClosePairs mNP, $t = 3$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
19	ClosePairs mNP1, $t = 3$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
19	ClosePairs mNP2, $t = 3$					$\downarrow$
19	ClosePairs NJumps, $t = 3$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
19	ClosePairs mNP2S, $t = 3$					$\varepsilon^{\Downarrow}$
20	ClosePairs NP, $t = 7$	$\Downarrow$	$\Downarrow$	$\Downarrow$		
20	ClosePairs mNP, $t = 7$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
20	ClosePairs mNP1, $t = 7$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\Downarrow$
20	ClosePairs mNP2, $t = 7$				$\Downarrow$	$\Downarrow$
20	ClosePairs NJumps, $t = 7$	$\uparrow$	$\uparrow$	$\uparrow$	$\Downarrow$	
20	ClosePairs mNP2S, $t = 7$				$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$
21	ClosePairsBitMatch, $t = 2$	$\uparrow$	$\uparrow$	$\uparrow$		$\uparrow$
22	ClosePairsBitMatch, $t = 4$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
23	SimpPoker, $d = 16$			$\varepsilon^{\Downarrow}$		
24	SimpPoker, $d = 16$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$		
25	SimpPoker, $d = 64$			$\varepsilon^{\Downarrow}$		
26	SimpPoker, $d = 64$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$		
28	CouponCollector, $d = 4$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$		$\varepsilon^{\Downarrow}$
29	CouponCollector, $d = 16$			$\varepsilon^{\Downarrow}$		

Table C.4: Test results 1–29 from running TestU01's "Crush" on the Delphi system generator.

Test		p-value				
	name	D_{32}	D_{16}	D_8	D_7	D_6
30	CouponCollector, $d = 16$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$		$\varepsilon^{\Downarrow}$
32	Gap, $r = 27$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$		
34	Gap, $r = 22$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
36	Run of U01, $r = 15$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
37	Permutation, $r = 0$	$\uparrow$	$\uparrow$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Uparrow}$	$\uparrow$
38	Permutation, $r = 15$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$	$\uparrow$		$\uparrow$
39	CollisionPermut, $r = 0$		$\downarrow$	$\uparrow$		
40	CollisionPermut, $r = 15$	$\varepsilon^{\Downarrow}$			$\uparrow$	
41	MaxOft, $t = 5$		$\uparrow$	$\uparrow$		$\uparrow$
42	MaxOft, $t = 10$	$\uparrow$	$\uparrow$	$\uparrow$		$\uparrow$
43	MaxOft, $t = 20$		$\uparrow$	$\uparrow$		$\uparrow$
44	MaxOft, $t = 30$	$\uparrow$	$\uparrow$	$\uparrow$		$\uparrow$
49	AppearanceSpacings, $r = 0$	$\varepsilon^{\Downarrow}$				
50	AppearanceSpacings, $r = 20$	$\uparrow$				
52	WeightDistrib, $r = 8$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
53	WeightDistrib, $r = 16$	$\varepsilon^{\Downarrow}$				
54	WeightDistrib, $r = 24$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
57	MatrixRank, 60×60	$\varepsilon^{\Downarrow}$				
58+59	MatrixRank, 300×300	$\varepsilon^{\Downarrow}$				
60+61	MatrixRank, 1200×1200	$\varepsilon^{\Downarrow}$				
63	GCD, $r = 0$	$\varepsilon^{\Downarrow}$				
64	GCD, $r = 10$	$\varepsilon^{\Downarrow}$				
65+66	RandomWalk1 H, M, J, R, C ($L = 90$)	$\varepsilon^{\Downarrow}$				
67+68	RandomWalk1 H, M, J, R, C ($L = 1000$)	$\varepsilon^{\Downarrow}$				
69+70	RandomWalk1 H, M, J, R, C ($L = 10000$)	$\varepsilon^{\Downarrow}$				
72	LinearComp, $r = 29$	$\uparrow$				
74	Fourier3, $r = 0$	$\varepsilon^{\Downarrow}$				
75	Fourier3, $r = 20$	$\varepsilon^{\Downarrow}$	$\downarrow$			
77	LongestHeadRun, $r = 20$	$\varepsilon^{\Downarrow}$				
79	PeriodsInStrings, $r = 15$	$\uparrow$				
80	HammingWeight2, $r = 0$	$\uparrow$				
81	HammingWeight2, $r = 20$	$\uparrow$				
82	HammingCorr, $L = 30$	$\varepsilon^{\Downarrow}$				
83	HammingCorr, $L = 300$	$\uparrow$				
84	HammingCorr, $L = 1200$	$\uparrow$				
85	HammingIndep, $L = 30$	$\varepsilon^{\Downarrow}$				
86	HammingIndep, $L = 30$	$\varepsilon^{\Downarrow}$	$\varepsilon^{\Downarrow}$			
87+88	HammingIndep, $L = 300$	$\varepsilon^{\Downarrow}$				
89+90	HammingIndep, $L = 1200$	$\varepsilon^{\Downarrow}$				
92	Run of bits, $r = 20$	$\varepsilon^{\Downarrow}$				
95	AutoCor, $d = 30$	$\uparrow$				
96	AutoCor, $d = 10$	$\uparrow$				

Table C.5: Test results 30–96 from running TestU01's "Crush" on the Delphi system generator.

Appendix D

Comparisons of $\mathcal{G}_k$ and $\mathcal{J}$ for some parameter sets

Each of the figures from D.1 to D.4 compare the results of running $\mathcal{G}_k$ and $\mathcal{J}$ with two different parameter sets. Thus a total of eight (4 times 2) distinct parameter sets are compared in these figures.

Fig. D.1 starts by comparing the behaviour of the two programs with the *goods* $= 7$ and $maxTries = 300$. The results are captured in plots D.1.a, D.1.b, D.1.c and D.1.d. Plots D.1.a and D.1.b show the *mean relative prices* of the goods over 300000 periods for both versions. The *mean relative price* of a good at some point shows how much the price of the good differs from the equilibrium price on average (among all the agents). So, a value of -30 at some point indicates that the mean price is 30% lower than the equilibrium price at that point. The third plot, D.1.c, is to show the price difference of means between the first two plots ($\mathcal{J}$-$\mathcal{G}_k$) and the last plot, D.1.d, shows the result of the rank-sum test over time.

The same structure repeats for the plots D.1.e, D.1.f, D.1.g and D.1.h, but for these subplots, *goods* $= 5$. Figures D.2, D.3 and D.4 follow the same pattern.

One thing to note about the rank-sum test is that sometimes it seems to contradict the price difference plot. One example is in plots D.2.g and D.2.h, where the price difference is close to zero towards the end for both goods. Nevertheless, the rank-sum test suggests that the prices of *good 2* are different between the two programs. This can be explained by the low variance towards the end—although the prices seem to be close enough to the naked eye, they are not the same and together with a low variance the rank-sum test gives a very small probability that they have *the exact same median*.

For the rest of the 12 tested parameter sets, we only include the results of the rank-sum tests (Fig. D.5). The plots D.5.f, D.5.g, D.5.i and D.5.l more or less indicate a failure at least in the later periods. This is also in correspondence with the KS-tests presented in Table 6.3.

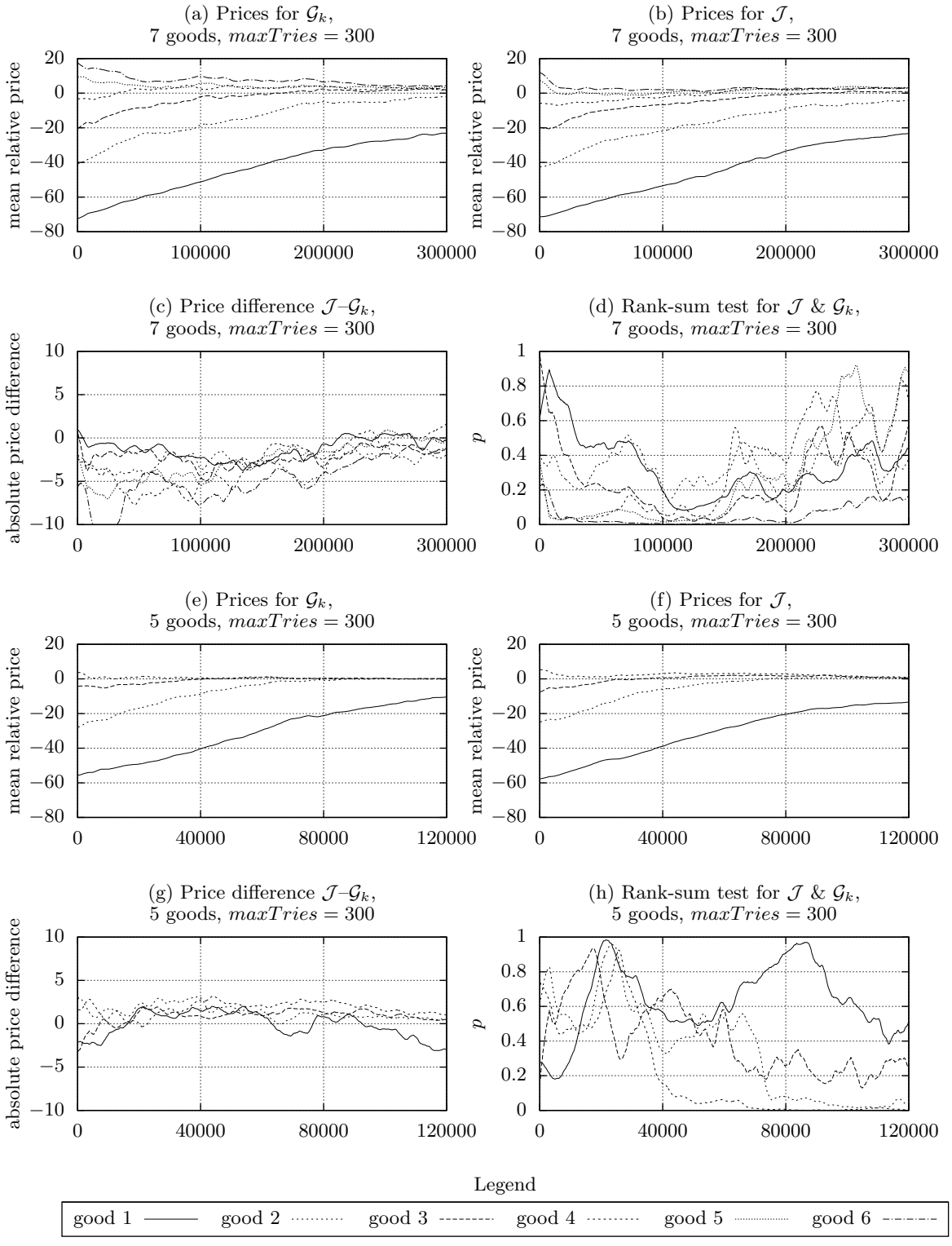


Figure D.1: Our implementation $\mathcal{J}$ compared to the original implementation $\mathcal{G}_k$; the number of goods set to 5 and 7.

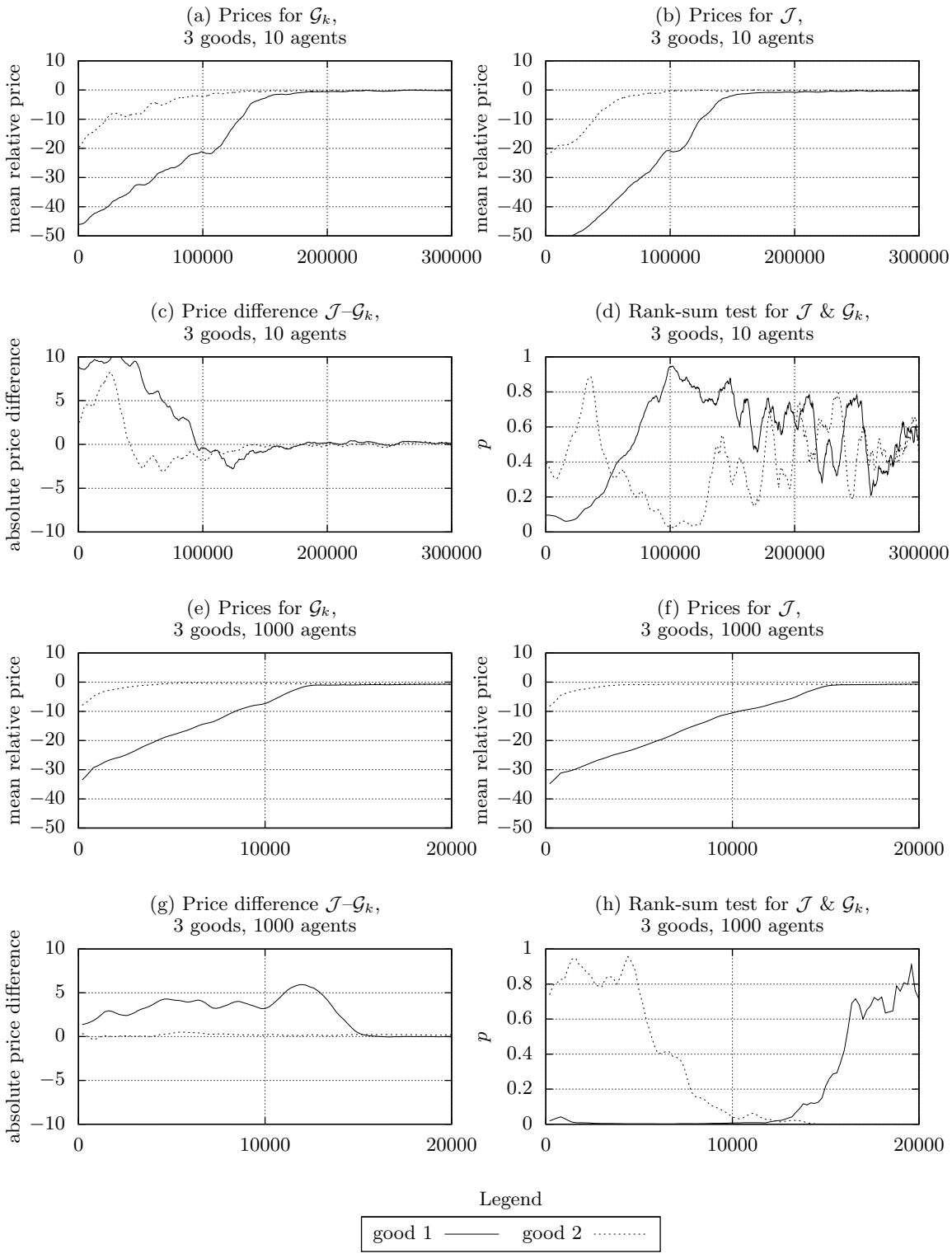


Figure D.2: Our implementation $\mathcal{J}$ compared to the original implementation $\mathcal{G}_k$; the number of agents per good set to 10 and 1000.

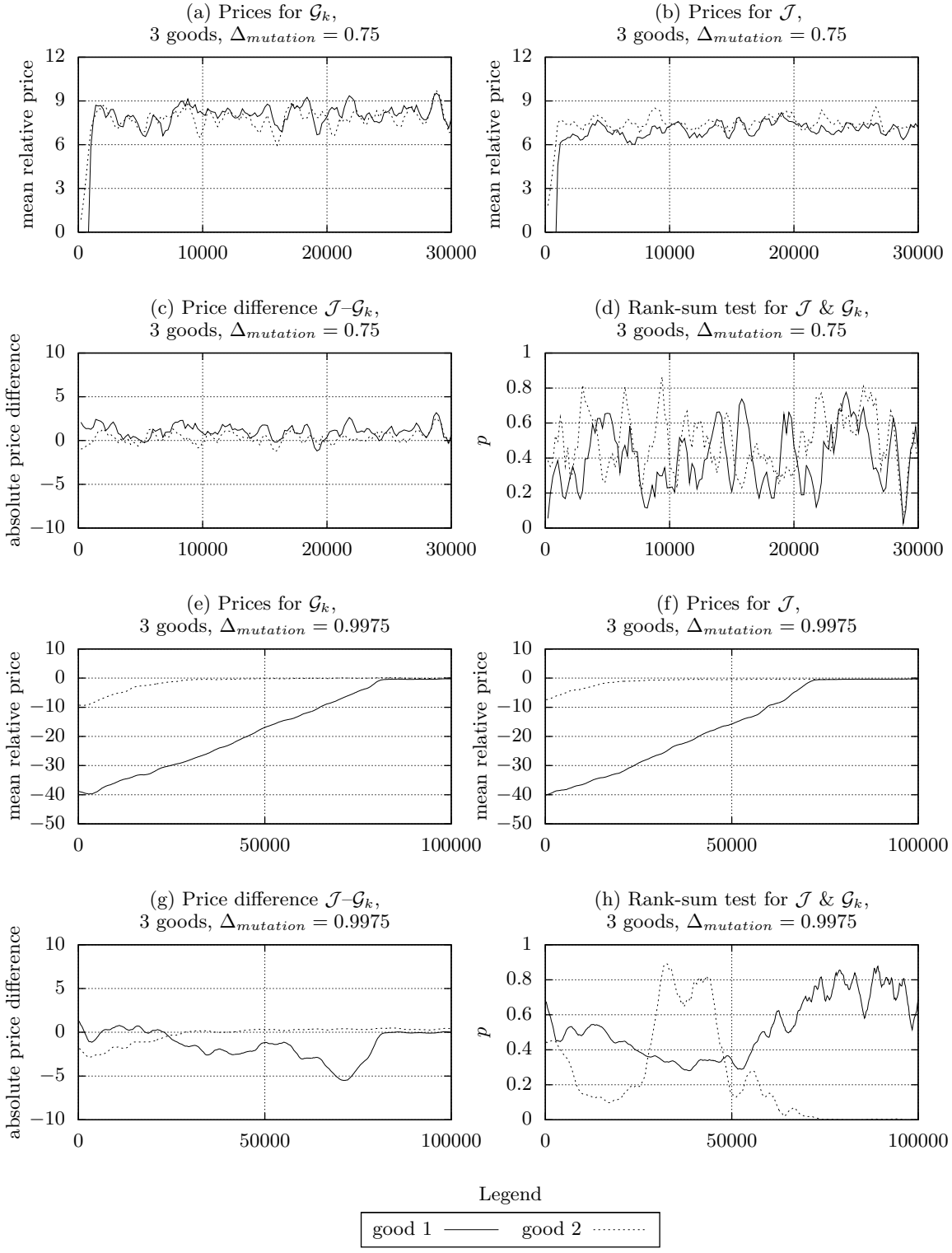


Figure D.3: Our implementation $\mathcal{J}$ compared to the original implementation $\mathcal{G}_k$; $\Delta_{mutation}$ set to 0.75 and 0.9975.

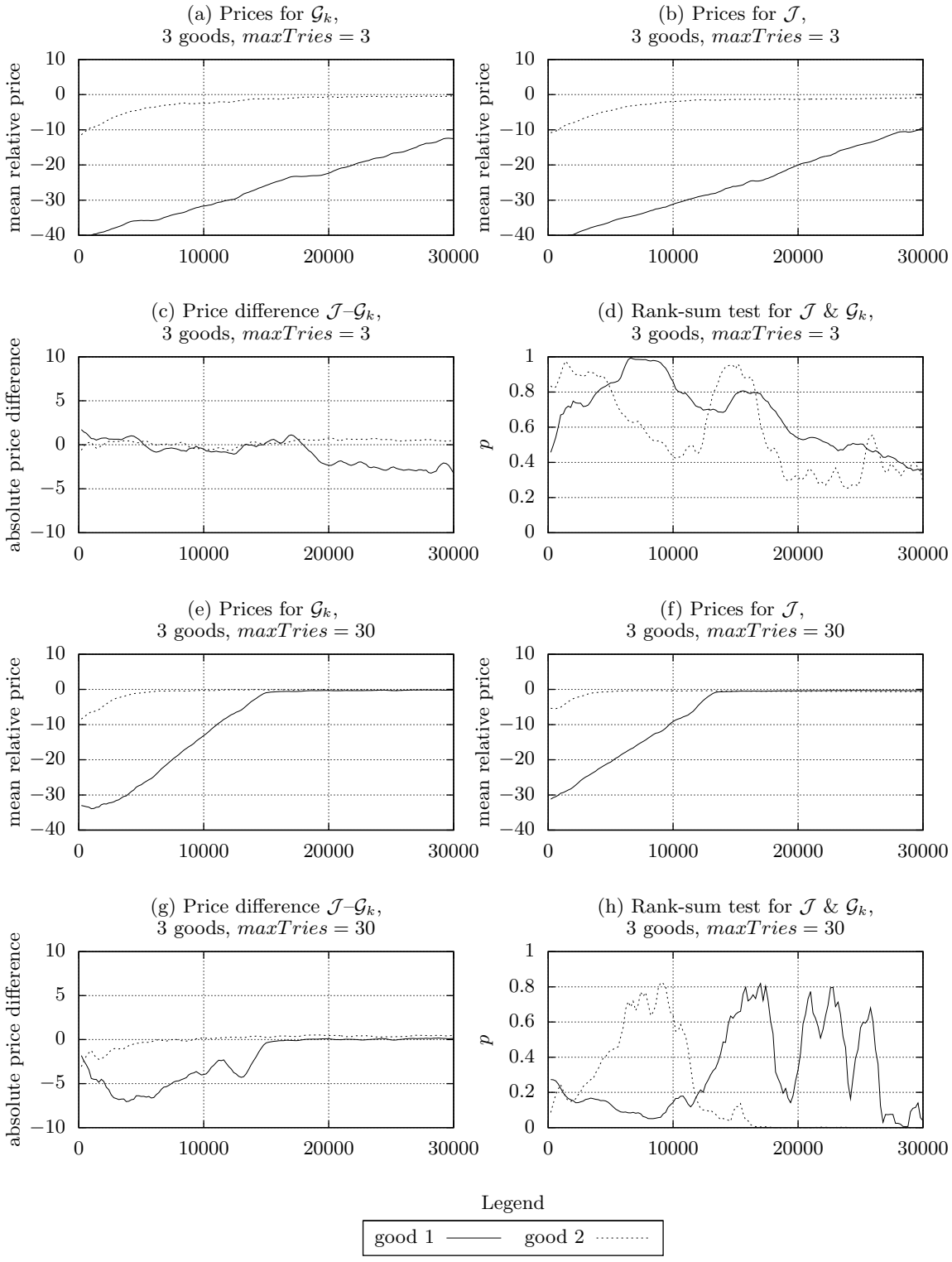


Figure D.4: Our implementation $\mathcal{J}$ compared to the original implementation $\mathcal{G}_k$; the maximum number of trade attempts set to 3 and 30.

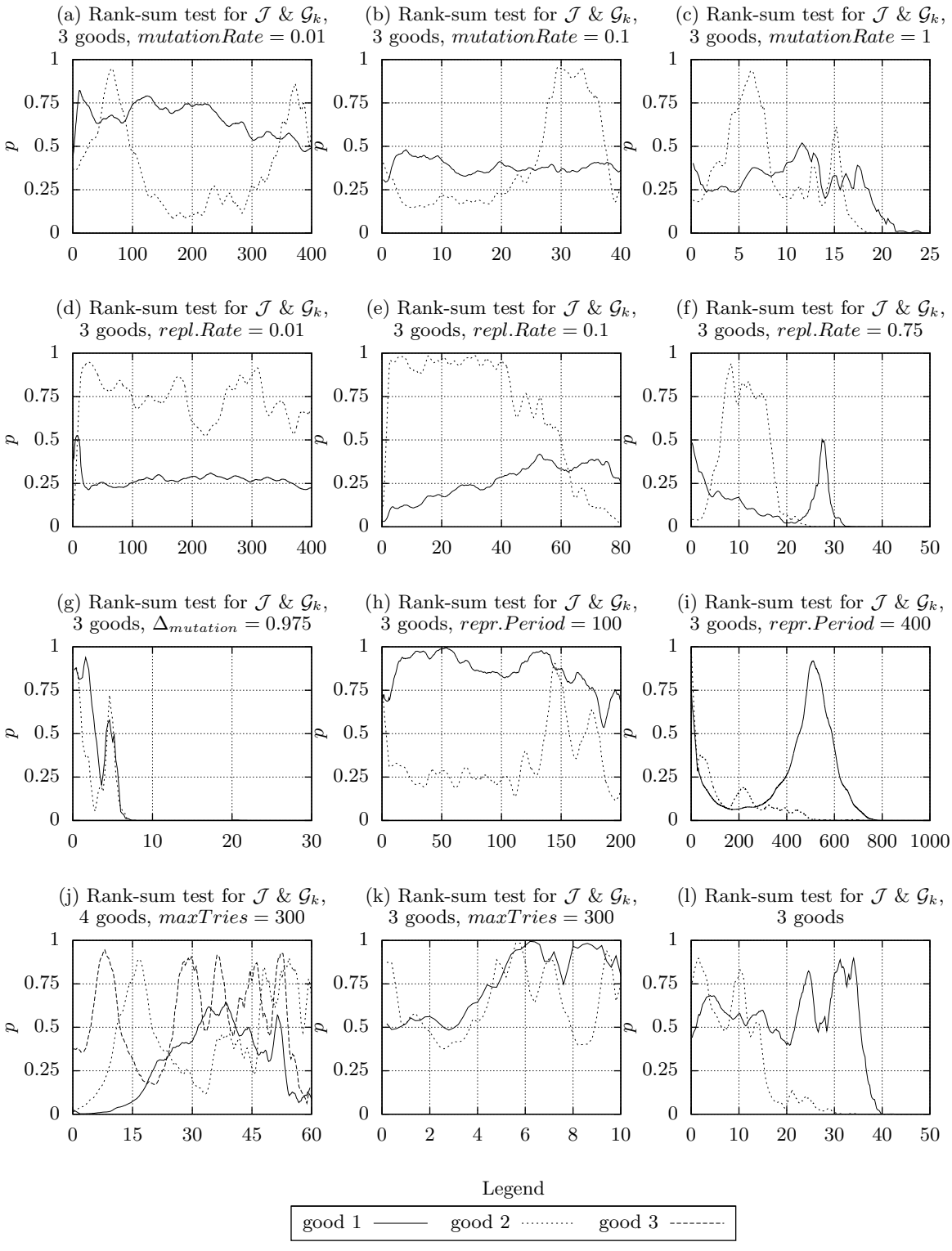


Figure D.5: Our implementation $\mathcal{J}$ compared to the original implementation $\mathcal{G}_k$ showing just the results of rank-sum test; various parameters. All periods in multiples of 1000.