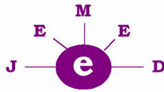
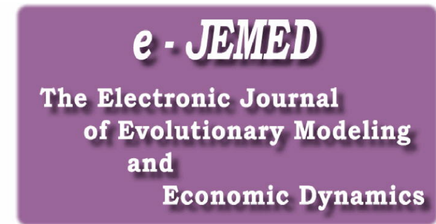


http://www.e-jemed.org/

ISSN : 1298-0137



Article number: 1003

Please cite this article as following:

Valente, M., Andersen, E.S., 2002, "A hands-on approach to evolutionary simulation: Nelson–Winter models in the Laboratory for Simulation Development", *The Electronic Journal of Evolutionary Modeling and Economic Dynamics*, n° 1003, http://www.e-jemed.org/1003/index.php

A hands-on approach to evolutionary simulation: Nelson–Winter models in the Laboratory for Simulation Development

Marco Valente

University of L'Aquila

Esben Sloth Andersen

DRUID and Department of Business Studies, University of Aalborg

Abstract

The topic of this paper is evolutionary-economic models and how they are implemented in a new, effective system for programming and simulating such models. The evolutionary-economics simulation models are exemplified by the Nelson–Winter family of models of Schumpeterian competition in an industry (or an economy). To abbreviate we call such models NelWin models. The new system for the programming and simulation of such models is called the Laboratory for Simulation Development—abbreviated as Lsd.

The paper is meant to allow readers to use the Lsd version of a basic NelWin model: observe the model content, run the simulation, interpret the results, modify the parameterisation, etc. Since the paper deals with the implementation of a fairly complex set of models in a fairly complex programming and simulation system, it does not contain full documentation of NelWin and Lsd. Instead we hope to give the reader a first introduction to NelWin and Lsd and inspire a further exploration of them. For the further exploration the reader will have to consult the increasing literature on evolutionary-economic modelling and NelWin, the literature on programming and simulation, and the extensive documentation on Lsd are available from the Lsd web site: `www.business.auc.dk/lsd/`.

The paper is constructed as follows:

In section 1 there is an overview over the goals and components of NelWin and the Lsd system. This includes a short discussion of the difficulties of evolutionary simulation, which defines the goals of the Lsd system development project. Furthermore short information is given on the components of the Lsd system and their installation in MS Windows and Unix/Linux.

Section 2 gives a quick introduction by immediately starting and running NelWin in the Lsd system. This includes a description of the main Lsd functionalities and windows.

In section 3 the details of running NelWin models—and other Lsd models—are given. This includes the Lsd facilities for inspecting the models and their initialisation settings, how to run Lsd simulations and study and print the results. Furthermore there is a discussion of how to set up Lsd simulation experiments.

Section 4 turns to the issue of an easy way into the programming of evolutionary models by incrementally changing the models in e.g. the NelWin tradition. This includes a discussion of Lsd's C++ based equation language, the methods of compiling Lsd models, and experiments that extend the basic NelWin model.

Section 5 contains conclusions and perspectives that relate to the way the Lsd system can be used to enhance the development of the Nelson–Winter tradition of evolutionary modelling and, probably, other evolutionary modelling traditions.

The appendix describes the main features of the language (C++ and Lsd) used by developers to write Lsd models.

Keywords: evolutionary modelling, simulation of industrial dynamics, Nelson and Winter, tools for simulation

JEL: C60, C88, L00, L10

Copyright: Marco Valente and Esben Sloth Andersen, 2002

Contents

1. INTRODUCTION TO NELWIN AND LSD 1

1.1 NelWin and the need for revitalising evolutionary modelling........ 1
1.2 The goals of the Laboratory for Simulation Development 3
1.3 The Lsd system and its installation........ 5
1.4 The elements of the paper........ 9

2 STARTING THE LSD SYSTEM WITH NELWIN 10

2.1 A hands-on approach 10
2.2 The Lsd Browser........ 13
2.3 The Lsd Model Structure Window 15
2.4 A simulation and its Run Time Plot Window........ 16
2.5 The Data Analysis Window: Example........ 18
2.6 The Data Analysis Window: Possibilities 20
2.7 Closing the session 21

3 ANALYSING NELWIN WITH LSD 22

3.1 A closer look at NelWin 22
3.2 Lsd equations........ 22
3.3 HTML reports on equations and initial values 23
3.4 Summarising the information on the NelWin model........ 25
3.5 Simulation Settings........ 29
3.6 Initial Data 31
3.7 Planning a set of simulation experiments 35
3.8 More on the Number of instances of Objects 35
3.9 Log Window........ 37
3.10 Options for saving, debugging and runtime plots........ 39

4 MODIFYING AND BUILDING LSD MODELS........ 39

4.1 Becoming a Lsd programmer 39
4.2 Specifying modifications of the NelWin model 41
4.3 The Lsd Model Manager as a core tool 42
4.4 Lsd equations........ 43
4.5 Compiling the model with a modified equation from LMM 49
4.6 Compiling the model with a new equation 50
4.7 Running an experiment with the a Lsd model 53

5 CONCLUSIONS AND PERSPECTIVES 55

REFERENCES 56

APPENDIX A : LSD GRAMMAR FOR EQUATIONS........ 58

A1 Lsd equations and C++........ 58
A2 Equations in Lsd models........ 59
A3 Local C++ variables in Lsd equations 60
A4 Lsd global C++ Variable: **p->** and **c->** 61
A5 List of Lsd functions........ 61

1. INTRODUCTION TO NELWIN AND LSD

1.1 NelWin and the need for revitalising evolutionary modelling

Because of its pioneering status and its relative simplicity, the model of Schumpeterian competition developed by Nelson and Winter (1982a, Chs. 12–14) has obtained a prominent role in defining what evolutionary economics is about. From this model we know that economic evolution can be depicted as a process in which firms follow rules or procedures that can occasionally be mutated or adapted. We also know that an important example of economic evolution takes place within an industry (or a one-sector economy) where new process techniques are introduced and imitated. These and other aspects of the Nelson–Winter model have to some extent defined a trajectory of further research on the conditions of R&D as a determinant of industrial concentration, dynamic competition in alternative technological regimes, the relationship between innovators and imitators, etc.[1] A variant of the Nelson–Winter model (1982a, Ch. 9) has also had some influence in promoting evolutionary growth theory.[2]

The use of computer simulation has an important but too little understood function in the evolution of this Nelson–Winter tradition of evolutionary economics. Its main function is to facilitate the integration of the different mechanisms of the evolutionary process, which are normally considered as belonging to quite different areas of investigation. These mechanisms are the processes of transmission, variety creation, and selection. The integration of these mechanisms into an evolutionary process presupposes two opposing capabilities: ability to cope with a wide diversity of mechanisms, and ability to remove the details and integrate the mechanisms into an initially crude conception of an evolutionary process. The computer helps to organise the synthesis to the very last steps, since 'the simulation format does impose its own constructive discipline in the modelling of dynamic systems: the program must contain a complete specification of how the system state at $t + 1$ depends on that at t and on exogenous factors, or it will not run.' (Nelson and Winter 1982a, 208 f.) Computer models also reveal complexities of evolutionary models that explain the weaknesses of the informal approaches to evolutionary processes like the contributions of Schumpeter: these processes are normally so complex that it is nearly impossible to master them intellectually by means of verbal modes of thinking.

Although we to some extent may characterise much of modern evolutionary economics by the formula 'Schumpeter plus a computer', computer simulations have not been an easy tool. The problem has basically been that the high degree of multi-skilling needed for developing and implementing

[1] See e.g. Silverberg, Dosi, and Orsenigo (1988), Chiaromonte and Dosi (1993), Silverberg and Verspagen (1994), Kwasnicki (1996), and Malerba et al. (1999).
[2] See the critical surveys in Silverberg and Verspagen (1998a) and (1998b).

evolutionary simulation models has created quite high barriers of entry to the field. One way of overcoming the problem has been to create teams of researchers, in which some do the modelling and others the computer implementations of the models. For instance, Nelson and Winter developed the early models while it was largely PhD students who implemented their models in FORTRAN and ran them on mainframe computers. While such a division of labour gave quick results, it also hindered the cumulative improvement of the models, and led to strong tendencies of lock-in with respect to model specifications and even parameter settings.[3] Another problem was and still is that there are extremely few researchers who actually are reengineering and testing even well known simulation models. Since it is hard and time consuming to program and debug simulation models, this situation is also highly problematic: there are very few to criticise sloppy programming and obvious errors and, even more important, to give praise to high-quality programming. This is a reflection of the lack of a research community dealing collaboratively with simulation models, so that can exchange concrete ideas about models and their implementations. Given the lack of such a community, each researcher tends to start the programming tasks from scratch and move into idiosyncratic model specifications.

The problematic situation is not easy to change. Editors of scientific journals that publish papers with simulation results could ask for public access to the underlying code, but such editors might get very few papers for one or more of the following reasons: programs are written in the sloppy style, authors fear that they loose the results of their heavy investments in code generation, authors think that no one would care to read their programs, and authors think that their programs might not be up to date since they are only guessing what others are doing. Researchers engaged in university teaching might consider giving courses in evolutionary programming and simulation, but they fear they will get very few students because of the common knowledge about the difficulties in the field, and even if they get students, there might emerge some dissatisfaction because of the lack of a stock of interesting programs and exercises in the public domain. However, the situation is changing—see e.g. Conte, Hegselmann, and Terna (1997) on tools and research areas, and Gilbert and Troitzsch (1999) on teaching materials.[4] Although the research underlying the present paper can be shown to support this general improvement of the situation for simulation work,[5] we shall however concentrate on its relationship to the Nelson–Winter tradition—both for its own sake and as an example with implications for other traditions.

---

[3] These problems can easily be seen by studying Nelson and Winter's own family of models in Nelson and Winter (1974, 1977, 1978, 1982b), Nelson, Winter and Schuette (1976), and Winter (1984).

[4] In another paper we shall compare Lsd to other attempts to create tools for simulation work in the social sciences, e.g. in relation to Complex Adaptive Systems and Agent-Based Computational Economics, see e.g. Conte, Hegselmann, and Terna (1997) and the Tesfatsion site: `www.econ.iastate.edu/tesfatsi/sylalife.htm`.

[5] In the present paper we shall not comment on other contributions to promote simulation work. These contributions range from the work around the DYNAMO/Stella simulation environments of the Systems Dynamics Group at MIT (see e.g. Sterman 2000) to the many contributions relating to the Swarm environments of the Santa Fe Institute (see e.g. Epstein and Axtell 1996). We hope later to have a chance to comment upon the relationships between Lsd and some of these core contributions.

In relation to the Nelson–Winter tradition there is, luckily, a kind of standard model that students and researchers can start with. However, while the overall feature of this model of Schumpeterian competition is fairly well known, the details of the implementation and simulation has been very little discussed. But it is the common knowledge of these details that makes the standard model important. When these specifications become wide-spread and well-known, they can be used to facilitate communication even on other models, and they can by used for the testing of simulation systems. We think that a stylized version of the Nelson–Winter models can serve such a function. This standard NelWin model was first proposed by Andersen (1996, Ch. 4) and implemented in programs using the symbolic mathematics package Maple.[6] Later the standard model was slightly modified and implemented in the Laboratory for Simulation Development by Marco Valente, Department of Business Studies, Aalborg University (now University of L'Aquila) and Murat Yildizoglu, BETA, University of Strasbourg (now University of Bordeaux). This version of NelWin is used in the present paper.[7] The idea is simply to use a fairly wide-known evolutionary model for discussing the Lsd system, not to canonise a particular version of NelWin. Therefore, we shall emphasise that other NelWin standards have been suggested (e.g. Andersen 2001b). Furthermore, it should be noticed that that the NelWin model has been used as an inspiration for developing new evolutionary models even by the authors of the present paper—see e.g. Andersen (1999, 2001a, 2001b) and Valente (1999a, 1999b, 1999c).

1.2 The goals of the Laboratory for Simulation Development

In this paper we show how NelWin can be implemented, studied and extended as implemented in a new programming system called Laboratory for Simulation Development (Lsd). This system was developed and is maintained by Marco Valente for IIASA, Austria (with researchers like Nelson, Winter, Dosi, Silverberg, et al.), and for DRUID, Aalborg University, Denmark (with Andersen). The results of the Lsd project have most extensively been presented in Valente (1999b). The underlying goal of the project[8] is to facilitate the development and use of computer simulations in economics—especially evolutionary economics. The main background of the development of the Lsd system is the fact that there are quite high barriers to entry to the field of micro-based simulation in economics. At the same time the outcomes of simulation work often do not live up to the expectations and the

[6] See Andersen (1996, Appendix) and Andersen's old Maple models (now being moved to Lsd) at `www.business.auc.dk/evolution/esa`.

[7] In the present version of the paper we use the revised version of NelWin that was implemented by Yildozoglu in October 2001.

[8] Lsd is an on-going project, which had its latest broad review (of Lsd 4.0) during a meeting on the promotion of evolutionary economic simulation held at BETA, University of Strasbourg on 18 October 2001. Among the 15 participants were several persons mentioned in the present paper, including Dosi, Nelson, Silverberg, Verspagen, Winter, and Yildozoglu. At the meeting a fairly large number of improvements were suggested, and not all of them have been implemented in Lsd 4.1 (the version used for the present paper). Although major new features can be expected, they will be implemented with an emphasis on backward compatibility.

invested efforts. Lsd may be described as simulation language that tries to change this situation in two ways.

First, Lsd allows users with no training in computer programming to understand and use simulation programs, even programs implementing complex models. Such users can access an evolutionary model developed in Lsd through a ready-and-easy-to-use stand-alone program. Users access the model through a fairly intuitive graphical interface[9] providing the possibility to fully explore the model content, test the simulation runs with given configurations, and try new configurations with the maximum flexibility. Users can also fairly easily begin to reprogram models and recompile models— provided that they limit themselves to simple changes of equations. Thus even ordinary users with MS Windows systems implicitly start to apply a good deal of the tools that are delivered together with all versions of Lsd, and these are actually interfaces to the tools that Unix/Linux users have grown accustomed to (see e.g. Welsh et al. 1999).

Second, Lsd allows modellers to build efficient and powerful simulation models, fully exploiting the computational power of modern computers. To obtain this goal, Lsd gives an interface for using the powerful programming language C++, which provides very fast code and gives basically no limits to the computational content of models.[10] To make the programming and testing work efficient, Lsd offers an interface with a variety of tools for the implementation of the final program. Thus programmers can concentrate their work exclusively on writing the code for the equations of the models, while the system automatically generates every interface and other technical requirements for the actual working of the computer program implementing the simulation model. These tools mean that writing a Lsd model does not require to write a traditional computer program, but only to express the equations of the model according to the relevant and fairly simple parts of the C++ grammar. The Lsd system does the rest, including the provision of a standard interface for the users of the model. Furthermore, a simulation model implemented in Lsd can be transferred to many different computer platforms— as may be the ones used by programmers, modellers and students.

In the present paper we shall give a hands-on introduction to the Lsd system in order to demonstrate how these goals have been obtained. More specifically, we shall demonstrate how some of the barriers to entry to the field of evolutionary simulation have been radically diminished, so that even MA students in a couple of hours can analyse a simple model like NelWin, begin to modify it and study the results of the recompiled model. Since both the Lsd system and evolutionary models are fairly

[9] The graphical interface is build by means of the Tk toolkit and the related parts of the Tcl language, as described by Ousterhout (1994) and a lot of subsequent literature.

[10] C++ has been designed for the most difficult tasks of large-scale programming (including operation systems), so it has a very strong support for modularity and other aspects of advanced programming, and it is rapidly becoming the industry standard. Furthermore, the C++ language grew out of experiences with the classical simulation-oriented language (SIMULA), so it has also strong support for simulation-oriented concepts (cf. Stroustrup 1994, 2000).

complex entities, we cannot give a full manual. We are, furthermore, dealing with moving targets since both Lsd and many evolutionary models are ‘open source’ software projects (see Andersen and Valente 1999).[11] Therefore, it is a good idea to keep updated by e.g. visiting the Lsd web site: `www.business.auc.dk/lsd/`. The version covered by the present paper is Lsd 4.1 (released in November 2001).

1.3 The Lsd system and its installation

1.3.1 Introduction to the use and programming of Lsd models

Although it is possible to use a Lsd model without thinking of the underlying software, it is useful to have a minimum knowledge in the area. In short, Lsd is a sort of computer language: the modeller defines how the computations should be done in terms of difference equations, which are incorporated in a single model program together with the system Lsd code for the actual management of simulation runs. That is, a model program contains a part of code which is model specific (the equations), and another one which is common for all the Lsd models (interfaces, simulation engine, debugger, etc.). Once the model program is completed, the user has available a large set of Lsd utilities to control the simulation runs and exploit the results (statistics, graphs, export data). These allow to control the model parameters and simulation properties setting, determine the data to save during a simulation, analysing saved data, etc. There are, therefore, two levels of using Lsd models: once concern the ‘hard’ computational content of the model (i.e. equations’ coding) and another the ‘soft’ data management of the input and output of the model. The ‘soft’ level is dealt with by Lsd model’s own interfaces, so that, for example, a user can load a given parameterisation for a model, modify it and save for later use, always with the same Lsd model program. Instead, the ‘hard’ content of the model (how the variables are computed) is C++ compiled code, and therefore cannot be changed without reconstructing the whole model program.[12]

The use of C++ is motivated by several reasons. The main ones are:

- efficiency: Lsd models are virtually unconstrained in dimensions and extremely fast;
- flexibility: Lsd models can express any computational algorithm;
- portability: the same model can run on any Unix and Windows machine.[13]

---

[11] The perspectives and practices of open source software development are e.g. described by DiBona et al. (1999) and Fogel (1999).

[12] Actually, Lsd modellers do not build a C++ program, but simply express the equations in C++ terms, facilitated by many Lsd specific functions to manage the Lsd model data. Therefore, the knowledge of C++ requested to write a C++ model is extremely small. More on this later...

[13] On Macintosh systems Lsd runs in Windows emulators.

As long as C++ is used for delivering pre-compiled evolutionary models to Lsd users, these characteristics are a clear-cut advantage. But as soon as the user of an Lsd model wants to change that model, there are problems—especially for many MS Windows users who have no access to the programs (compilers) that transform the modified model into an executable application. The modification work can, furthermore, be made much easier if a well-defined environment can be expected for all or most Lsd users and would-be developers. For this reason the distribution of Lsd includes a program, Lsd Model Manager (LMM), that provides an efficient programming environment for Lsd models. From LMM is possible to (re)write the equations' code, compile a model, 'hard' debug it (via the GNU gdb debugger). In general, LMM allows users to scan the models available and choose which one to run, taking care of the possible necessity to compile the model.

If the desired simulation session does not need to modify the equations of the model, users can use LMM just to run the Lsd model program, whose interfaces permit to change all the numerical aspects of a model. Otherwise, back in LMM, users can access the equations' code through the C++ editor embedded in LMM, modify the code, and run again the modified version of the simulation[14]. All these operations are offered by simple menu entries in the LMM window, so users can ignore the actual mechanisms triggered by LMM to build the Lsd models. It is worth to stress the difference between LMM and Lsd models. LMM manage the Lsd models by providing users with tools for the construction of Lsd models, that is, the computational content. On the other hand, the individual Lsd models are endowed with facilities to manage the numerical content of the model.

1.3.2 Lsd distribution contents

Even though it is not strictly necessary for using a Lsd model, in the following is briefly reported the content of a full-blown Lsd distribution, i.e., the files used and their overall meaning. The distribution includes all the necessary code to build a simulation model, a full documentation of the whole system, and several example models ready to run. One of them, the Nelson–Winter model (Nelson and Winter, 1982a, Ch.12), will be discussed here in detail. The other models can be explored using the accompanying documentation and use the same interfaces here described for the Nelson–Winter model.

The models are stored in individual directories, or folders, containing their specific files. Users do not need (and should avoid) to access these files by other means than the system interfaces. Whatever operation one may need to do on a model, from the compilation process to analyse data produced in a

[14] LMM always run 'make' before the actual run of a Lsd model program, ensuring that the latest changes to the equations file are included in the Lsd program. Moreover, LMM shows any error message possibly issued by the 'make' command during compilation.

previous simulation, is done always using graphical interfaces with the possibility to access the relevant help page. The data representing a model are the following:

- The equations' code; these are chunks of C++ code to be executed in order to compute the value of a variable. LMM allows users to show the file containing the equations' code in an editor designed to write the specific C++ and Lsd code of a model's equations.
- Model structure and configuration. The first operation when running an existing Lsd model is to load a model configuration (including the model structure). This file (extension '**`.lsd`**') lists the Objects, Variables and Parameters of the model, and contains all the numerical values to start a simulation run. More configuration files are created to store different initialisations for the same model.
- Model documentation. Each model is assigned by the modeller a short 'description' file that summarise briefly the content of the model (this file is shown in LMM when a model is firstly selected). Moreover, the Lsd model program allows users to access two files, created automatically: one lists in a hypertext all the elements of the model, including the equations code, and the other lists the elements in tables to be possibly filled with further information by the modeller (e.g. range of values to initialise a parameter).
- Model and System compilation options (only for expert users or for fixing installation errors). LMM creates automatically two files storing the information on how to compile a Lsd model program. In case of errors (e.g. non-standard location of required libraries), or for specific requests (e.g. generation of optimised code), LMM allows users to set these options.
- Simulation results, if produced by the user. Lsd allows saving data from a simulation run in several formats (different types of text files), in order either to review them in Lsd or to export in an external program (e.g. SAS, SPSS, GNUPLOT). Moreover, Lsd can produce encapsulated postscript files with the pictures produced in the Analysis of Results module.

Besides the set of example models, Lsd is distributed with its own source code, i.e., all the technical functions used to actually run a simulation and shared by any Lsd model program. These files are listed in the directory '**`\src`**' under the main Lsd directory. LMM, and its source file, is stored in the main Lsd directory.

The MS Windows distribution assumes that, as most Windows machines, there are no compilers or the standard libraries normally installed in Unix systems. Therefore, a directory ('**`\gnu`**', in the main Lsd directory) contains all the software necessary to compile and run a Lsd model (e.g. compiler, standard libraries, C++ header files, Tcl/Tk library). This software is taken from the CYGWIN distribution by Cygnus (`www.cygnus.com/cygwin`) which offers a port of the main Gnu software for Windows,

distributed under the GNU General Public License.

In the following are briefly described the procedures to install the Windows and Unix versions of Lsd[15].

1.3.3 Windows installation

The Lsd installation for MS Windows has been tested for any version currently available, including Windows XP. It is a single self-installing program, where users need only to select the directory under which the system must be installed. No other installation or configuration is required. The Lsd distribution can be downloaded as a single self-installing file (approximately 8 MB) from `www.business.auc.dk/lsd`.

After the download run the installation file following the instructions (which basically requires users to specify the name of the installing directory). Once the installation is completed the Lsd Model Manager (LMM) can be run starting from the menu Start in the group of buttons called 'Lsd Model Manager', or by running the file '**`run.bat`**' in the main Lsd directory. When LMM starts the user can choose whether to select an existing model, to create a new one, or just use LMM as an editor for text files. When a model is selected, it can be started by choosing the menu item **`Model/Run`**. The Lsd model program will automatically run, and it is ready to load a configuration and run a simulation.

1.3.4 Unix installation and requirements

The Unix version is distributed as source code, in a zipped compressed file. Unpack the file with the command (the Lsd version number may vary):

```
# unzip -U -a Lsd41unix.zip
```

After that it is necessary to move into the newly created directory (e.g. `# cd Lsd4.1`) and compile LMM with the command:

```
# gcc modman.cpp -ltcl8.3 -ltk8.3 -o lmm
```

Possibly, the user needs to replace the command with different version numbers of the Tcl/Tk libraries (e.g. `-ltcl8.2 -ltk8.2`). To run LMM use the command:

```
# ./lmm
```

[15] Here is referred to the most recent distribution, 4.1. Models developed with previous distributions of Lsd are all compatible, and can be recompiled with the latest version. To add a model to an installed version of the Lsd system just add a new directory under the main Lsd directory and place in it all the model files.

The system needs to have installed the following packages (normally already present on most systems):

- Tcl/Tk libraries, up to the version 8.3,
- GNU compiler and standard libraries,
- GNUPLOT (not strictly necessary, used to produce scatter plots, etc.).
- Netscape (not strictly necessary, used to show the manual pages)

Once LMM is running, compile the example models through its interfaces (menu entry **`Model/Run`**). This will activate the 'make' for the makefile's for the example models distributed and will run the Lsd model program. In case of errors the problem likely depends on the different version of the Tcl/Tk library version installed on your system. Use the 'System Compilation Options' to fix this problem (once solved for a single model, the solution is automatically applied to any other model).

1.4 The elements of the paper

The paper can be studied in three ways. First, it can be read as an account of Lsd and its implementation of NelWin and similar models. Second, it can be used as a manual for exploring the Lsd system—given that the reader has access to an installed version of the Lsd system. Third, it can be read as an account for a new style of developing and exchanging evolutionary models that solves several of the problems mentioned in section 1.1. The best way of exploring the paper is probably to use all the three approaches.

The paper is constructed as follows: Section 2 gives a quick introduction by immediately starting and running NelWin in the Lsd system. This includes a description of the main Lsd functionalities and windows. In section 3 the details of running NelWin models—and other Lsd models—are given. This includes the Lsd facilities for inspecting the models and their initalisation settings, how to use optional Lsd simulation settings (e.g. multiple runs) and how to study and print the results (e.g. series statistics and graphical plots). Furthermore there is a discussion of how to set up Lsd simulation experiments. Section 4 turns to the issue of an easy way into the programming of evolutionary models by incrementally changing the models in e.g. the NelWin tradition. This includes a discussion of Lsd's C++ based equation language, the methods of compiling Lsd models, and experiments that extend the basic NelWin model (further information is found in the appendix). Section 5 contains conclusions and perspectives that relate to the way the Lsd system can be used to overcome some of the difficulties of the Nelson–Winter tradition of evolutionary modelling and, probably, other evolutionary modelling traditions.

2 STARTING THE LSD SYSTEM WITH NELWIN

2.1 A hands-on approach

The best way of exploring complex constructs like NelWin and Lsd is to start using them through a hands-on approach. This *in-medias-res* strategy is applied in the following, and we are only gradually adding background information. We simply assume that you have the Lsd system on your computer. Your task is to localise and start Lsd Model Manager, or LMM, that is included in the Lsd system. Then you can study the functioning and the facilities of a model implemented in Lsd. To allow a focus on Lsd, we shall in this and the next sections ignore the detailed functioning of the NelWin model. Thus you will only gradually develop an understanding of the details of the NelWin model.

Table 1: Lsd's main windows		
Window name	**Main functions and suggestions**	**Section**
Lsd Model Manager	An independent part of the Lsd system used for the selection, creation or modification (limited to the computational content) of Lsd models.	4.3
Lsd Browser	Opened automatically. Lsd's main control centre with menus as described in Table 2. The window is labelled by the loaded model and shows one Lsd Object (a model's entity) content. In some cases you have to minimise other windows to see the Lsd Browser	2.2
Lsd Model Structure	Opened automatically after a model is loaded with menu File/Load in the Lsd Browser. The window shows the Objects of the loaded model. When the cursor is over an Object, you see the contained variables and parameters. If you click on an Object, you jump to the Browser with the Object open	2.3
Lsd Data Editor	Opened by menu Data/Init.Values in Lsd Browser. In this window you set the values of a specific Object required to start a simulation: parameters and variables that are used with a time lag. The Data Editor shows editable data for the Object that is currently shown in the Lsd Browser	3.6
Lsd Object Number	Opened by menu Data/Set Number of Obj/All Obj. Number in Lsd Browser. This window allows to determine the number of entities in the model.	3.8
# Run Time Plot	Opened automatically. During a simulation this window shows the gradually calculated results for the variables selected for run time plotting. If you name several simulations in a single run, there will gradually be opened new Run Time Plots, one for each simulation	2.4
Log	Opened automatically. Contains messages from the system (e.g. errors and simulation steps) and allows to control simulation flows (e.g. interrupt or abort the current simulation run).	3.9
Data Analysis	Opened by Data/Analysis Result menu in Lsd Browser. Can either load data saved from a previous simulation or data from the simulation that has just been made. The window gives access to a large number of features to analyse the simulation results.	2.5, 2.6
Plot	These windows contain the graphical plots produced with Data Analysis. They are placed in the background when double-clicked.	2.5, 2.6
Equation	Opened by double clicking one of the variables listed in the Lsd Browser window and pressing Equation (and maybe selecting the equation file). Shows the equation of the variable. If you double click on one of the variables (not parameters!), you will get a new window with the equation for that variable.	3.2
Model Report & Help	These are two HTML files (generated by the modeller) that can be found in the folder of the model. They are accessed from the Menu Help in the. The Model Report shows the links between each element of the model (e.g. in which equation a parameter is used). The Model Help shows tables of the model elements, possibly filled with comments and information by the modeller. Model Help and Report are reciprocally linked.	3.3

Table 1 reports the main elements of the Lsd system of interfaces. It is mainly designed for later reference and so we shall not comment now the individual windows in detail. To start a Lsd model, locate and run the LMM program. This program (in Windows called **`lmm.exe`**) can be found in the **`Lsd4.1`** folder. When LMM starts, the program asks which action you want to perform. Choose the '**`Open an Existing Model`**' and, in the list presented, select the '**`Nelson and Winter`**' model, and click '**`Ok`**'. Now LMM has selected the model Nelson and Winter (as you can check in the upper-left corner of the LMM window).

Now you can run the Lsd model program: locate in menu **`Model`** the entry **`Run`**. The LMM window

will disappear, replaced by a message telling you to wait. At the very first run of a Lsd model this process is likely to take one or two minutes: LMM is building every component the Lsd NelWin model program, while later runs will exploit the same compiled code, at least for the parts common to every model. When the LMM window reappears again it will be overlapped by the two main windows of a Lsd model program: Browser and Log. At this point LMM is not used any longer for the simulation session[16], and you can concentrate on the Lsd own main window: the Lsd Browser.

[16] LMM and Lsd model programs are completely separated. In fact, you may also shut down LMM keeping on working with the Lsd model program. However, you may likely prefer to keep LMM in the background, in case you want to test later a different model. Just beware that the icons of LMM and Lsd are identical, which may cause confusion in the icon bar of the running programs.

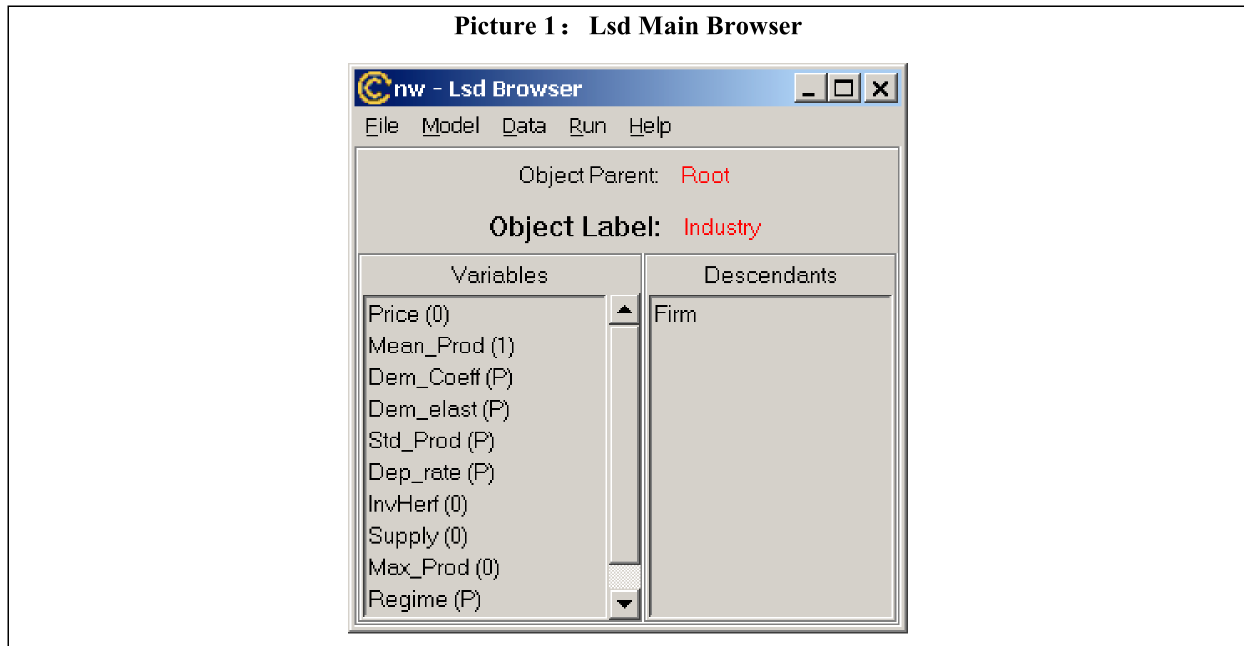
Table 2: Menus of the Lsd Browser Window	
Menu	**Item**
File Manage configuration files	• **Load** (load a configuration) • **Re-Load** (re-load the same configuration after a sim. run). • **Save** (save the current configuration) • **Empty** (discard the current configuration) • **Quit** (close the Lsd model program)
Model (for modellers only) Modifies the structure of the object shown in the Browser.	• **Add a Variable** (create a new variable in the object). • **Add a Parameter** (create a new parameter in the object) • **Add a Descending Obj.** (create a new object in the obj). • **Insert New Parent** (create a new obj. containing the obj.) • **Change Obj**. **Name** (modify, or remove, the obj) • **Equation file** (set the file name containing the equations). • **Create Report** (generate the model report) • **Create Help** (generate the model help)
Data Manage the numerical content of the model	• **Set Number of Object->All types of Obj.** (determine the number of copies for each object type in the model) • **Set Number of Object-> Only current type of Obj** (determine the number of copies for one type of object) • **Init Values** (determine the values of parameters and lagged variables for the all the copies of the current type of object) • **Analysis Result** (activate the Analysis of Results module) • **Save Results** (save the results from the latest simulation, together with the configuration that produced them) • **Data Browse** (inspect the model current status, browsing each single copy of each Object type)
Run Manage the running options.	• **Run** (start a simulation run) • **Sim Settings** (set simulation options, like number of steps, or of simulation runs) • **Remove Debug Flags** (remove debug flags in each variable of the model: Lsd debugger will not be active) • **Remove Save Flags** (remove save flags from each variable of the model: no data will be saved) • **Remove Plot Flags** (remove the Run Time Plot flags: no plot is produced during a simulation run) • **Show Save Vars.** (show in Log the data with the flag Save on) • **Remove Runtime Plots** (destroy all the Run Time Plot windows).
Help Open help pages	• **Lsd Help** (open in your web browser a quick-help page for the most frequently used operations, each of which is linked to the relevant page in the main Lsd manual) • **Model Help** (open in your web browser the model help) • **Model Report** (open in your web browser the model report)

2.2 The Lsd Browser

The Lsd program will open a window referred to as the Lsd Browser (there is also a Log window to which we shall return later). At this time, the Lsd simulation model is ‘empty’ because no model definition has been loaded: the model contains the equations’ code, but not the variables’ labels. The Lsd browser serves as the main control centre for any action and it also provides a detailed account of the model content. The browser window has five menus (`File, Model, Data, Run,` and `Help`). Table 2 gives an overview over the items of the individual menus. Their functionality will be commented upon later.

The Lsd Browser (see Picture 1) plays two roles: shows information on the structure of the loaded Lsd model and serves as control centre for a simulation session. The first operation when a model is launched is to load a configuration file: select **File/Load** and then select the lsd configuration file **nw.lsd**. Now the browser window is filled with information. Concentrate on this information and ignore for a moment a Lsd Model Structure Window that has shown up in the background.[17]

Picture 1 : Lsd Main Browser



To understand the information in the Browser Window, it is important to note that Lsd models are built after some of the principles of Object Oriented Programming.[18] This means that each model is assembled as a set of hierarchically ordered Objects—representing entities of the simulated reality, like industries and firms. A lower position in the hierarchy means that the Object is contained in its ancestor. Each Object has a name or label, and it contains several Variables, Parameters, or other Objects. At present, after a model is just loaded, you look at the Object = **Root** (the overall Object of the model) that contains no variables and parameters but just another Object = **Industry**. If you double click on the name **Industry** the Lsd Browser moves to show this Object and you can consider its contents: 10 Variables and Parameters[19] (like **Price** and **InvHerf** = the inverse Herfindahl index of concentration) as well as another type of Object = **Firm**. If you double click on **Firm**, you see its 11 Variables and Parameters—like the state variables **A** (productivity) and **K** (physical capital), as well as **ms** (market share), etc. (we shall return to the meanings of the variables).

---

[17] Lsd generates many windows during the different activities, and it is therefore important to manage them easily to avoid assuming a window as 'disappeared' when it is simply hidden behind other ones. The Windows tool bar shows always sets of icons for all the windows available, and is sufficient to click on the icon to raise a window on the foreground. It is often useful the combination ALT+TAB to raise cyclically the active windows.

[18] However, experts of Object Oriented Programming will find the definition of Objects pretty limited, since e.g. the contents of a Lsd Object is not protected and there are no concepts like inheritance.

[19] The Variables are marked with integer numbers, in the picture all 0's or 1's. These integers indicated how many lagged values for the Variables are used in the model (see below on this). Parameters are marked with (P).

To 'move up' the Lsd Browser along the hierarchy of the model you need to click on the name of the 'parent Object' (e.g. **`Industry`** when the Browser shows **`Firm`**). Note that the Browser shows only the structure of the Objects in a model, but does not contain any indication on their actual number. For example, it shows only one line referring to **`Firm`** as contained in **`Industry`**, although, in fact, there are many 'copies' of such Object **`Firm`**.

To run the model we, of course, need to supply values for Parameters and initial values for lagged Variables. But don't worry. You have already loaded a set of default values along with the model structure.[20]

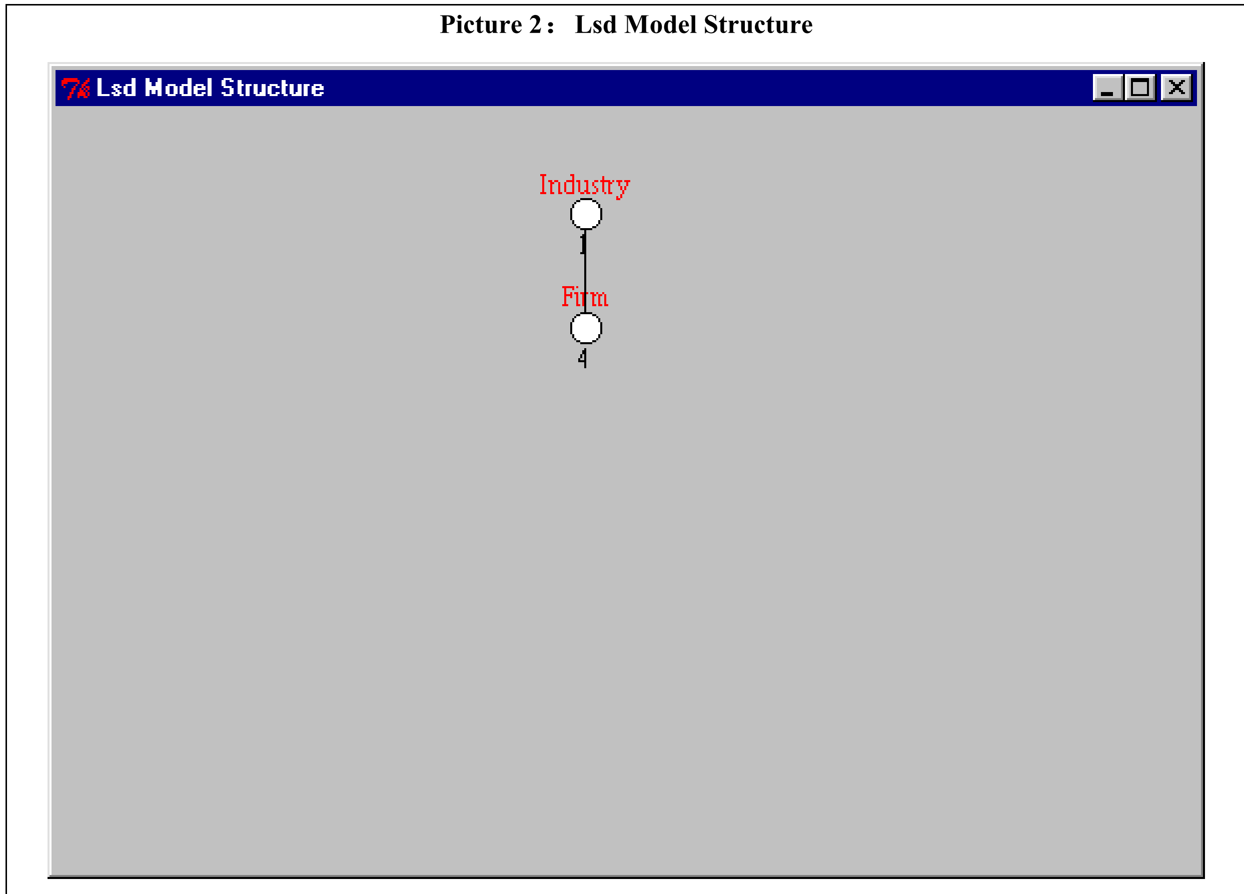
2.3 The Lsd Model Structure Window

Now bring in front the Lsd Model Structure Window (see Picture 2) by clicking on it or on its icon in the toolbar. If the Lsd Browser disappears from your sight, don't worry. It can still be accessed through the program line at the bottom of your screen, named '**`nw - Lsd Browser`**' (try to return to the Browser by clicking the name and then return to the Lsd Model Structure Window again). Do never close Lsd windows by using the closing facility of windows (the X in the upper right corner of a window). This will abruptly shut down the Lsd program and you will loose all the information contained.[21] Navigate by means of the bar with collapsed windows placed at the bottom of the screen. In this way you keep the basic Lsd windows open, while you close other Lsd windows with the Exit button, etc. When a session is finished, you should close the overall Lsd system through the Lsd Browser by choosing **`File/Quit`**.

[20] Curious users may have a look at the Lsd model files (extension '**`.lsd`**'). They are just plain text files that can be viewed with any text editor, e.g. WordPad, or the same LMM. It will be clear how Lsd reads the information about a model, with a first section describing the model structure (e.g. that **`Industry`** contains **`Firm`**) and a second section defining their numerical values (e.g. that there is 1 **`Industry`** and 4 **`Firms`**). This file also contains other information about the model, like the number of simulation steps, or the equation file that generated the model's computational content.

[21] If the Lsd model program is close, just re-run it from LMM (menu **`Model/Run`**) and re-load the model configuration.

Picture 2 : Lsd Model Structure



Back in the Lsd Model Structure Window, you see a very simple structure: one **`Industry`** contains four **`Firms`** (note that the Root Object is not showed in this window, since, in general, it should never contain any information relevant for the model). If you place the cursor over the symbols for **`Industry`** or **`Firm`**, you see a new small window that lists temporary the content of that Object (i.e. a list of the Variables and Parameters contained in the Object). This provides quick information on the model content. However, the main instrument to investigate the model content is to use the main Browser. If you double click on one type of Object, you return to the Browser Window inspecting that Object. Therefore, there are two systems to navigate a Lsd model: using the Lsd Browser's Objects labels, or double-clicking the Objects symbols in the Lsd Model Structure window.

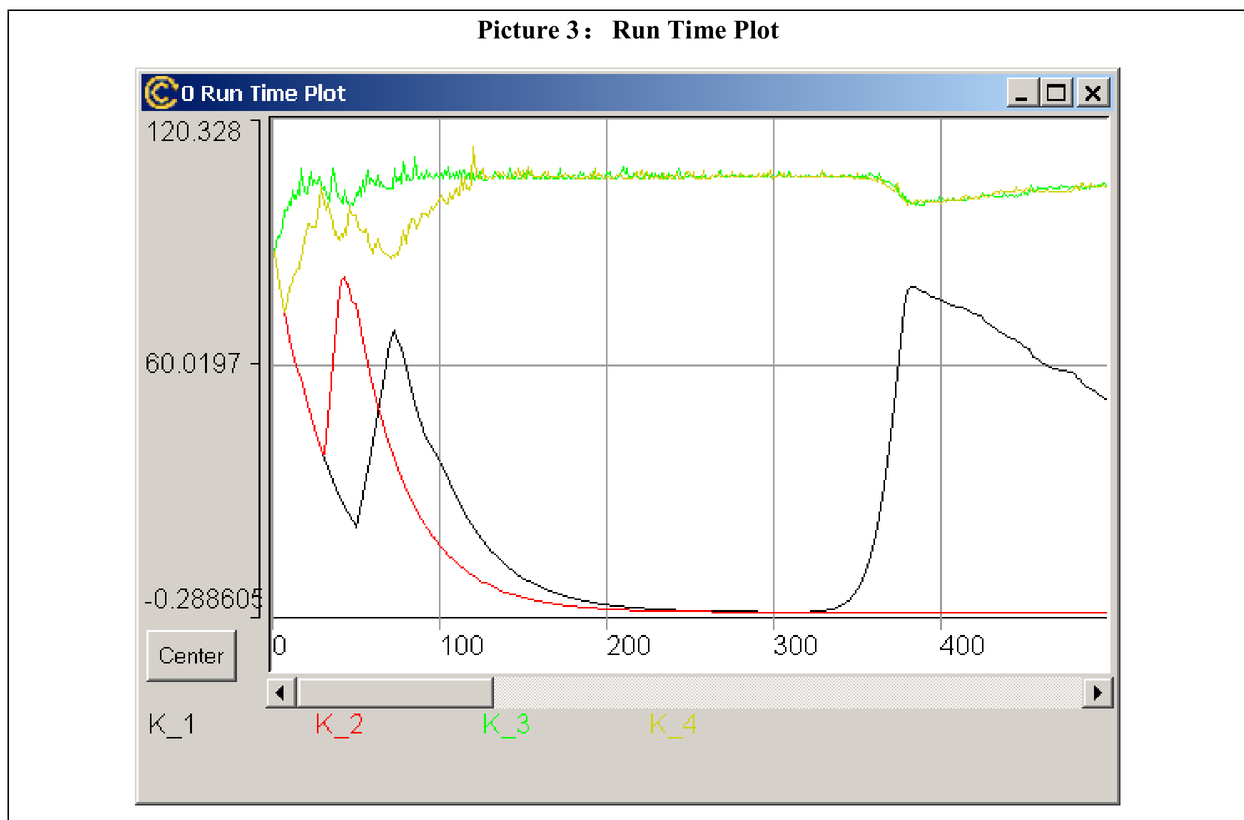
2.4 A simulation and its Run Time Plot Window

It is now time to try out the NelWin model with the default settings of Parameters, Variables, and number of Objects. A simulation has to start with a 'fresh' version of the data file (stored in **`nw.lsd`**), so if you have already made a trial simulation, you should load the file again (**`File/Re-Load`**, **`nw.lsd`**). This procedure has to be repeated every time you run a new simulation, because after a simulation run Lsd keeps the latest values that, in general, cannot be used to start a new simulation

run.[22] Given that we have a fresh data set, we run NelWin by choosing **`Run/Run`** and clicking the button **`OK`**. The simulation is pre-set to run for 2,000 time steps.

During the simulation two windows are giving information. One is the Log Window (ignore it for the moment!). The other is the Run Time Plot Window (see Picture 3). The latter window plots the values from pre-selected Variables. The selected Variable is **`K`** (the level of physical capital) for all the **`Firms`**.

Picture 3: Run Time Plot



The Plot Window indicates the **`K`**s of the different **`Firms`** (**`K_1, K_2, K_3, K_4`**) in different colours. Depending on the stochastic success of the firms' innovative and imitative activities, they expand their capital or allows it do depreciate. However, the set-up of the model is so that even the losers have extra chances (there is no bankruptcy mechanism in the simple model). So there is just a smaller and smaller fluctuations in the process of Schumpeterian competition. After the simulation is over, the window 'disappears' (actually, it goes in the background behind the newly raised Lsd Browser and Lsd Model Structure). You can see it again by selecting 'Run Time ...' from the icons at the bar at the bottom of the screen.

[22] Users can however 'overstretch' a simulation run beyond the original number of time steps. This can be done saving the model configuration after the end of the simulation, then load the just saved model and run it. That is, the latest step of a simulation run is defined as a new configuration from which to start a new run.

Later when you have more **Industry**s and **Firms**, the Plot Window automatically generates as many series as necessary, using different colours to differentiate among them, and applying a coding system to distinguish the elements with the same Variable name. You will also remark that the window automatically update the Y scale so to include the values generated during the simulation. Users can manually shift horizontally the window using the horizontal scroll bar to select a given portion of the simulation time, or press the button **Center** to reach the currently computed point. In fact, the window does not compress the X-axes (time steps), and when the simulation steps exceed 500, only a portion of the window can be observed (though the plot is constantly but invisibly updated).

2.5 The Data Analysis Window: Example

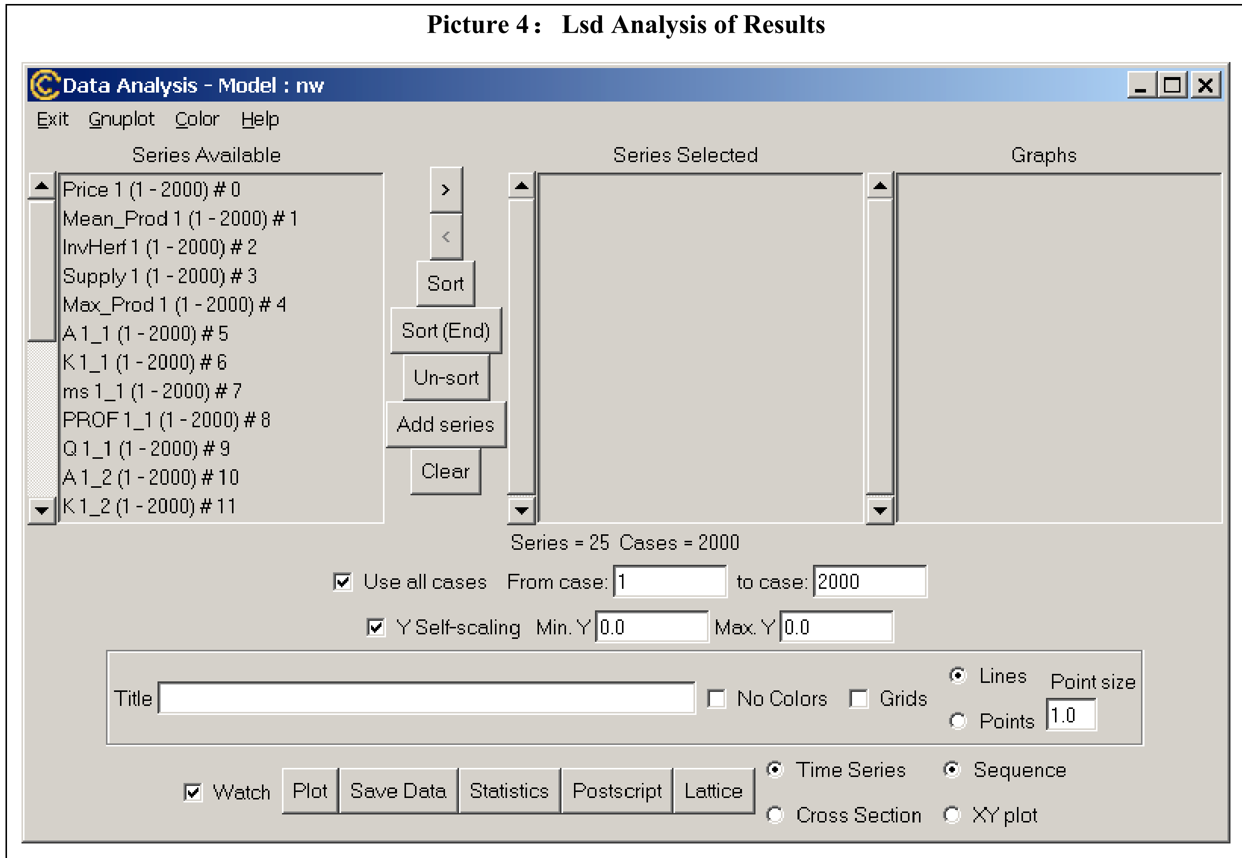
During the simulation we have only followed the movement of the physical capitals of individual **Firms**. However, many other data have been saved in the memory of the computer (as specified in the default settings of the model configuration you loaded and ran). In Lsd model users can choose for themselves which data they want to save during a simulation. These choices are automatically saved in the Lsd Model File when the simulation is run. Running the simulation without any change we have implicitly accepted the choices made by the modeller (included in the **nw.lsd** model configuration file) [23].

Lsd offers a relatively powerful set of tools to analyse the results from simulation runs, including the possibility to export data in standard formats for more technical analysis.

Let us have a few tries of the possibilities. First locate the Browser Window (**nw - Lsd Browser**), then select **Data/Analysis Result**, and finally select the button **Simulation**. Now we see the Data Analysis Window (see Picture 4) with three main columns: Variables Available, Variables to Plot, and a list of Graphs.

[23] Just to check, select the menu **Run/Show Save Vars.**, and raise the Log window. Lsd will have reported the variables to be saved during a simulation run.

Picture 4 : Lsd Analysis of Results



At the moment there is simple a list of available Variables, reporting the name of the Variable and the codes necessary to distinguish them as included in different copies of the Object. For example, **`A_1_3`** and **`K_1_3`** refer to the productivity and capital of the third firm in the first (and only) **`Industry`** of the market. The number within parenthesis (1-2000) means that the Variables have existed (i.e. the Object containing them have existed) from step 1 to 2000[24] (the integer after # is a simple counter).

To make the list handy we click the button **`Sort`** between the first two columns.[25] Then we shift-click and drag to select the **`K`**s for the four **`Firms`**.[26] Then we click the Move Left button (marked by '**`>`**'). Note that in the entry called **`Title`** contains the name of the first variable to plot. This name is only used as reference to the plot, and can be freely modified by users. Finally we click the **`Plot`** button in the lower part of the Data Analysis Window. Now Lsd plots the same information that we have already seen in the Run Time Plot Window—but in a way that shows all the information for the 2000 periods in a single graph. When the graph is created Lsd automatically places it in the foreground to

---

[24] This information is redundant in this model, but becomes important in case of data saved from Object that are created and/or destroyed during the simulation run.

[25] The 'unsorted' list presents the variable grouped according the Object they are contained in, starting from the highest Objects (i.e. **`Industry`**) to the lowest (**`Firms`**). The **`Sort (End)`** button is used in case of missing data, when the analysis must deal with Objects created and destroyed within a simulation run.

[26] The window applies the standard rule for managing selections. Clicking on an item selects it, or deselects it if previously selected, also deselecting every previously selected item. Keeping the key Shift pressed and clicking with the mouse over two items selects every item in between. Keeping the key Ctrl pressed adds to the selection individual items.

better observe it. To bring the main Analysis window in the foreground again, e.g. for creating a new graph, you can, besides using the usual toolbar icon, double-click any part of the graph. If you want to bring a graph in the foreground you can double-click its label in the 'Graph' list. These shortcuts allow to easily manage even a large number of graphs. A graph can be removed by selecting it in the Graphs list and pressing the key Delete (or clicking on its title with the right button mouse).

To prepare for the next plot, click the `Clear` button (removes the items in the '`Vars. to Plot`' list, equivalent of selecting all items in the list `Vars. to Plot` and click on '`>`'), select the productivities (`A_1_1, A_1_2, A_1_3, A_1_4`), click Move Left ('`>`'), optionally deselect the checkmark `Watch` (if you want to speed up the rendering of graphics), and click `Plot`. Now you see the narrow band of productivity development in the firms. To study the details of productivity evolution it is obviously an advantage to focus on a few years. To obtain this you access the main Analysis window (double-click the graph), deselect the mark for `X Self-scaling`, change the last number from 2000 to 100, and choose `Plot`. The new graph will use the same variables present in the `Vars. to Plot list`, but using only the values from step 1 to 100. Now you see the productivity differentials that exist in the beginning of the run. Some firms are quick to innovate or imitate while others are less lucky.

Finally, it is time to study some of the industry-level variables: double-click the graph hiding the main Analysis window, click `Clear`, select and move `Max_Prod` and `Mean_Prod`, and `Plot` 100 periods. Now you see that in some periods there is a gap between maximum and mean productivity of the industry while this gap is closed in other periods. We also inspect the index of concentration: double-click the graph hiding the main Analysis window, click `Clear`, select and move `InvHerf`, and `Plot`. In the beginning the inverse Herfindahl index shows that the four firms have practically equal market shares, in the middle of the period there are somewhat larger size differences, but at the end of the period concentration has again become smaller. Still, the differences are remarkably small! So we see that we later have to study the set-up and the parameters of the NelWin model to understand why this is the case.

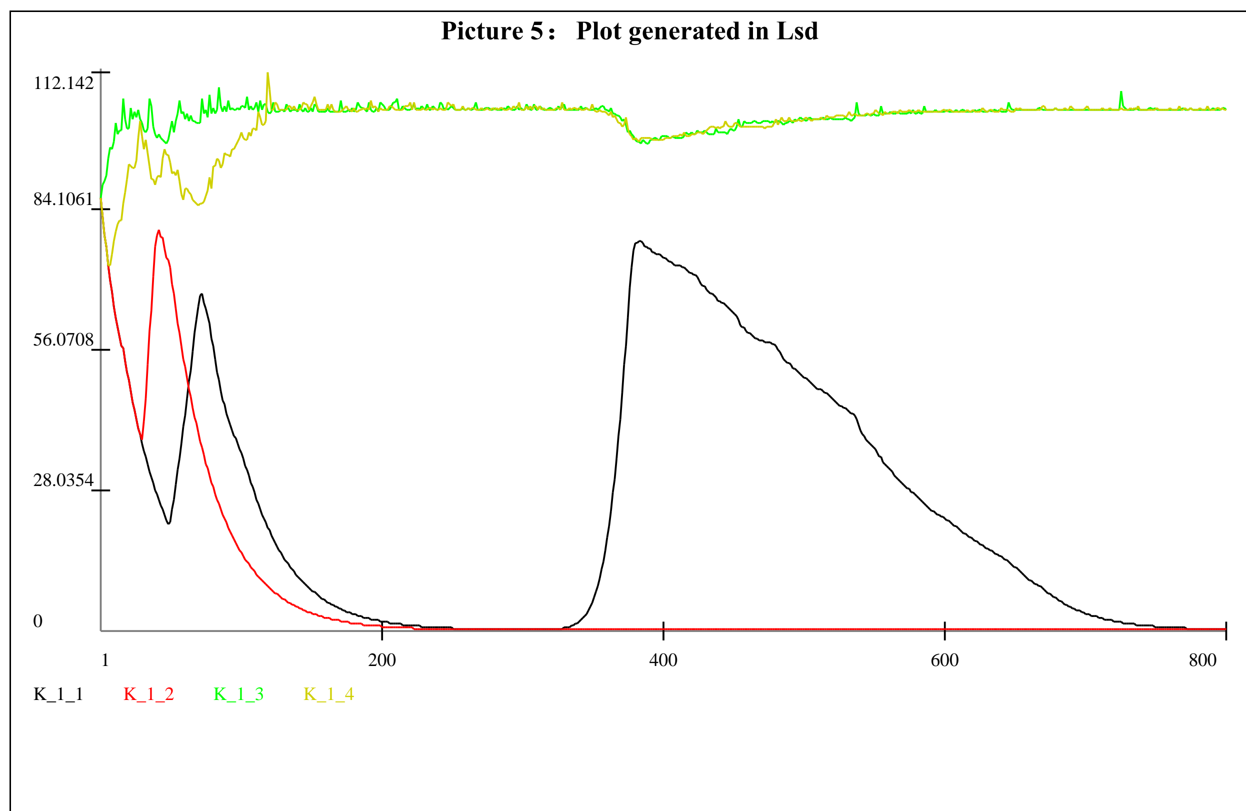
2.6 The Data Analysis Window: Possibilities

Before closing down our `Analysis of Results` session, it is useful to inspect the Data Analysis Window a little closer. We quickly recognise that when a simulation is finished, we have in the Lsd `Analysis of Results` module possibilities that allows an extensive study of the data produced during the simulation. Using simple and fairly intuitive graphical commands it is possible to:

- Create plots according a variety of combinations (e.g. time series, cross-section, scatter plots, 2D lattices)
- Determine automatically the X and Y scale of the graphs, or select them manually

- Produce descriptive statistics of given series, either cross section or over time
- Export data from the simulation using a Lsd or other formats (e.g. fixed or variable column text files)
- Export graphs (in EPS format); when choosing this possibility we can determine the dimensions and orientation of the saved plot—which can later be included in any word processors' document.

The **`Analysis of Results`** module can be applied both to the values of the just finished simulation or to result files from previously saved simulation runs (always maintaining the coding systems to distinguish between different instances of the same elements). This module is particularly useful because of its efficiency. In fact, exploiting the internal format for storing Lsd data values, it is possible to plot graphs that would take several minutes using commonly used packages in a matter of seconds (have you ever tried to plot series with thousands of values in Excel?). Picture 5 gives an example of a simple plot that has been saved as an EPS file (width about 140 mm), added a Windows preview by using PostScript utilities (GhostScript 5.50 and GSview 2.7), and inserted in MS Word. Of course, TeX/LaTeX users can directly embed their graphs in a document.



Picture 5: Plot generated in Lsd

2.7 Closing the session

Concluding this early exploration of the Lsd's implementation of NelWin, we learnt that there is a computer program, the Lsd NelWin model, which is able to 'load' a definition of a model, contained

in a file with extension '`.lsd`'. When the model is loaded we can browse the model at our will and run the simulation with a given configuration of the model. Finally, the results of the simulation can be viewed with a special Lsd module. The most important aspect to keep in mind is that the program and the file describing the model are separated. In fact, we may (and we will) create many different configurations of the same basic model that make use the same simulation program (i.e. equations, but different configuration files (initializations).

However, before testing all these possibilities, we need to understand NelWin better. So at the moment we choose **`Exit`** from the Data Analysis Window, in the Lsd Browser choose **`File/Quit`**, and select the **`Yes`** button.

3 ANALYSING NELWIN WITH LSD

3.1 A closer look at NelWin

We now have got a rough impression of both Lsd and NelWin. The next step is to find out how we can obtain more information on the model and the simulations. To do so we restart the NelWin model program, again by clicking '**`Run`**' on the LMM window when the Nelson–Winter model is selected. As before, load the data file (**`nw.lsd`**), but this time save it under a new name by choosing **`File/Save`**, writing **`nwtest1`** and clicking **`Save`**. Now we can play around with the model structure but always return to the standard settings[27] (stored in **`nw.lsd`**). Given these steps, we are ready to safely perform experiments. However, it is necessary first to have a closer look at the NelWin model—both the equations and the values of Parameters and lagged Variables, that is the values that are of importance to compute the very first step in a simulation run. After we have inspected this information we shall turn to a new round of simulation experiments.

3.2 Lsd equations

Variables and Parameters are numerical values and the difference is that Variables are associated to an equation that changes their values at each time-step, while Parameters are not modified by the normal simulation flow.[28] To fully understand the actual functioning of a model, it is therefore necessary to

[27] Lsd always saves the data used for the initialisation of a model immediately before running a simulation. Therefore, it can happen to make modifications to a configuration, which erase the original one as soon as a simulation is run, and being unable to remember what was changed…

[28] Lsd philosophy is to express common operations in simple ways, but allowing overruling to code particular computations. For example, the default option is that Variables are computed once and only once and Parameters do never change during a simulation run. But sophisticated models can overrule these default choices permitting, for example, to have a variable computed many times during the same step or writing new values for Parameters during the simulation.

access the code implementing the equations for Variables. This code can be inspected (not changed![29]) by double clicking on a Variable label in the Browser list, and then choosing the button Equation.[30] (In some cases, depending on the directories naming where Lsd is installed, a dialogue box may show up asking for the name of the file containing the equations. In that case you should select the Lsd file **fun_nw.cpp**). Try to double click Variable **Price** in the Object **Industry** (the price of the output of the industry). Now you see the Lsd code with some comments stating that **Price** is calculated by the following simple equation:

```
Price(t) = Dem_Coeff / (Supply(0)^Dem_elast)
```

In other words, the price of the output of the industry in a certain period (*t*) is determined by a fixed demand coefficient divided by the quantity supplied, raised to the power of the demand elasticity. The meaning of the zero after supply is that there is no time lag in the value of **Supply** when calculating the **Price**.

For the moment—and also for many more experienced users of Lsd models—the actual code provides as many mysteries as pieces of information. Especially the Lsd/C++ code for accessing information may look quite cryptic. However, the modeller has introduced in their code comments so users have a clear understanding of the equation.

The windows showing the equations code have several facilities, like differentiated colours for different parts of the code, text search to access quickly elements of interest, and hypertext links that allow to directly open new windows showing the code for related equations. However, for the moment we ignore these facilities. Instead we close the Equation Window by clicking the button Close.

3.3 HTML reports on equations and initial values

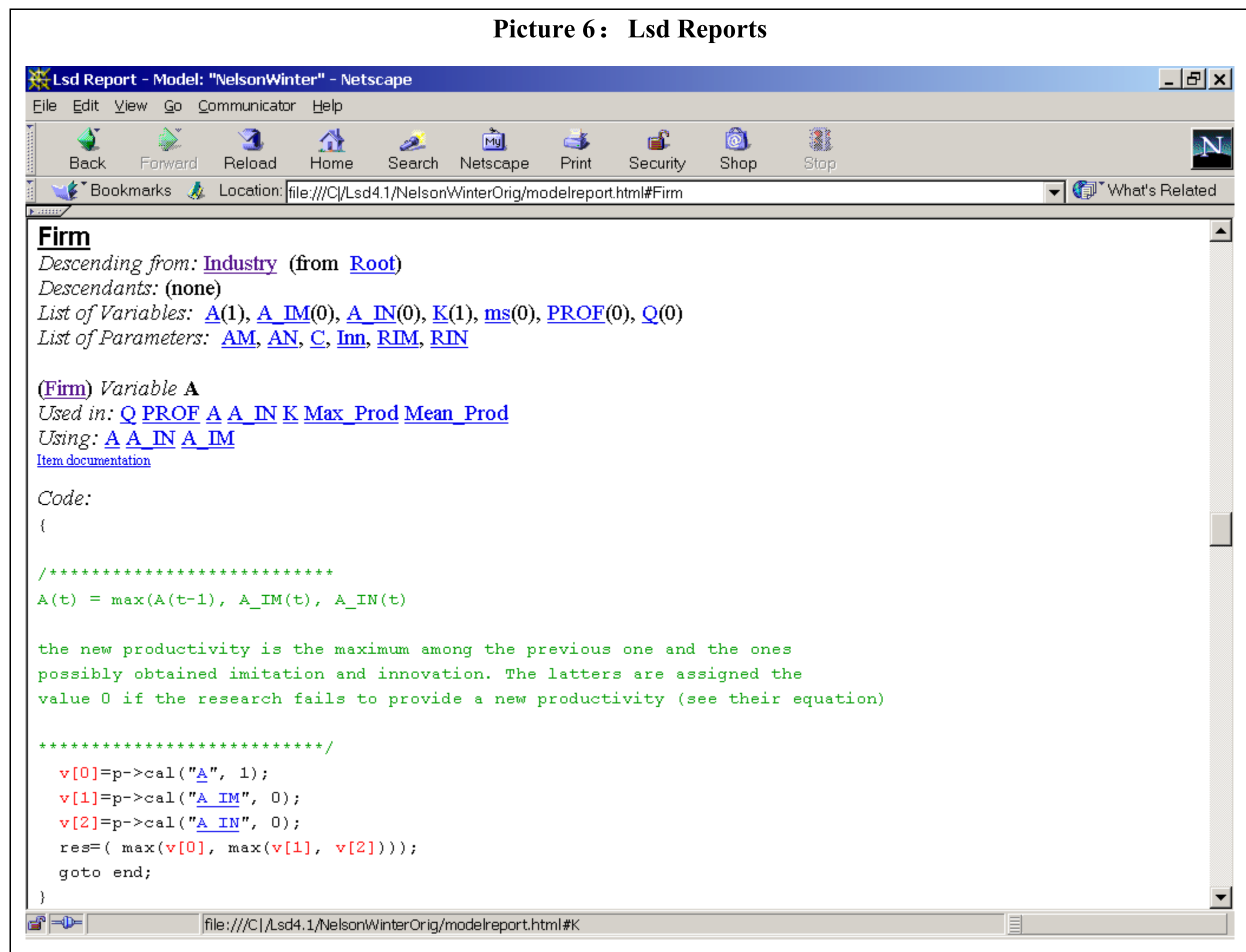
It is important to have a global overview over a Lsd model like NelWin so that one can browse the structure of the model at will, looking at the content of some entity, observing some equation and neglecting others. Although the Lsd interfaces provide a complete description of the model content, it is not always simple to collect information on the elements of the model. For this reason the Lsd Model Browser gives direct access to two documentation files in HTML format (inspected in a web browser like Netscape or Internet Explorer): **Help/Model Help** and **Help/Model Report**. The **Model Help** document can take varying forms, but the Lsd system automatically generates a template that includes the objects, variables and parameters of the model (to be filled in by the model

[29] The code shown is part of the very program that you are running, and therefore the modifications cannot have any effect. Instead, once the model is not running, you can modify the equations via the Lsd Model Manager which, when compiling the model, will include the modifications in the new program.

developer). The **`Model Report`** document is more standardised, since it is automatically generated from the equations' file (**`fun_nw.cpp`**), a Lsd configuration file (like **`nw.lsd`**), and text files with introductory information.

Choosing **`Help/Model Report`** you access[31] the latter HLML file. The document begins with some acknowledgement concerning the model. Scrolling down the document you find all the elements of the model. For example, in Picture 6 is shown the beginning of the section devoted to the Object **`Firm`** (including the equation that updates the productivity **`A`**).

Picture 6 : Lsd Reports



The Model Report contains a heading section (called 'Model Structure') that contains three lists for the elements of the model, that is the Objects of the model (the entities), the Variables and the Parameters. These lists provide a handy way to find quickly the elements of interest. After the summary sections, the report contains the whole list of model elements described in detail. These are grouped according the Object they are contained in, and are presented in such a way that every element shows the connection with the other elements of the model. Each Object has its own

---

[30] Every Lsd interface is endowed with a number of handy shortcuts to speed up common operations. To show the content of an equation, for example, you can also click with the right button of the mouse on the Variable of interest.

alphabetical list of Variables and Parameters contained, together with the 'parent' Object which contains it, and the list of 'descending' Objects that are contained in it. After each Object there is a sequence of sections referring to the detailed content of its Variables and Parameters. For each Variable is reported:

- the list of the Variables that use this Variable in their equation;
- the list of elements (Variables and Parameters) that are used in the Variable's equation;
- the initial values (optional, and only if the Variable is used with lagged values);
- the code for its equation (optional).

Parameters are described in the same way but, of course, do not contain the list of elements used and the equation's code.

The reports are very useful for many different purposes. First, they allow users of a Lsd model to observe any part of the model choosing their preferred perspective (e.g. more or less technical, starting from the aggregate elements or from the micro one, etc.). But the reports can also be used to document models, for example including them when submitting scientific papers, or placing them in the web pages related a model. Moreover, modellers can use the reports to control the state of development of their models and check for possible problems. It is worth to stress that the reports are generated in a totally automatic way. Thus, for example, modellers can use the reports while programming to check the cross-effects of any modification planned in the model (or to debug possible errors). The code for the equations contained in the Lsd report copies the actual code used to create the Lsd model program, and therefore may be quite hard to read for non-programmers. However, the code includes also the comments, presented in different colour, so that the modeller can easily indicate the meaning of the equation in standard notation.

3.4 Summarising the information on the NelWin model

In Table 3 we try to summarise the information on the structure of the model and the initial values. This summary will be handy for our simulation experiments. The meaning of some of the variables and parameters will be explained in relation to the equations of Table 4.

Note that lag = 0 means that the variable is used by other equations in the model only after it has been updated, and therefore there is no reason for keeping its lagged values. Instead, lag = 1 for variable **X**

[31] On Windows systems the Model Report will appear in the default web browser, while on Unix system it will be opened in Netscape.

means that some equation in the model uses **X** at time $t-1$ to compute its value at time t, and therefore the Lsd system must keep these older values.

Table 3 : Summary of Objects, Variables and Parameters in NelWin					
Name	**Meaning**	**Type**	**Initial value**	**Lag**	**Comment**
Industry (1)					
Bank	Bank loans in proportion to profits	Parameter	0	-	
Dem_Coeff	Demand coefficient	Parameter	67	-	
Dem_elast	Demand elasticity	Parameter	1	-	
Dep_rate	Depreciation rate	Parameter	0.03	-	
InvHerf	Inverse Herfindahl index	Variable		0	
Max_Prod	Maximum productivity	Variable		0	
Mean_Prod	Mean productivity	Variable (with lag)	0.16	1	
Price	Price	Variable		0	
Regime	Selector of type of cumulative innovation: (1) industry based and (2) firm based	Parameter	2	-	Example of control parameter
Std_Prod	Standard deviation of innovative results	Parameter	0.01	-	
Supply	Supplied output from the industry	Variable		0	
Firm (4)					
A	Capital productivity	Variable (with lag)	0.16 0.16 0.16 0.16	1	
A_IM	Outcome of imitative R&D	Variable		0	
A_IN	Outcome of innovative R&D	Variable		0	
AM	Probability of success per unit of imitative R&D	Parameter	0.125	-	
AN	Probability of success per unit of innovative R&D	Parameter	1.25	-	
C	Total unit costs (incl. variable costs, capital rental, and depreciation)	Parameter	0.16	-	
Inn	Existence of innovative R&D (false=0 or true=1)	Parameter	0 0 1 1	-	Two firms with and two without inno
K	Physical capital	Variable (with lag)	89.7 89.7 89.7 89.7	1	
ms	Market share	Variable		0	
PROF	Profit	Variable		0	
Q	Quantity produced	Variable		0	
RIM	Imitative effort per unit of capital	Parameter	0.00112	-	
RIN	Innovative effort per unit of capital	Parameter	0.0223	-	

Of course, Variables used with lagged values need, as in the case of Parameters, initial values to be used at the very first time step of the simulation runs. For example, consider the case for the Variable **A**, the firm's productivity. For Firm i at time t, the equation is (see eqn. 8 in Table 4 below) computed

as:

```
A[i,t] = max(A[i,t-1], max(A_IM[i,t], A_IN[i,t])).
```

That is, the value of the new productivity is the maximum between the results of innovation and imitation, if this is higher than productivity from the last time step. Of course, the innovative and imitative productivities do not need to keep record of the lagged values, since only the most recent value is used. On the other hand, Variable A needs to keep its previous period's value, since this is used in its own equation. Thus the model needs an initial value for A supplied by the user, in order to compute the **`A[i,1]`** at the very first time step. The Lsd notation is always in terms of number of lags compared to the present time step, and so the former equation will be written as:

```
A = max(A[1], max(A_IM[0], A_IN[0])).
```

Table 4: NelWin equations in a modified notation	
Short-term process	
1. Firm's supply	Q[i,t] = K[i,t-1]*A[i,t-1]
2. Industry's supply	Supply[t] = Sum(Q[i,t]), for all i
3. Price formation	P[t] = Dem_Coeff / (Supply[t]^Dem_elast)
Technical change	
4. Mean productivity	Mean_Prod[t] = Sum(A[i,t]) / n
5a. Innovative outcomes with Regime = 1	if (RANDOM(K[i,t-1]*RIN[i]*AN)[32] AND Inn[i]=1) then A_IN[i,t] = Normal(Mean_Prod[t-1], Std_Prod) else A_IN[i,t] = 0
5b. Innovative outcomes with Regime = 2	if (RANDOM(K[i,t-1]*RIN[i]*AN) AND Inn[i]=1) then A_IN[i,t] = Normal(A[t-1], Std_Prod) else A_IN[i,t] = 0
6. Maximum productivity	Max_Prod[t] = max(A[i,t-1]), for all i
7. Imitative outcomes	if (RANDOM(K[i,t]*RIM[i]*AM)) then A_IM[i,t] = Max_Prod[t] else A_IM[i,t] = 0
8. Choice of technology	A[i,t] = max(A[i,t-1], A_IM[i,t], A_IN[i,t])
Capital formation	
9. Profit rate	PROF[i,t] = P[t]*A[i,t-1] – C – RIM[i] – RIN[i]*Inn[i]
10a. Maximum gross investment rate	if PROF[i,t] <= 0 then MaxInvestRate[i,t] = PROF[i,t] + Dep_rate[33] else MaxInvestRate[i,t] = PROF[i,t]*(1 + Bank) + Dep_rate
10b. Relative Mark-up	RelMarkUp[i,t] = C[i,t] / (K[i,t-1]*A[i,t])
10c. Desired gross investment rate	DesInvestRate[i,t] = Dep_rate + 1 – (Dem_elast / (Dem_elast – ms[i,t]))*RelMarkUp
10d. Final gross investment rate	FinalInvestRate[i,t] = max(0, min(DesInvestRate[i,t], MaxInvestRate[i,t]))
10e. New capital stock	K[i,t] = K[i,t-1]*(1 – Dep_rate + FinalInvestRate[i,t])

In Table 4 we have structured the equations that determine the values of the variables of Table 3 under three headings. The short-term process is covered by equations 1–3, which are pretty straightforward. Technical change is treated in the somewhat more tricky equations 4–8. It is useful to start from eqn. 8

[32] This is not a full specification of the special function, which we abbreviate as **`RANDOM(lambda)`**. It defines a more-or-less Poisson-like process. It returns either 0 or 1. In C++ a zero is interpreted as FALSE while anything else is interpreted as TRUE.

[33] Since **`C`** included depreciation, we have to add it again.

that was discussed above. Here the given technique **A[i,t-1]** is compared with the results of imitative activities (**A_IM[i,t]**) and innovative activities (**A_IN[i,t]**). Imitation (eqn. 7) takes place if the firm has had a 'draw' from a Poisson-like process that is influenced by the productivity of a unit of imitative R&D (**AM**) times the level of imitative R&D (**R_IM[i]*K[i,t-1]**). In that case it obtains the best-practice technique of the industry (eqn. 6). Innovation is calculated in two alternative ways (depending on the control parameter **Regime**). In both cases it is first determined whether an innovative 'draw' has been made from a Poisson-like process (like in the case of imitation). If this is the case, the actual innovation is drawn from a normal distribution with a fixed standard deviation (**Std_Prod**) and a varying mean. This mean can either be the average productivity of the industry (**Mean_Prod[i,t]**, see eqn. 5a and eqn. 4) or the already obtained productivity of the firm itself (**A[i,t-1]**, eqn. 5b).

The last section of Table 4 concerns Nelson and Winter's formulations about the accumulation of capital: In evolutionary theory the relevant issue is often gross investment (which might influence technical change, but not in the present model). To calculate **K[i,t]** the NelWin program only includes a single equation, which becomes the most complex equation of the program. This is not good programming style,[34] so we have chosen to spilt it up into several equations (10a–10e) that together perform the computation of **K**. The auxiliary equations concern **MaxInvest** (10a), **RelMarkUp** (10b), **DesInvestRate** (10c), and **FinalInvestRate** (10d). However, as the program is constructed, these variables are not available to the user. Instead they are non-stored intermediate results in the computation of **K** (10e). Let us consider these intermediate equations:[35] **Depr_rate** is included in the unit costs, **C** (used in eqn. 9), and therefore it has to be added to **MaxInvestRate** in eqn. 10a. Since the **DesInvestRate** of eqn. 10c is a gross concept, it also has to take the **Depr_rate** into account. In eqn. 10d we see that gross investment can never be negative. This means that the capital stock of a firm (eqn. 10e) can never decrease by more than the depreciation rate.

Table 4 defines the equations in a notation that is a compromise between ordinary mathematical specifications, computer languages like C++, and the concrete equations found in the standard Lsd NelWin program. The reading of it should not be too difficult, especially if the reader has studied

---

[34] Complex equations that do not store their intermediate results are more difficult to debug than more finely decomposed equations. Since distributed Lsd models are (implicitly) covered by the same GNU General Public License as the Lsd system as a whole, we could of course have changed the NelWin model according to our design principles. For pedagogical reasons we have preferred to leave the rewrite of NelWin to the reader as an exercise (see below). It should, however, be pointed out that Lsd models are moving targets. The first version of the present paper used an older version of NelWin to suggest improvements. But in the present version of NelWin these suggestions have already been included (an a bug has been removed). We intend to keep a more or less unchanged version of the present (17 Dec 2001) NelWin program in future distributions of the Lsd system (where it, however, might be called NelWinTutorial or something like that).

[35] The details of investment behaviour are not well documented in Nelson and Winter (1982, Ch. 12), so we have had to consult the appendix of Nelson and Winter (1978) as well as Winter (1984).

other expositions of the Nelson–Winter family of models (like Winter 1984 and Andersen 1996, Ch. 4).

3.5 Simulation Settings

Before we turn to the more interesting experiments, it is useful to note that there are ways to change the simulation exercises that do not affect the actual content of a model, and, of course, we can easily vary them in respect of the standard choices loaded together with the Lsd model file. This simple way of experimenting with NelWin can be accessed by choosing `Run/Sim. Settings`. Do so! This choice allows us to access the following possibilities:

- The number of simulation runs.
- The seed for the pseudo-random number generator
- The number of simulation steps for each simulation run
- The time when the simulation should be interrupted to control the model state

The third facility is important for studying NelWin. Often we can get an impression of the process of evolution by studying in-depth a short simulation run for 25 or 100 or 500 periods. However, this strategy removes our interest from the long-term behaviour of the evolutionary system: will it converge or keep evolving? With a fast computer we can easily perform a simulation for 30,000 periods (if you have overestimated the speed of the computer, you can stop the simulation by accessing the Log Window and click the Stop button[36]).

The second facility, the seed for pseudo-random numbers, is more interesting. It reminds us that computers use an artificial system to generate random number. These are not really random values, but pre-defined series that, taken sequentially, replicate a so-called pseudo-random series. When a simulation is started, the chosen 'seed' (an integer value) resets the internal function to a different level that can be replicated in case the same seed is used. The result is that a simulation using the same initialisation and same seed will replicate exactly the same (pseudo-) random events. In the NelWin model it is especially the short-term behaviour that is heavily influenced by the timing of random events. So it is interesting to find different patterns of productivity advance and concentration by studying results for seed values of 1, 2, 3, 4, ... (You might also take 1, 100, 1,000, 10,000—since each integer gives a unique sequence of events.)

[36] The graphical representation for the Run Time Plots slows down the simulation time. An efficient trick is to minimise the Run Time Plot window during a simulation, so that the system does not have to continuously update the graphical window, speeding up the simulation sensibly.

The first facility, number of simulation runs, is used for running a battery of simulation runs. If you ask, for example, 5 runs, Lsd uses the same values for the model's parameterisation, while the (pseudo-) random number will differ in each simulation. In this case, the specified seed applies only for the first simulation run. Then it sequentially increments the seed for each of the following simulations, so to make a robustness test of the result against the random component. The problem with multiple simulation runs are due to the necessity to, both, access the results of each individual simulation run and have the possibility to compare results from different runs. Lsd takes automatically care of these aspects. If you decide to make more than one simulation run, the system automatically generates one result file containing all the series produced during each simulation run. Moreover, it generates also a summary result file containing the values of the Variables saved at the very last time step for each simulation. This summary result file will eventually contain one 'step' for each simulation run, so that they can easily be compared. Both kinds of result file (total and for each simulation) can be accessed using the module for the Analysis of Results described above, although the user must remember that in the case of the total simulation result file, the values do not refer to different times, but to different simulations.

While in the case of single simulation runs the Run Time Plot can be useful, in the case of multiple runs you may not want them. In fact, Lsd generates one Plot window for each simulation run, and your screen can get pretty crowded[37]. Choosing the menu entry **`Run/Remove Runtime Flags`** you tell Lsd to no plot any Variable during the simulation runs, which increases sensibly the simulations' speed. Of course, this option does not prevent Lsd to save the data for later analysis. In these cases, during the simulations, Lsd will provide indications on the current state of the simulation using the Log windows, writing the step number just executed. However, even this operation can be avoided to further increase the simulation speed by clicking on the Log window's button **`Fast`**. Clicking on **`Observe`** the system returns to write messages.

Lsd offers also two particular modes for running a simulation, used by programmers to test the internal working of their models. The **Debug** mode causes the simulation to be interrupted any time a Variable, selected for this purpose, is computed. This permits to appreciate in detail the effects of an equation, since the programmer can observe all the values of the model at the very moment that the Variable selected finished the computation. Secondly, with the option **Stack Info** set on, it is possible to run the simulation producing the sequence of Variable as they are computed. This permit to appreciate the

[37] If this happens (running 100 simulation runs generates 100 windows if you opted for run time plots!), you can delete all these windows at once using the menu entry Run/Remove Runtime Plots.

actual scheduling, which is automatically determined by the system interpreting the equations' code.[38] Advanced ordinary users can also gain insight be using these ways of studying e.g. the NelWin model.

We mention here also another feature of Lsd models. During a simulation run a large set of controls permit to signal all possible errors, and try to "save" a simulation run even after a severe error, like a division by zero. In these cases, the simulation is aborted but the user can still observe the state of the model and perform data analysis up to the latest time step available. These utilities are invaluable in debugging models, saving a large amount of time in searching for the coding errors.

3.6 Initial Data

In order to start a simulation exercise, it is necessary to provide the model with the initial data to be used for the Parameters and for the lagged Variables, whose values are used at the very first time of the simulation runs. The Lsd Model File (**`nw.lsd`** or the newly named **`nwtest1.lsd`**) already contains the initialisation chosen by the model writer, but users can modify each value to test different settings.

There are—as we saw in Table 3—two types of numerical values used during the simulation. One concerns the values of Parameters, that must be set before the simulation starts. The second concerns the lagged values for Variables. In fact, in general some equations in a model make use of the past values of some Variable, and these values must be provided exogenously, because at the very first time step the model does not have available those values from previous steps. Of course, if a Variable is used with many lags in some Variables' equations, then the users will have to provide as many initial values for that Variable. Moreover, any single instance of each Parameter, or any Variable with a lag, contained in an Object replicated in many instances must be individually initialised. Data initialisation is made in Lsd for each single type of Object (e.g., for the NelWin model, the **`Firm`** and the **`Industry`**), namely, the one currently shown in the Lsd Browser window.

Choosing the entry **`Init. Values`** in menu **`Data`**, the system will create a window reporting all the currently set values for all the instances of the Object currently shown by the Browser. The Data Editor Window (see Picture 7) is similar to a spreadsheet screen. Each line refers to a Parameter or to the lags of a Variable (Variables used with no lags in the model do not appear in this window because they do not necessitate any initialisation). Each column refers to an instance of the Object, using the automatic coding system seen above, and therefore you will see four columns for data if you asked the initial data when the Browser shows **`Firm`** and one column for data concerning **`Industry`**. Users can

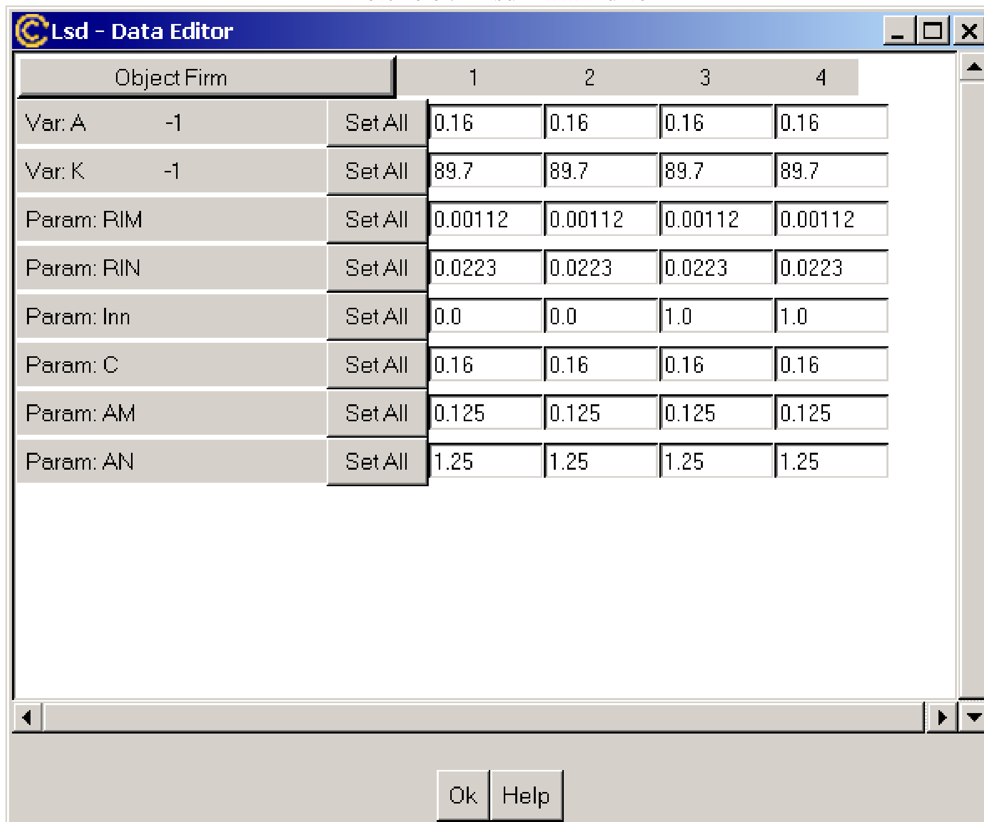
[38] Model writers in Lsd do not decide explicitly the scheduling of the computations for the variables within a time step. This is automatically solved by Lsd by using the implicit order indicated by the lags in the equations. However, errors may occur (the so-called dead locks), in which case Lsd issues an error message helping to solve the problem.

insert new values in the corresponding cells manually, or can assign globally new values to a whole line pressing the button **Set All**. In this latter case, they have several alternatives, like assigning identical values, incremental values or random values (the function **Set All** may be redundant for the NelWin model, but is crucial for models involving hundreds of thousands of Objects).

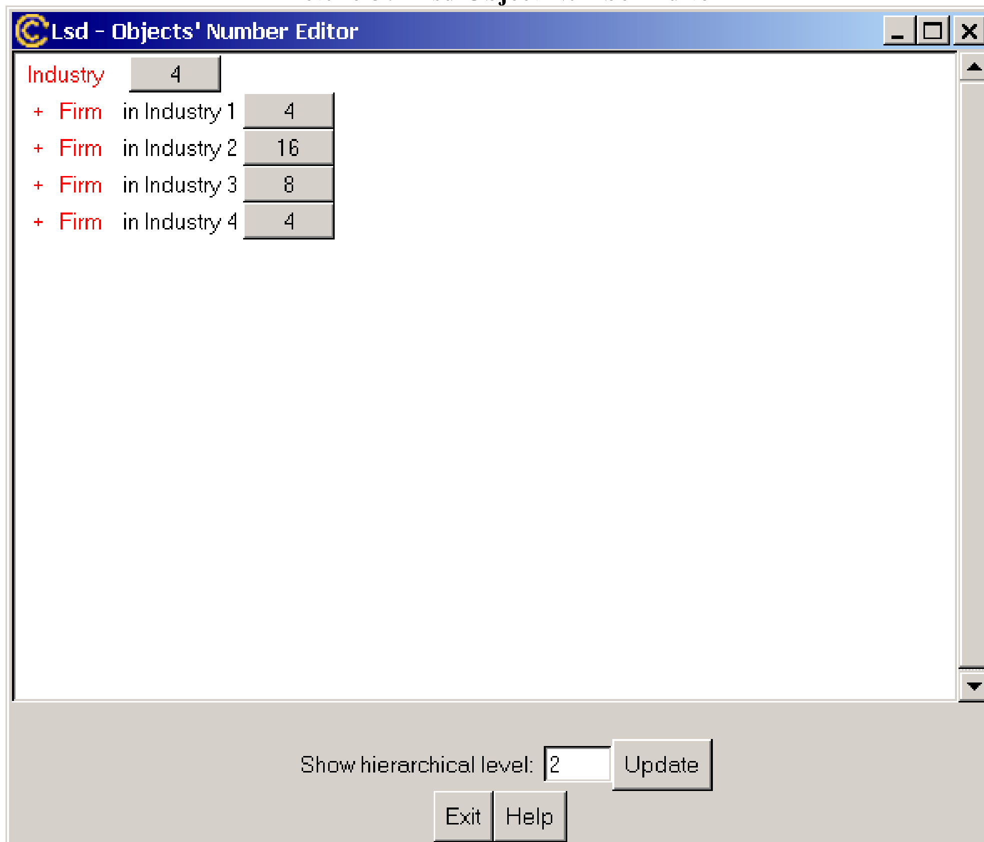
Besides the initial values for the model, there is also another type of numerical values characterising simulation models: the number of instances for each type of Object in the model. For example, the default configuration of the NelWin model includes one **Industry** and four **Firm**s. Choosing the menu entry **Data/Set Number of Obj/All Obj. Number** you will see a window showing the number of copies for each Object type. The window initially shows only the Object **Industry**, of which the model contains only one copy. On the right of the **Industry** a text will indicate that there are other objects, contained in **Industry,** not shown. Click on the text (or ask the window to show 2 levels in the hierarchy), and 4 **Firm**s will appear. You can change the number of Objects just pressing on the numbers. For example, clicking on the number 1 on the side of **Industry** you can include in the model, e.g. 4 **Industry**s. The newly created Objects replicate the structure of the existing one, including the 4 copies of Objects **Firm** contained. Clicking on the number 4 on the side of the new groups of Firm you can modify also these values, for example introducing 8 or 16 Firms in an **Industry** (see Picture 8).

Note that these configurations are extremely important for a simulation model to be fully explored. In particular, Lsd is well suited to implement the so-called agent-based models, where nested groups of entities interact at different hierarchical levels. In fact, the code for different levels can be added gradually, with the system automatically checking the consistency and resolving the most complex problems normally found when extending computer programs.

Picture 7 : Lsd Data Editor



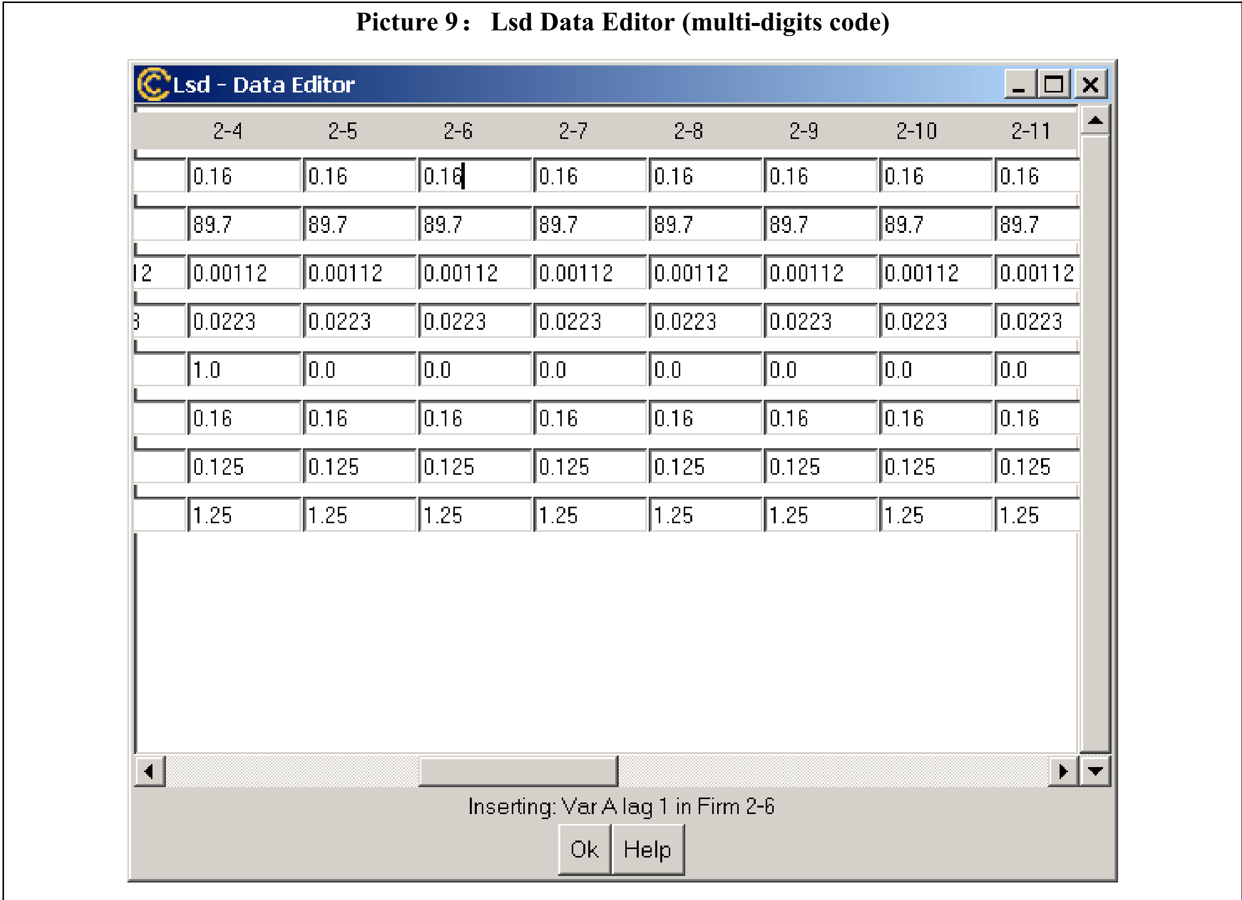
Picture 8 : Lsd Object Number Editor



Lsd automatically handles the effects of these changes both in the model to run and on the interfaces

for users to deal with the new structure. For instance, the Parameter **RIN** (intensity of innovative R&D) and Variable **A** (productivity) will be present in each **Firm**, or Variable **Price** in each Industry, but the user needs to refer explicitly to each instance to, for example, initialise the model or plot the results. Lsd automatically generates a coding system that allows individuating the precise instance of each element (see Picture 9, showing the initial values for **Firm** in the new configuration). So, for example, the Variable **A_2_6** will refer to the instance of the 6th **Firm** contained in the 2nd **Industry**.

Picture 9: Lsd Data Editor (multi-digits code)



The code for the columns is generated automatically any time it is necessary by the different modules of the system, and is dynamically updated in case users modify the number of entities in the model[39]. For the time being we will not use this settings because we need firstly to decide which simulation settings are interesting to be tested. Therefore, exit the initial data window and load a fresh Lsd model file, containing the original configuration[40].

[39] The same coding system is applied automatically to refer to the results, even in case the model admits the creation or destruction of Objects during a simulation run.

[40] Lsd models do not save the modifications to a configuration file until, either, the user does not save it explicitly or a simulation is started. Therefore, after you messed around with the interfaces to initialise the model, you can always load the original configuration and undo all the changes.

3.7 Planning a set of simulation experiments

To define systematic experiments with NelWin we ought to find themes from industrial economics like (1) factors influencing industrial concentration, and (2) factors influencing productivity growth and static efficiency. As the simulation results to explain we might take (1) industrial concentration measured by the inverse Herfindahl index (**`InvHerf`**) and (2) mean productivity of the Industry (**`Mean_Prod`**).

To perform the experiments, we start selecting a few Parameters from NelWin—partly relating to the basic conditions of technological change in the industry, partly to the investment behaviour of Firms. The Parameters might be (1) number of **`Firm`**s, (2) standard deviation of innovative results (**`Std_Prod`**), and (3) the external finance of firms based on their profits (**`Bank`**). Before performing the simulations it is important to specify our expectations about the consequence of different settings—to see whether or not the simulations surprise us. (The set of possibilities of parameter settings does not cover the full intentions of Nelson and Winter's models because the present Lsd implementation of Lsd is rather primitive. Its the improvement will be a test topic for would-be Lsd programmers).

For each of the parameters we might—in relation to Table 3—both a 'high' and a 'low' value. For example, we may set

- number of **`Firm`**s is 16 and 4;
- **`Std_Prod`** is 0.03 and 0.01;
- **`Bank`** is 2 and 0.

For each of the 8 combinations we need to set the correct values using **`Data/Init Values`**, and run the simulation. Given the presence of randomness, we better use several simulation runs to study the resulting concentration and productivity conditions, taking the average values for the total result files for each battery of simulations. Hopefully, you should be able to do this (boring) job, although at the moment we do not have the time to describe it in detail.

3.8 More on the Number of instances of Objects

We have seen above that users can easily modify the number of instances in a model, but this topic deserves further attention. Choosing the entry **`All Obj.Num.`** in menu **`Data\Set Number of Obj.`** the system will show the number of instances for each type of Object in the model. Users can click on the numbers and modify the values. The behaviour of the interfaces is differentiated depending if the number of entities is increased or decreased. If the user wants to include more instances in the model, then the system will append new copies at the end of the previously existing

set. Note that the new instances will contain numerical values for Parameters and lagged Variables that will need to be set by the user before running the simulation (an error message will prevent a simulation to be run without previously setting them).[41]

Let us see how to increase the number of `Firm`s. These numbers (like the default value: 1 `Industry` and 4 `Firms`) are parameters of the NelWin model (and other Lsd models). During a simulation there is (normally[42]) a fixed number of Objects and after the simulation Lsd is conserving the results of the run. So because we want to inspect (and potentially change) the number of Objects, if we have already run a simulation we need to load a model file with the original configuration,[43] choosing `File/Load`[44]. After that we want to have a 16 `Firm`s model. Besides following the procedure seen above (menu `Data/Set Number of Obj./All Obj. Num`), you can use the alternative one described below. Go to the Lsd Model Structure Window, double click `Firm` (this moves the Lsd Browser to point the `Firm` Object). In the Lsd Browser select `Data/Set Number of Obj./Only Current Type of Obj.`, write 16, click OK. As also indicated in the Lsd Model Structure Window, the model now contains 16 `Firm`s. This alternative procedure is very quick and is rather useful in case of large models where it is difficult to individuate the correct line in the standard window showing all the Objects are once.

Unfortunately, there is still a problem. In the original Parameter settings (see Table 3), there is a slight difference between the `Firms`. This difference has not been conserved in the increased set of `Firms`, since it was actually `Firm` number 1 that has been cloned for the new ones. So we have to make a few corrections manually. In the Lsd Browser, choose `Firm`, then `Data/Init Values`. In the Data Editor Window we see a problem with the Parameter `Inn`. Originally, 2 `Firms` were solely imitators while 2 `Firms` were both innovators and imitators, (i.e. 0.0, 0.0, 1.0, 1.0). Now all the `Firms` from 5 to 16 are only imitators, since they are clones of the very first instance. Thus we have a boring `Industry`, with 14 imitators and only two innovators. To change this we manually write '0' in the cells for `Inn` for the first half of `Firms` and '1' for the second half. After this change it might be time for simulations and for analysing the results, according to the routines described in the previous sections. But this time we can work more systematically.

---

[41] Clicking on the names of one Object type is a shortcut activating the Initial Data window for that Object, so that it is possible to move back and forth changing the number of some entities and setting their initial data. On the same token, the Initial Data window permits to activate the window to set the number of Objects by clicking on the label of the Object.

[42] Lsd permits to implement simply and efficiently models that create and destroy entities during the simulation runs. Data produced by the 'transient' Objects are duly saved for Analysis of Results, where missing values are automatically managed by the system and, for example, are not displayed in the graphical plots.

[43] Lsd prevents, issuing a message, to modify models elements after a simulation run, assuming that users will mistakenly try to run a simulation with the values resulting from a previous simulation.

[44] While `File/Load` asks for the name of the file to load, the entry `File/Reload` automatically chooses the Lsd model file with the same name as the one currently loaded, so to speed up the access this frequently used option.

If we have saved the model data with 16 **Firms** (and this happens automatically when a simulation run is launched), and open the file again, we might consider reducing the number of firms. The Lsd system offers two possibilities to eliminate existing instances. The first option tells the system to remove the last instances in the set, so to reach the desired number. The second option allows user to specify each particular instance to remove. The first option is, of course, much quicker, while the second allows for selective removal.

3.9 Log Window

Another issue to learn more about is the Log Window. Any time a Lsd program is running a window named Log is available. To see the Log Window, click it or click its icon at the bottom of the screen. This window serves two purposes: it communicates messages to the users and provides a control centre during the simulation runs, when the main Browser is inactive. Any time you find a Lsd model not responding to a command, bring the Log window in the foreground because, very likely, there will be a message telling what happened. The message reported in the Log window can be error messages (which normally contain indications to fix the problem[45]), communications about the beginning and the end of the simulation runs, or other information requested by users. For example, the descriptive statistics discussed in the Analysis of Results are reported in the Log window. Moreover, when the users decided to not have any run time plot during simulation runs (which reduces the speed of execution), the Log window reports the steps already computed and other information.[46]

The Log window contains five buttons:

- **Stop**. Quit the simulation
- **Fast**. Maximise the speed of the simulation (don't issue redundant messages)
- **Observe**. Return to the verbose simulation mode.
- **Debug**. Interrupt the simulation.
- **Help**. Open the Lsd manual on the page concerning the Log window.

The **Debug** action deserves further comments. Users can select some Variable to be debugged. This means that when the simulation enters in debug mode (see menu **Run/Simulation Settings**,

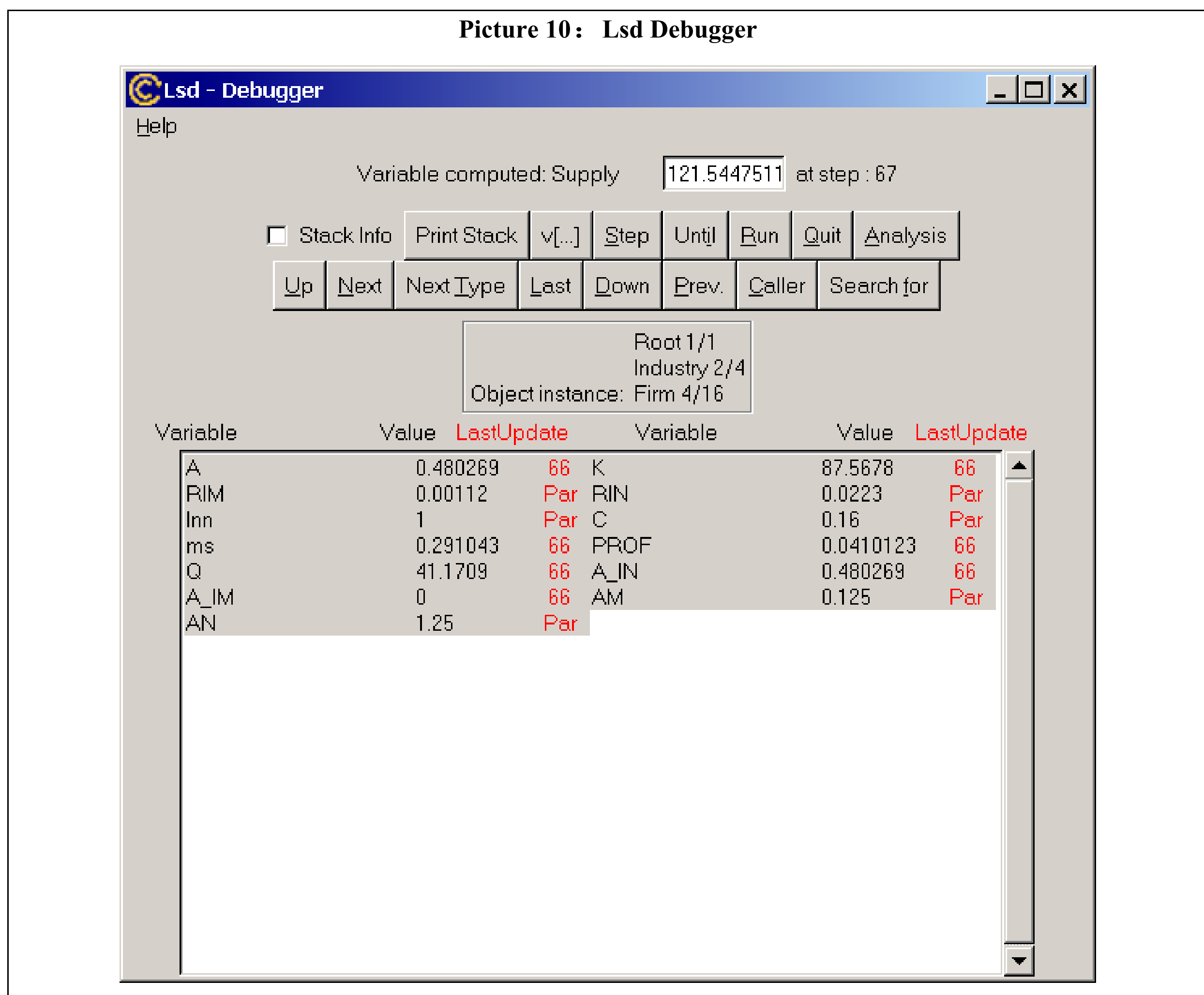
[45] Lsd models are protected against any kind of errors, eliminating the possibility to experience a crash of the program as result of (mis-)using the available interfaces, and for the errors common in simulation programs. If during a simulation an error occur (e.g. a dead lock, or a division by zero), the simulation is interrupted with the possibility to observe the state of the model, to run the **Analysis of Result** module on the data produced until the error occurred, and detailed information to fix the problem printed in the Log window.

[46] Lsd provides the time length of a simulation run, when this is finished. A known bug is the wrong computation of time in when Lsd runs in particular environment. However, this but does not affect the behaviour of Lsd models.

where users can set the step at which turn the debug mode on), the system interrupts the simulation as soon as the equation for one Variable tagged to be debugged is computed.

The interruption triggers a debugging session that permits to inspect the Lsd Debugger Window (Picture 10) where it is possible to:

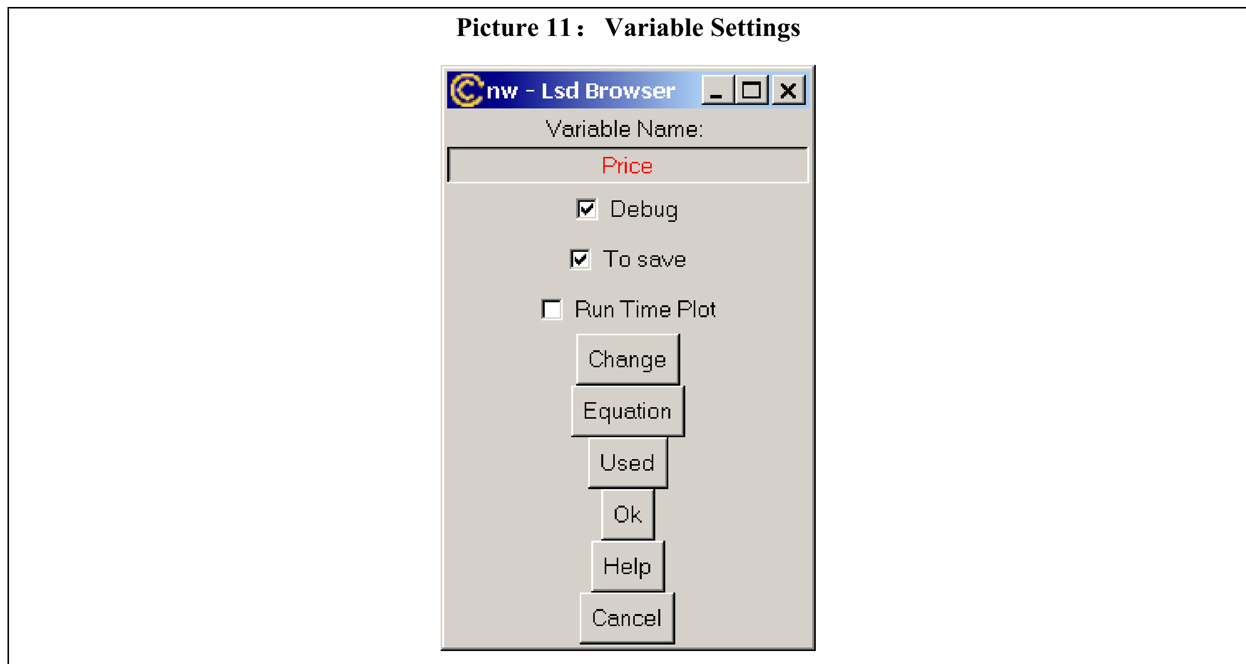
- observe any element of the model,
- modify their values,
- perform the analysis of the results up to most recent step completed,
- set on and off any debugging information,
- set conditional stops for the next debugging session.



Picture 10 : Lsd Debugger

Though the debugging facilities are thought to help model writers to remove errors from the model, it can be very useful for users who can analyse in detail any aspect of the model.

3.10 Options for saving, debugging and runtime plots



Picture 11 : Variable Settings

Before running a simulation users can change how Variables are treated during a simulation run, namely whether they need to be saved, debugged, and/or plotted in the run time plot. Using a "fresh" configuration, double click on the Variables' labels shown in the Browser window and you can control these options (see Picture 11[47]). As already mentioned, users can also obtain the equation setting the value of the Variable. Clicking on the button `Used` it is also possible to obtain a list of the equations of the model that make use of the values of the Variable of interest.[48] We shall return to these facilities in section 4.2.

4 MODIFYING AND BUILDING LSD MODELS

4.1 Becoming a Lsd programmer

Until now we have discussed the Lsd system from a model user's viewpoint. That is, we modified the numerical specifications of the model, but not its computational content. There are, however, sharply diminishing returns to the continued modification of parameters and initial variable values of a given model. We have seen that the Lsd version of NelWin allows us more space for modification than normally available using standard programming languages. Thus, the numbers of industries and firms were variables in Lsd sessions. However, besides changing the numerical specification of a model one

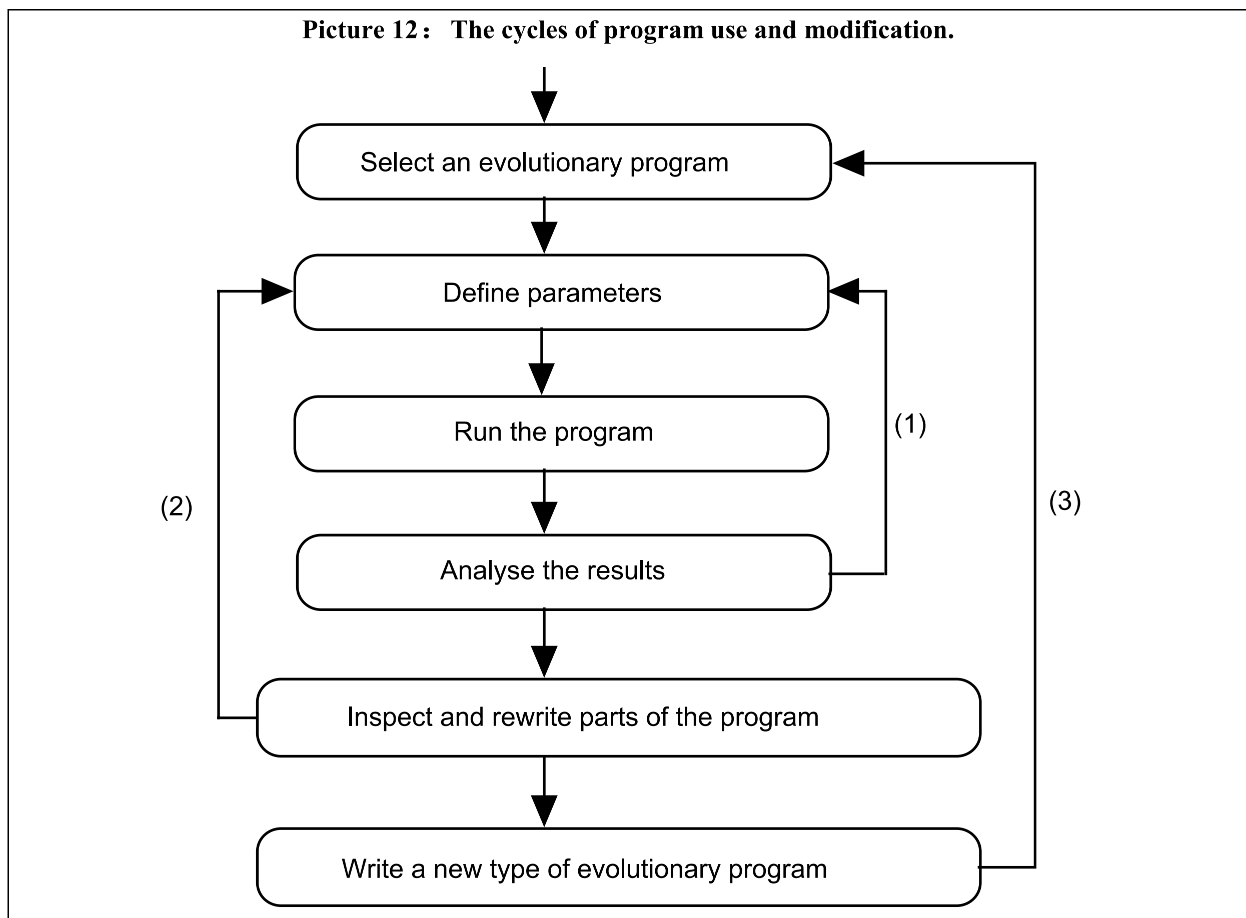
---

[47] Note that users have also access to change the very nature of the Variable (button `Change`). This should not be used by users since any change to such an element of the model in general needs a modification of the model's equations.
[48] This facility requires the name and the pattern of the equation files properly set. Use menu entry `File/Equation File` to change this, if required.

crucial step is still missing: often the most obvious reaction to the reflections about the simulation results is to revise your basic theory and behavioural model rather than just its parameters. That is, change the computational structure of the model, and not only its numerical values.

In ordinary programs this is normally practically impossible because of the sheer complexity of the task. In Lsd models like NelWin there are, however, plenty of tasks for reprogramming that can be easily performed. In this section we describe how you can quickly *start* exploring aspects of the art of programming. The trick is to relate to an existing model that is available to you both as a stand-alone program (like the Lsd model you are running) and as Lsd code text (that is source file for the equations of the model). We summarise your work with such a model in Picture 12. What we have done in the preceding sections is to follow loop 1. We started by selecting one of the supplied Lsd programs. Then we repeatedly defined parameters and initial values, performed the simulation, analysed the results, and defined interesting new experiments. Now it is time to follow loop 2. By reflecting on the results of our experiments, we define tasks for the rewriting of the model's code. When we have finished this rewrite and recompilation, we start another round of parameter experiments.

Picture 12 : The cycles of program use and modification.



While working with the rewriting of existing evolutionary programs like NelWin, we shall gradually find out that some changes are quite easy while other changes presuppose significant Lsd skills. Only the simple tasks will be covered in this paper. However, we shall have a glimpse of a more ambitious

task: given a series of gradual modifications of a type of model (like NelWin), we begin to understand its basic limitations. On this background we may define a more radical change—either within the Nelson–Winter tradition as suggested by Malerba et al. (1999) or by import of inspiration from one of the other traditions of evolutionary modelling (see e.g. Andersen 1996). However, there is quite a long way from defining a new family of evolutionary models to be able to program them effectively (loop 3 in Picture 12) and to persuade other researchers that the new type is worth while studying. The difficulty is, fortunately, diminished with Lsd, which allows you to distribute stand-alone examples of the new model type. Valente (1999a, 1999b, and 1999c) has developed several models in Lsd to substantiate this claim. In fact, Lsd has been explicitly designed from the start to provide the possibility to build models gradually and to re-use code. To be more specific, an equation can be imported into different models, where it can be adapted seamlessly even to a completely different computational structure. The precondition is the obvious one that the elements used in the equation (entities, Variables and Parameters) are present in the new model, even if they are computed in totally different ways.[49]

4.2 Specifying modifications of the NelWin model

To define tasks for a revision of the NelWin model, it is useful to return to the analysis of the standard model (see Table 4). Here we suggest two relatively simple changes:

- We change the innovative regime so that the cumulative progress takes a standard exponential form. By doing so we shall come closer to the original Nelson–Winter model.
- We split up the complex equation for capital accumulation according to the specification in Table 4. By doing so we shall be able to study separately the maximum gross investment and the desired investment.

These tasks are specified in Table 5. In equations 5a′ and 5b′ we use the inverse logarithmic and exponential functions. In C++ the natural logarithmic function is called `log` (and not ln), while the exponential function is called `exp`. First we take the natural logarithm of the mean of the normal distribution, and then we apply the exponential function to the normally distributed result. In this way we get rid of scale effects of the present version of NelWin that means that R&D becomes less and less productive over time. This change is relatively easy to perform since it only involves a minor change in the text of an existing equation of the NelWin program.

[49] The Lsd documentation (Model Reports) and the error catching facilities (e.g. signalling a missing element) allows an extremely easy and fast adaptation of the code even in different environments.

Table 5: Suggested modifications of NelWin (changes expressed with bold characters)	
5a.´ **Changed** innovative outcomes with Regime = 1	if (RANDOM(K[i,t-1]*RIN[i]*AN) AND Inn[i]=1) then A_IN[i,t] = **exp(**Normal(**log(**Mean_Prod[t-1]**)**, Std_Prod)**)** else A_IN[i,t] = 0
5b.´ **Changed** innovative outcomes with Regime = 2	if (RANDOM(K[i,t-1]*RIN[i]*AN) AND Inn[i]=1) then A_IN[i,t] = **exp(**Normal(**log(**A[t-1]**)**, Std_Prod)**)** else A_IN[i,t] = 0
10a.´ Maximum gross investment rate **by a separate equation**	if PROF[i,t] <= 0 then MaxInvestRate[i,t] = PROF[i,t] + Dep_rate else MaxInvestRate[i,t] = PROF[i,t]*(1 + Bank) + Dep_rate
10c.´ Desired gross investment rate **by a separate equation**	DesInvestRate[i,t] = Dep_rate + 1 – (Dem_elast / (Dem_elast – ms[i,t]))* C[i,t] / (K[i,t-1]*A[i,t])

The other task depicted in Table 5 does not change the meaning of the NelWin model, but it is more radical in programming terms. The task defined by equations 10a´ and 10c´ implies (1) creating two new variables (**`MaxInvestRate`** and **`DesInvestRate`**) in the NelWin model structure, (2) writing of the related equations, and (3) changing the original specification of the equation for **`K`**. It is obvious that these steps require much more care than the simple rewrite of existing equations.

4.3 The Lsd Model Manager as a core tool

Even though we just want to make minor rewrites of an existing Lsd model, we need to know a little about the writing and compiling of a Lsd model. A Lsd model program is composed by a part of code, also referred to as system code, common to every Lsd model (e.g. interfaces, file management, etc.), and another part which is model specific, the equations of the model contained in a single C++ source file. A Lsd model program is produced by the compiler linking the code from the general system code and from the model specific equations' file[50].

To produce a modified model, users just need to edit the equation file with the desired changes and (try to) run the Lsd model program. After that, LMM will take care to produce the model modified according to your editing and, in case, helping to fix the errors[51]. Before continuing, let's have a quick look at the LMM window (if you have close it, run it again from menu **`Start/Lsd Model Manager`**, or otherwise bring it in the foreground. For more details, look also at the LMM manual, available from menu **`Help/Help on LMM`**).

The LMM window appears just as a common editor: a main text page that can be filled with any text file, and a set of menus. However, LMM is a special editor in several respects. The menu **`Model`**

[50] Users can tell LMM how to compile a Lsd model program by means of two sets of options: Model Compilation Options and System Compilation Options. The first affects only the model, while the latter concern all models (see the LMM help pages on this topic for more information). These options are used to generate a makefile governing the compiler.
[51] Programmers will be interested in knowing that all LMM does is simply to run "make" with a makefile automatically prepared with the options set in LMM (see LMM help on Compilation Options). If no error occur during compilation LMM runs directly the Lsd model program. If errors occur, then the output of "make" is redirected in a new window.

allows users to manage the Lsd models installed on your system. For example, choosing **`Select Model`** you tell LMM to work on a specific model. When a model is selected, its name (and version number) appear on the bar just below the menu bar, along with the line/column currently pointed to by the text cursor. The menu **`Model`** offers all the operations that a user needs to work with a specific model. For example, besides running the Lsd model program, it allows to show the files relevant for the model (e.g. the file containing the equations), to change the compilation options, to run the GDB debugger[52] with the model, etc. The menu **`Edit`**, besides the usual cut/copy/paste, offers command of particular use to Lsd programmers. For a complete reference on all aspects of LMM, use the LMM help page, located in menu **`Help/Help on LMM`**. This page, in the usual HTML format, gives information on each individual aspect of the menu entries.

Sometimes, when changing the equations of a model, it may be useful to keep the original version, to be sure to not disrupt a working version. For these cases LMM offers the possibility to create verbatim copies of models, so that the changes do not affect the original version of the model. Therefore, the first operation we will do is to create a copy of the Nelson and Winter model to work on.

Open the menu **`Model/Select Model`** and select the Nelson–Winter model, if this is not already selected. After that, choose **`Model/Copy Model`** You are requested to provide a name, version number and directory location for the new model. Be sure to not provide names already existing in other models (LMM will warn if this happen, failing to overwrite previous models). When the model copy is successful created, open the menu **`Model/Show equations`**, which will fill your LMM editor page with the equations of the Nelson–Winter model copy, obviously identical to the original version, for the moment.

4.4 Lsd equations

The code for the equations must be thought of as individual and independent blocks of operations, each producing a numerical value, that 'float around' in the Lsd model program. When a simulation run is executed, the Lsd program checks which variable needs to be updated at the current time step. Then, it looks in its pool of available equations searching for one having the same name as the variable that needs to be computed. If this is found the program executes its code and assigns the resulting value to the variable, passing to the next variable or, if all variable have been updated, to the next time step.

[52] We have seen that Lsd model programs allow to interrupt a simulation at the end of the computation of a variable, in order to inspect the model status. The GDB debugger is the standard tool for C++ programmers: it allows to interrupt a program and to observe the behaviour of the program after one single line. Normally, Lsd users need the GDB debugger only to inspect the behaviour of very complex equations.

We now have to inspect the Lsd/C++ code for the equations of the NelWin model. In fact, the equations for a Lsd model are written as a purely C++ code, extended with a set of Lsd functions that modellers can use to perform the most common operation. The result is the mix of C++ and Lsd key words we have seen when inspecting the Equation Windows of Lsd. It is important to note that you—in the present version of Lsd—cannot change of the equation from the code window within a Lsd session. Instead you have to use a pure text editor for that purpose, like the one offered by LMM.[53]

The file with the equations for the Nelson–Winter model begins with several heading lines that you should not modify. After that there is the code for the equations listed in sequence of blocks.[54] The equations are written individually and are executed by each instance of the Variable to which they refer. For example, if there are many Objects **`Firm`** containing a Variable **`PROF`**, each of them will use exactly the same code, although, of course, some the 'parameters' of the equation (e.g. productivity **`A`**) will be firm specific.

The Lsd grammar for an equation is quite intuitive. As we said, an equation form a block that returns numerical value. Its internal structure may vary greatly depending on the complexity of the operations performed, but most of times it is composed by the following parts:

Table 6: Parts in an Equation Block

Part	Meaning	Example
Header	States the beginning of the block for the equation, indicating the Variable label to which it refers	`if (!strcmp(label, "A_IN"))` `{`

Comment	Comment the content of the equation, very useful to present the model to non Lsd experts.	`/**********************`
		`Equation computing new productivities for innovators`
		`**********************/`

Load required values	Most of equations are logical-mathematical elaborations over values taken from the model. In the equations these values are requested with Lsd specific functions (more on this later) and stored in temporary local variables v[0], v[1], etc.	`v[0] = p->cal("K", 1);`
		`v[1] = p->cal("RIN", 0);`
		`v[2] = p->cal("Inn", 0);`
		`v[3] = p->cal("Mean_Prod", 1);`
		`v[4] = p->cal("Std_Prod",0);`
		`v[5] = p->cal("AN", 0);`
		`v[6] = p->cal("A", 1);`
		`v[7] = p->cal("Regime", 0);`

[53] This is a crucial aspect, which novice programmers may have some troubles to grasp. The equations code is transformed by the compiler in a program that is embedded in the Lsd model program, which will use a variable equation any time it needs to compute the value for such a variable. It is therefore useless for the Lsd model program to modify the equations, since these need to be re-compiled to create a new Lsd model program.

[54] The order in which the equations are written does not affect at all the actual order they are executed during a simulation.

Equation C++ code and final result	The core of the equation is the C++ expression of the elaboration required, with the result assigned to the local variable “res”, whose value will be assigned to the Lsd Variable in the model.	`if (v[2] == 0)`
		`  res = 0;`
		
		`if(v[2]==1 && (RND<(v[0]*v[1]*v[5]) ) )`
		`  {`
		`    if (v[7] == 1)`
		`      res = norm(v[3],v[4]);`
		`    else`
		`      res = norm(v[6],v[4]);`
		`  }`
		`else`
		`  res=0;`

Equation ending	Indicate that the equation block is terminated.	`goto end;` `}`

Therefore, a block for an equation appears as reported below:

```
if (!strcmp(label, "A_IN"))
 {

   /***************************
      [comment deleted]
   ***************************/

v[0] = p->cal("K", 1);
v[1] = p->cal("RIN", 0);
v[2] = p->cal("Inn", 0);
v[3] = p->cal("Mean_Prod", 1);
v[4] = p->cal("Std_Prod",0); //The fixed standard deviation
v[5] = p->cal("AN", 0);
v[6] = p->cal("A", 1);
v[7] = p->cal("Regime", 0);

if (v[2] == 0)
  res = 0; //no innovation

if( v[2] == 1 && (RND < (v[0]*v[1]*v[5]) ) )
  {
    if (v[7] == 1)
      res = norm(v[3],v[4]); // Industry cumulation
    else
      res = norm(v[6],v[4]); // Cumulative at firm level
  }
else
  res=0; //no innovation

goto end;
}
```

The first line is used to mark the beginning of the block for an equation (in this case, it is the equation for Variable **`A_IN`**). Users willing to locate the equation for a Variable can therefore search for the name of the Variable, skipping any occurrence until a line like the first one above includes the name of the desired Variable. Usually, the first line within the block for an equation is a comment, with no meaning for the compiler, helping readers to understand what the equation does. In the example above

the comment has been removed—to avoid distracting attention from the C++/Lsd code[55]. The code starts by assigning values to seven temporary variables (**v[0], ..., v[7]**). Each Lsd equation can make use of the temporary variables **v[0]** up to **v[39]** (more can be obtained, if necessary), which store intermediate values during the computation of the equation. The seven assignments have an identical structure: the value of an element is requested by the system and assigned to a temporary variable. The requested variables/parameters are: **K** (lagged variable), **RIN** (unlagged parameter), **Inn** (unlagged parameter), **Mean_Prod** (lagged variable), **Std_Prod** (unlagged parameter), **AN** (unlagged parameter), **A** (lagged variable), and **Regime** (unlagged parameter). The Lsd function to request all these values to the model is, as you see:

```
p->cal("VarX", lag)
```

This function returns the value of VarX in period t – lag, where t is the current time of the simulation step.

When the values have been collected, the system performs a computation with the temporary variables. In the present case we have two conditional C++ expressions (in the somewhat arcane C++ syntax). The first conditional expression asks whether **Inn** = 0 (here C++ uses identity **==,** rather than assignment =). If this is the case, then equation result '**res**' is assigned to 0. Then it is checked whether two conditions are fulfilled: (a) a newly generated random number must be less than capital times research intensity times probability of success, and (b) the firm must be performing innovative research. If these conditions are fulfilled, the equation result '**res**', that is, the new value for the Variable **A_IN**, is drawn from a normal distribution (**norm**). As previously mentioned there are two alternative computations, depending on the value of **Regime**. It is obviously these computations that we want to change. In the case that the parameter is equal to 1, then the mean of the normal distribution is taken from the industry mean productivity; otherwise it is taken from the firm's own productivity. If the two conditions are not fulfilled, the result is set to zero (no innovative result). You should note that the lsd internal variable **res** is not a Lsd model Variable. It takes the value that later the system delivers to the actual Lsd Variable for that time step. Finally, the Lsd system is told that it has found the correct equation so that it should go to the end of the equations instead of continuing to check the list of equations sequentially.

All Lsd equations have this structure. They begin with a heading line signalling the Variable it refers to. At the end, the equation *must* contain an assignment to the '**res**' internal variable, which is used to

---

[55] However, we have kept two inline comment that indicates the expression to be changed. Note that the commenting lines appear in different colour, to be differentiated from actual 'active' code

pass the value to the actual Lsd Variable which the equation refers to, and the key-word '**goto end**;'. In between the heading line and the last two lines modellers can place any C++/Lsd code.[56]

Let us now turn to the particular Lsd syntax. The expression

```
p->cal("Label",n);
```

is the typical Lsd command for obtaining the value of Variable or Parameter called Label with lag **n**.

The symbol '**p->**' indicates that the Lsd function is executed by the same Object where the Variable computed is stored. This ensures that, if many instances contain the same Variables, the correct ones are used. In our example **A_IN** is stored in **Firm**, meaning that any copy of **A_IN** will be computed with the value of the copy **K** stored in the same copy of object **Firm.**[57]

The 'cal' function is one of the Lsd functions that modellers can add to any legal C++ code to express the equation's computation. It simply provides the value of 'Label' stored in '**p->**' or in other Objects connected to **p->**. For example, in the equation described above, any Object **Firm** contains a Variable **K**, a Parameter **INN**, etc. On the other hand, also **A_IN** is a Variable contained in any **Firm**. The use of **p->** means that the value, for example, of **K** must be placed in the same Object as the **A_IN** which is computed. Therefore, the equation will use the values that the modeller intended.

Actually, '**p->cal**' does something more than what said above. Consider the line saying:

```
v[3] = p->cal("Mean_Prod", 1);
```

Actually, **Mean_Prod** cannot be found in any **Firm** at all, since it is a Variable contained in Object Industry. How does **p->cal** behave in this case? The general rule is the following:

- if the search cannot find the desired Variable in the same Object, search in the descending (contained) Object containing the Variable under computation (**A_IN** in our example);
- if the search still fails, search in the upper Object than the one containing the Variable.

These rules are applied recursively, so that the whole model is explored looking for a given element (of course, the implementation prevents the research to return in already explored Objects). Note that the result of the rules is quite intuitive: in our example, the second step finds **Mean_Prod**, because an Object Industry is found. Moreover, consider the case that the user has created many Objects **Industry**. The same identical line of code ensures that, among the many Variables **Mean_Prod**, the

[56] In theory, one may even write a full C++ program as a 'Lsd equation', although it may not make much sense. But this means that modellers can copy within an equation block every kind of code, for example using libraries from the public domain.
[57] Modellers have several possibilities to get values from different objects, if required. However, Lsd has an automatic system to search for values in the model starting from each Object, so that almost always the object **p->** is the correct one to use.

correct one is used, that is, the copy stored in the Industry containing the **`Firm`** whose **`A_IN`** is executed.

The lag is always expressed as a relative time lag. For example, in the equation showed above, all the parameters were of course requested with lag 0, while there were lagged values for **`K`** and **`Mean_Prod`**, as indicated by the use of 1 in the place of lag. The use of lags permits modellers to totally control the relative time ranking of the computations for the different equations only when necessary, but does not force on them the tedious job of specifying the exact scheduling for the whole set of equations. For example, you can check yourself (e.g. with the Lsd NelWin automatic report) that the model states that **`A_IN`** uses the value of **`K`**, that **`A`** uses the value of **`A_IN`**, and, finally, **`K`** uses **`A_IN`**. Of course, this circularity requires that somewhere a lag is introduced. During a simulation run the Lsd system automatically execute the equations updating the variables in the correct order, as specified implicitly in the lags' structure.

It is worth stressing that the execution order for the equations is, in general, a major problem when writing simulation models. In fact, designing a model one tends to think to each equation in isolation and does not have a clear understanding on the effects of changing the order of execution for the different equations. Moreover, things get easily very complicated when the number of variables becomes even slightly large. This means that you risk realising that the order of execution in your model, resulting from a series of more or less casual choices, is wrong. But the revision of this aspect is one of the most difficult things to do with normal languages, and often implies to rewrite the program from the scratch. Although it can appear slightly confusing at the beginning, the Lsd lag management is extremely useful for this purpose. In fact, it requires modellers to specify only the 'local' order within each equation independently from the others, while the system takes care of finding the overall global order of execution. Moreover, the modification of a previous choice can be made by simply replacing the lag notations, and is supported by reliable controls over their consistency.

There are many other Lsd programming facilities for modellers simplifying the task of writing simulation programs, but for our present purposes the above explanation is sufficient. Just take notice that in any standard programming language the above information for an equation would not be accepted. In fact, we have a model with many Firms, and therefore many **`K`**s, **`A`**s etc., but the equation does not specify explicitly which one must be used in the equations. The Lsd grammar exploits the default Lsd system that induces automatically which instance to use. Moreover, the induction takes place only at run time, when a simulation is actually run. Therefore, exactly the same code above can be moved in a different model, and it would work anyway even if the elements used (e.g. **`Mean_Prod`** and **`K`**) were placed in different Objects. This ensures a very simple re-usability of the code through different models.

4.5 Compiling the model with a modified equation from LMM

As should be clear by now, to modify the computational behaviour of a model (as opposed to the numerical changes), you need to modify the source code and compile a new Lsd model program. Before that, however, it is useful to make sure that you remember the behaviour of the original NelWin program with respect to productivities. To do so, start NelWin, select **File/Load**, load **nw.lsd**, select **File/Save**, write **nw2.lsd**, and click **Save**. Now we double click **K (1).** In the small options window we deselect **Run Time Plot,** and press **OK**. Back in the Browser, we double click **A (1),** select **Run Time Plot**, do not change the setting of **To save**, and press **OK**. The effect of this change is that productivities are plotted during the simulation run, instead of the capitals **K**. Then we select **Data/Init. values**, set all **Inn** = 1, and press **OK**. Finally, run the program and inspect its behaviour with respect to productivities.

Now we come to the most important part of the demonstration, the modification of the ready-made version of NelWin. In the Lsd Model Manager open the equations' file for the Nelson and Winter model (menu entry **Show Equation** in menu **Model** when Nelson and Winter model is selected) and locate the block for **A_IN** beginning with the line:

```
if (!strcmp(label, "A_IN"))
```

(you can use the entry **Search** in menu **Edit** typing **A_IN** and then pressing F3 anytime you find an occurrence which is not the desired one).

In the equation you should change in the equation for **A_IN** quoted above the line:

```
  res = norm(v[3],v[4]); // Industry cumulation
else
  res = norm(v[6],v[4]); // Cumulative at firm level
```

to:

```
  res = exp(norm(log(v[3]),v[4])); // Industry cumulation
else
  res = exp(norm(log(v[6]),v[4])); // Cumulative at firm level
```

Save the file (button **Save**) and you are ready to compile the modified version of the Nelson and Winter model. Note that when you choose menu **Model/Run**, it took a shorter time to produce the program than before. Also in the results of the compilation you see that the compiler worked only on the equation file, but not on the Lsd system files. This is because it finds the equation file is more recent than the existing Lsd model program, and therefore updates it, while the other files were not changed, and their compiled version could be re-used.

After having launched the newly created Lsd model program, you should load the **nw2.lsd** file and run the simulation. The result was obviously different from the result obtained with the NelWin model that we have already studied. (Exercise: Why is it so?)

4.6 Compiling the model with a new equation

Changes of the Lsd equations often imply the introduction of new objects and/or new variables and parameters. This means that there will be a change not only in the file with Lsd equations (**`fun_nw.cpp`**) but also in the file with the data and the Lsd model structure (**`nw2.lsd`**). We have already seen how the first of these files can be changed with an editor program. The second file can, however, be modified more easily from the Lsd Browser window.

In Table 5 (and the related text) we have already sketched out a task that requires the creation of a new variable and a related equation: to decompose the equation that calculates the value of **`K`** into several intermediate equations. Let us concentrate on specifying the equation that computes the maximum change per unit of capital, **`MaxInvestRate`**.

We start with the issue of writing the equation for **`MaxInvestRate`**. Make sure that NelWin is loaded into LMM, and select **`Model/Show Equation`**. Then move to the definition of **`K`**. Click in the empty line before **`if(!strcmp(label, "K"))`**, add an extra line[58], choose **`Edit/Insert Lsd scripts`**, click **`Lsd Equation`** and **`Insert`**, write **`MaxInvestRate`**, and press **`OK`**[59]. Now you see that an equation template is inserted. In the comment area delete 'Comment' and write instead something like 'This is the first part of a decomposition of NelWin's K equation.' Finally change **`res=;`** to **`res=0;`** (zero!). Now you have a do-nothing version of **`MaxInvestRate`**. To make sure that everything has been done correctly, choose **`File/Save`** and **`Model/Run Lsd Model`**. After the compilation choose load **`nw2.lsd`**, **`Run/Run`**, and make sure that everything is working as before. In fact, we have added a new equation (a boring "equation": just returns 0…), but we did not include a new Variable in the model structure, and therefore Lsd simply ignores that code.

Before you come to a improve the model's code, you might meet error messages from the C++ compiler. In fact, the equation is part of a C++ program and therefore it must respect the C++ grammar. Errors referring to typing mistakes that violate the C++ grammar are rather frequent, and therefore it is good practice, even among expert programmers, to "compile" the program every few lines inserted, so that any error can be easily found. Normally, the compiler provides indications on the line number where the error occurred (e.g. you forgot the semicolon at the end of a line). If this happens, write down the line where the error (or the first error) occurred. Open again the equation file, and go to that line (shortcut Ctrl+l) and check carefully the lines you added. Note that the error messages issued by the compiler appear in a new window. If you close this window, you can re-open it just choosing **`Model/Show Compilation Result.`**

---

[58] Of course, the block of code for an equation cannot be located within the block for another equation.

To perform the next step, you have to inspect the rather complex code for the **K** equation:

```
if(!strcmp(label, "K"))
{
// Comment deleted
  v[0]=p->cal("K", 1);
  v[1]=p->cal("Price", 0);
  v[2]=p->cal("PROF", 0);
  v[3]=p->cal("A", 0);
  v[10]=p->cal("ms", 0);
  v[6]=p->cal("Bank",0);
  v[7]=p->cal("Dep_rate",0);
  v[8]=p->cal("Dem_elast",0);
  v[15]=p->cal("C",0);

  if(v[2]<=0)
   v[2]=v[2]+v[7]; // Only internal financing
  else
   v[2]=v[2]*(1+v[6])+v[7]; //Also external financing

  v[9]=v[15]/(v[1]*v[3]); //Relative Mark-up

  v[11]=v[7]+1-v[8]/(v[8]-v[10])*v[9]; //Desired investment rate

  v[12]=max(0, min(v[11], v[2])); //Final investment rate

  res=v[0]*(1-v[7]+v[12]); // New capital stock

  goto end;
}
```

The relevant part of the equation is the first if-else-sentence and the related temporary variables. Copy the text from **v[2]=p->cal("PROF",0);** to '**Also external financing**', paste this text into your new equation immediately after the endmark of the comment, delete the temporary variables that you are not using, and change the last two occurrences of **v[2]** to **res**. Now the equation should look like.

```
if(!strcmp(label, "MaxInvestRate"))
{
// This is the first part of a decomposition of NelWin's K equation.
  v[2]=p->cal("PROF", 0);
  v[6]=p->cal("Bank",0);
  v[7]=p->cal("Dep_rate",0);

  if(v[2]<=0)
    res = v[2]+v[7]; // Only internal financing
  else
    res = v[2]*(1+v[6])+v[7]; //Also external financing

  goto end;
}
```

Now you have written your first effective Lsd equation. However, to have this code as an active part of a simulation run we need to define the Lsd variable that uses it, that is, to include **MaxInvestRate** in the Lsd structure file. To do so we start the NelWin Lsd model, load **nw2.lsd**, and in the Lsd Model Structure window double click on **Firm**. Then choose **Model/Add a Variable** in the Lsd

---

[59] LMM offers several "wizards" to include in the text the most frequently used Lsd code, like an (empty) block for an equation, the function p->cal(…), and many others. This system simply users to make typing errors, and the resulting text

Browser, write **MaxInvestRate**, and click **OK**. Furthermore, you should double-click **MaxInvestRate** (at the bottom of the list), choose **To Save**, and click **OK**. With this operation you have prepared the NelWin model with a new Variable, **MaxInvestRate** and told that you want to observe its values in the analysis of results.

Now you can run the simulation model, and inspect the results of **MaxInvestRate** in the Data Analysis window. When this is done (and errors have been removed[60]), we may turn to the rewrite of the **K** equation. In fact, our new variable duly computed its values, but they were not used in other parts of the model. Now we modify the equation for **K** so that is exploits the computation done in **MaxInvestRate**.

There is no need of checking for redundant calls for values and for the aestetics of e.g. the numbering of temporary variables (such changes might add new errors). Just make the minimal changes as suggested below (the deletions are obvious; the additions are marked with **bold**):

```
if(!strcmp(label, "K"))
{
// Comment deleted
  v[0]=p->cal("K", 1);
  v[1]=p->cal("Price", 0);
  v[2]=p->cal("PROF", 0);
  v[3]=p->cal("A", 0);
  v[10]=p->cal("ms", 0);
  v[6]=p->cal("Bank",0);
  v[7]=p->cal("Dep_rate",0);
  v[8]=p->cal("Dem_elast",0);
  v[15]=p->cal("C",0);

  v[22] = p->cal("MaxInvestRate",0); // Call of the new variable

  v[9]=v[15]/(v[1]*v[3]); //Relative Mark-up

  v[11]=v[7]+1-v[8]/(v[8]-v[10])*v[9]; //Desired investment rate

  v[12]=max(0, min(v[11], v[22])); //Final investment rate

  res=v[0]*(1-v[7]+v[12]); // New capital stock

  goto end;
}
```

This change completes the exercise. Of course you should immediately make sure that everything works correctly, i.e. that you have really stored **MaxInvestRate** into variable 22. To make sure that you master the procedure, you should now do exactly the same for **DesInvestRate**. Here is, however, a snag. You should copy the code for both the relative mark-up factor and for the desired

---

introduced in the equation file can be edited like any other character.

[60] Besides errors in the equation code, Lsd users must pay attention to the exact correspondence between the names of variables in the code and the names of the same variables in the Lsd model structure. If, for example, you have inserted in the model the variable **maxinvestrate** then Lsd will complain, since no equation with that spelling is found in the equation files. To fix these kinds of errors double-click on the name of the misspelled variable, click on "change" and type the correct label.

investment rate into your new equation. The relative mark-up continues to be an internal variable, while `v[11]` should be changed to `res`.

4.7 Running an experiment with the a Lsd model

Running simulation models is useful in order to compare the dynamical patterns developing from different initializations. We have seen that Lsd offers a very simple way to modify the initial values of a model. However, there are also other features in Lsd models worth to be mentioned.

Suppose one wants to test the effects of the parameter **`Regime`**, allowing for industry- or firm-level cumulative knowledge. We may run two separate simulation runs with the two values for **`Regime`**, remembering the first exercise's results and comparing with the second, or we may run two different instances of the Lsd model program. In both cases we have troubles, either for the imprecision or for the large number of windows on our screen. Furthermore, simulation models frequently depend on random events, and we would like to repeat our exercise a large number of times, in order to be sure that are results are not due particular and rare combinations of random events. In the following it is described a Lsd exercise on the Nelson and Winter model where the two regimes are tested "in parallel" and the exercise is automatically repeated many times to be tested for robustness.

Load a fresh model file (press Ctrl+w to automatically remove the existing model and load again the data from the same Lsd model file). Then chose the menu command **`Data/Set Number of Obj./All types of Obj.`** In the resulting window, click on the number 1 close to the line '**`# of Industry`**', insert 2 and click **`Ok`**. Now you see that the model contains two Objects **`Industry`**, the second being an exact copy of the first, original, one.

While being in the window showing the number of Objects (after having asked to show the hierarchical level 2, **`Firm`**'s included) you can control the initial values for the Objects by clicking on the labels of the Objects. When you check the initial values for Industry set the value of **`Regime`** in the first column to 1, and 0 in the second.

Now you have the Lsd model loaded containing two Objects **`Industry`**, identical but for their regime's flag, containing 4 **`Firms`** each, evenly divided between imitators only and innovators. You can save the file with a new name, for example **`Regime.lsd`**, to not overwrite the previous Lsd model file (and to recover from any mistake you may have made). The last step is to ensure that the simulation results are shown as required. To do this, press the keys Ctrl+P (= menu **`Run/Remove Plot Flags`**), which will remove any variables set to be shown in the Run Time Plot.

Random events do matter in the simulation in a cumulative way, since early fluctuations affect the productivities which affect the capital stock which, in turn, determine the probability to have further

innovations. To make a test against the robustness of the results select the menu **Run/Sim.Setting**. Here choose to make 100 simulation runs, each of 500 time steps. Now start the simulation. Notice that the summary window asking confirmation before running the simulation will give you a lot of information: it reports the names of the files where the data will be stored for each of the simulation runs individually, and for the latest steps from each simulation in a single file (see below).

During a simulation run you will see in the Log window to appear one line for each stem computed[61]. Click in this window the button **Fast** to avoid these useless messages.

The system is now computing 100 simulations, for each of them storing the results in a single file reporting the whole series saved for that simulation run. Moreover, the values at the final step of each simulation run are stored in a Total Result file. After the end of the simulation session, open the Analysis of Results module, with the menu **Data/Analysis Result**. Choose **File** and look for the file with extension '.tot' (it is easier to specify the extension 'tot' in the scroll list 'Files of Type') and open it. You will see the standard window for the Analysis of Results, containing 100 'steps'. Beware that these values are actually not steps of a single run, but the last steps from the different simulations, and therefore the 'run time plot' in this case does not express a dynamics, but shows only the values reached in each simulation. Click on **Sort** to rank alphabetically the Variables, and select all the '**A**' (from **A_1_1** to **A_2_4**) and then click on '**>**' to move them in the '**Series Selected**' list. Now click on **Statistics** and observe the Log window. You will see the average value for each of the Variables, complete with other relevant descriptive information, to test for the robustness of your results.

You can proceed making further changes of the NelWin model. However, you will soon come to a point where you'd prefer to create a new model rather than modify an existing version. For doing this, just click on the button **New** in LMM. You will have a new model ready to be compiled, even though it cannot do anything. In fact, inspecting the equation files for a just created model, you see that there is only one example block for an equation, doing nothing but computing 0. However, this is still a perfectly functioning Lsd model program, once compiled, unless you try to make a simulation run. To build the model you will need to add Objects, Variables and Parameters, using the Lsd interfaces, and to define the equations' using the LMM editor in the equation file. Note that many calculations you may want to do in a new model may be similar to the ones contained in an existing model. Feel free to 'steal' the code from other models, which can be done copying the equation blocks (in this case, be careful to include in the selection the closing graph bracket, to avoid the compiler being wild).

---

[61] If this does not happen, then you likely have forgot to remove the Run Time Plot flags, and each simulation will create a new Run Time Plot window. Prepare to have your screen hosting 100 of them! To avoid this, simply stop the simulation run, reload the configuration (CTRL+W), remove the Run Time Plot flags (CTRL+P) and run the simulation again (CTRL+R).

It is likely that for sophisticated models you need to consult the Lsd documentation. This documentation is e.g. found in Valente (1999b and 1999c). Much material is also available as (not totally updated) web documents at the Lsd web site.

5 CONCLUSIONS AND PERSPECTIVES

In the present paper we have given a hands-on introduction to the Lsd system for both users and programmers of evolutionary models like NelWin. Actually, we have demonstrated that this distinction is somewhat artificial. Traditionally, the role of the user is to try out a model by changing parameters and initial values. But with a little help users can also experiment with changing the some of the core equations in the model, and thereby they are already beginning to overcome the barrier of entry to the area of programming. More importantly, they can begin to utilise their informal knowledge about evolutionary processes within the Laboratory for Simulation Development. This does not mean that programming skills can be avoided when developing and implementing research-oriented evolutionary simulation models, but it might encourage new entries to the field.

The paper has only implicitly discussed how Lsd improves the situation seen from the viewpoint of researchers. It is, however, obvious that the communication problems created by the division of labour between general modellers and specialists in computer implementation of these models can be diminished (described in section 1.1). To some extent modellers like Nelson and Winter and their followers are users of the simulation models provided by the experts in e.g. C++ programming. By means of Lsd these advanced users can prepare themselves for discussions with the computer specialists, and in some cases the intervention of these specialists can be avoided altogether. In a counterfactual way we can start to speculate what might have happened to the Nelson–Winter tradition if it had started with 'NelWin and Lsd' instead of 'NelWin and FORTRAN'. It is obvious that in this speculative alternative, there would have been a much more experimentation and a much weaker tendency towards lock-in with respect to model specifications and even parameter settings.

Another aspect of the Lsd alternative is that the distribution of NelWin would have been much easier. This would have implied that more researchers would more quickly do the testing and debugging of the models. This would have given a feed-back to the programmers since sloppy programming and obvious errors would have be revealed and praise would, hopefully, have been given to high-quality programming. Thus the establishment of a research community dealing collaboratively with the simulation models would more easily have emerged. In such a community that makes exchanges of ideas of programs under academic norms (and the GNU General Public License) a cumulative process of improving the models can take place. The standardised construction of Lsd models and the automatic generation of Lsd model reports that are ready for being placed at the web sites of the Internet would have given few excuses for not joining making exchanges. Given that researchers had

joined this exchange, they would less likely to use idiosyncratic model specifications. Similarly, editors of scientific journals might have felt entitled to demand public access to the underlying code, and it would have been much easier to create university courses in evolutionary programming and simulation.

We cannot change the past of evolutionary modelling and simulation, but our counterfactual speculations about the past might suggest important possibilities in the near future.

REFERENCES

Andersen, E.S. (1996). *Evolutionary Economics: Post-Schumpeterian Contributions*, paperback edn. London: Pinter/Cassell.

Andersen, E.S. (1999). Multisectoral Growth and National Innovation Systems. *Nordic Journal of Political Economy*, 25, 33–52.

Andersen, E.S. (2001a). Toward a Multiactivity Generalisation of the Nelson–Winter Model. Paper presented at DRUID's Nelson–Winter Conference, 12–15 June 2001. `http://www.business.auc.dk/druid/conferences/nw/paper1/andersen.pdf`.

Andersen, E.S. (2001b). Satiation in an Evolutionary Model of Structural Economic Dynamics. *Journal of Evolutionary Economics*, 11, 143–164.

Andersen, E.S., Jensen, A.K., Madsen, L., and Jørgensen, M. (1996). The Nelson and Winter Models Revisited: Prototypes for Computer-Based Reconstruction of Schumpeterian Competition. DRUID Working Papers 96–5, Danish Research Unit for Industrial Dynamics, Aalborg University.

Andersen, E.S. and Valente, M. (1999). The two software cultures and the evolution of evolutionary economic simulation. Danish Research Unit for Industrial Dynamics, Aalborg University. `http://www.business.auc.dk/evolution/esapapers/esa99/AndVal.pdf`.

Chiaromonte, F., and Dosi, G. (1993). The Micro Foundations of Competitiveness and their Macroeconomic Implications. In D. Foray and C. Freeman (eds.), *Technology and the Wealth of Nations: The Dynamics of Constructed Advantage*. London: Pinter.

Conte, R., Hegselmann, R., and Terna, P. (eds.) (1997). *Simulating Social Phenomena*. Berlin: Springer.

DiBona, C., Ockman, S., and Stone, M. (eds.) (1999). *Open Sources: Voices from the Open Source Revolution*. Sebastopol, Calif.: O'Reilly.

Epstein, J. M., and Axtell, R. (1996). *Growing Artificial Societies: Social Science From the Bottom Up*. Boston, Mass. and London , MIT Press.

Fogel, K. (1999). *Open Source Development with CVS*. Scottdale, Ariz.: Corolis.

Gilbert, N., and Troitzsch, K.G. (1999). *Simulation for the Social Scientist*. Buckingham: Open University Press.

Kwasnicki, W. (1996). *Knowledge, Innovation, and Economy: An Evolutionary Exploration*. Aldershot: Elgar.

Malerba, F., Nelson, R.R., Orsenigo, L., and Winter, S.G. (1999). 'History-friendly' Models of Industry Evolution: The Computer Industry. *Industrial and Corporate Change*, 8, 1–36.

Nelson, R. R., and Winter, S. G. (1974). Neoclassical vs Evolutionary Theories of Economic Growth: Critique and Prospectus. *Economic Journal*, 84, 886–905.

Nelson, R.R., Winter, S.G., and Schuette, H.L. (1976). Technical Change in an Evolutionary Model. *Quarterly Journal of Economics*, 90, 90–118.

Nelson, R.R., and Winter, S.G. (1977). Dynamic Competition and Technical Progress. In B. Balassa and R.R. Nelson (eds.), *Economic Progress, Private Values, and Public Policy: Essays in Honor of William Fellner*. Amsterdam: North-Holland.

Nelson, R.R., and Winter, S.G. (1978). Forces Generating and Limiting Concentration under Schumpeterian Competition. *Bell Journal of Economics*, 9, 524–548.

Nelson, R.R., and Winter, S.G. (1982a). *An Evolutionary Theory of Economic Change*. Cambridge, Mass. and London: Belknap Press.

Nelson, R.R., and Winter, S.G. (1982b). The Schumpeterian Tradeoff Revisited. *American Economic Review*, 72, 114–132.

Ousterhout, J.K. (1994). *Tcl and the Tk Toolkit*. Reading, Mass.: Addison-Wesley.

Silverberg, G., Dosi, G., and Orsenigo, L. (1988). Innovation, Diversity and Diffusion: A Self-Organization Model. *Economic Journal*, 98, 1032–1054.

Silverberg, G., and Verspagen, B. (1994). Collective Learning, Innovation and Growth in a Boundedly Rational, Evolutionary World. *Journal of Evolutionary Economics*, 4, 207–226.

Silverberg, G. and B. Verspagen (1998a). Economic Growth and Economic Evolution: A Modelling Perspective. In F. Schweitzer and G. Silverberg (eds.), *Evolution and Self-Organization in Economics*, Jahrbuch für Komplexität in den Natur-, Sozial und Geisteswissenschaften, Band 9. Berlin: Duncker & Humblot.

Silverberg, G., and B. Verspagen (1998b). Economic Growth as an Evolutionary Process. In J. Lesourne and A. Orléan (eds.), *Advances in Self-Organization and Evolutionary Economics*. London: Economica.

Sterman, J.D. (2000). *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Boston, Mass.: McGraw-Hill.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Reading, Mass.: Addison-Wesley.

Stroustrup, B. (2000). The C++ Programming Language, Special Edn. Boston, Mass.: Addison-Wesley.

Valente, M. (1999a). Consumer Behaviour and Technological Complexity in the Evolution of Markets: Volume I of Evolutionary Economics and Computer Simulation: A Model for the Evolution of Markets. PhD Thesis, Department of Business Studies, Aalborg University. `http://www.business.auc.dk/~mv/ThesisHome/Volume_I.pdf`.

Valente, M. (1999b). Laboratory for Simulation Development—A Proposal for Simulation in the Social Sciences: Volume II of Evolutionary Economics and Computer Simulation: A Model for the Evolution of Markets. PhD Thesis, Department of Business Studies, Aalborg University. `http://www.business.auc.dk/~mv/ThesisHome/Volume_II.pdf`.

Valente, M. (1999c). Lsd Model Reports: Volume III of Evolutionary Economics and Computer Simulation: A Model for the Evolution of Markets. PhD Thesis, Department of Business Studies, Aalborg University. `http://www.business.auc.dk/~mv/ThesisHome/Volume_III.pdf`.

Welsh, M., Dalheimer, M.K., and Kaufman, L. (1999). *Running Linux*. Sebastopol, Calif.: O'Reilly.

Winter, S.G. (1984). Schumpeterian Competition in Alternative Technological Regimes. *Journal of Economic Behavior and Organization*, 5, 287–320.

APPENDIX A: LSD GRAMMAR FOR EQUATIONS

A1 Lsd equations and C++

Modellers should think of the equations in a Lsd model individually, just like difference equation models. The equations are put in an arbitrary order in a single file, which is a pure C++ source code (with Lsd extensions). Modellers do not need to get concerned with any other aspect of the equation file, besides the equations.

A good modeller will put helpful comments into the definitions of the equations. In C++ there are two types: any text in a line that comes after // is a comment; any text between /* and */ is a comment. Both types of comments are ignored by the compiler.

An equation in the equations' file of a Lsd model is a block like:

```
if (!strcmp(label, "Var"))
 {
   /***************************
      Comments on the computation of the variable Var
      Here we just take Var = 1 + 1
   ***************************/

   v[0] = 1 + 1; // the code to compute Var
                 // Here: put the result of 1 + 1 into
                 // the temporary variable v[0]

   res = v[0];      // the value for Var at the generic time t
                    // is assigned to res
   goto end;  // ending the equation

 } // Don't forget the closing bracket
```

An equation can contain any C++ legal code. In C++, each command must terminate with a semicolon.

The two most frequently constructions are the **`if`** and the **`for`** control flows:

```
if (CONDITION)
 {
   // code executed if CONDITION is true
   // CONDITION is a logical construction like <, >, ==, !=
   // by convention, any value different from 0 is equivalent to TRUE
   // and any value equals to 0 is false
 }
else
   // a single line of code can be placed without brackets

for (i = 0; i < 40; i++)
 {
   // Repeat 40 times with i = 0 at the first time and i = 40 the last.
   // The first field (where is i = 0) can contain any command which is
   // always executed when the line with the for.. is reached. More
   // commands can be placed in the field, separated by commas ",".
   // The second field (i < 40) must contain a condition that, as long
   // it is true, makes the block within brackets to be computed again.
   // The last field (i++) is executed after each cycle before the
```

```
   // condition is controlled; more commands can be separated by commas
   // ",". The i++ command is equivalent to i = i + 1.
 }
```

Note that in C++ the graph brackets are very important, since they determine the blocks of code, for example for the IF and the FOR. Forgetting one bracket can produce errors very difficult to find. Note that the LMM editor allows determinining the matching brackets, which is very handy, particularly when many nested blocks make confusing their identification.

A2 Equations in Lsd models

There are two aspects that usually make complicated to write simulation programs: the order of execution of equations and the dealing with many elements with the same label.

Concerning the first problem, Lsd modellers do not need to consider the overall scheduling of all the equations, but only the relative order between related Variables. If the modeller wants to use in an equation the value of another Variable, there is no need to worry whether this has been already computed or not. Lsd will take care of finding out the correct order of execution ensuring that the order implicitly defined with the lags will be respected. If the model's code is temporally inconsistent—e.g. $X_t = f(Y_t)$ and $Y_t = g(X_t)$, then Lsd will issue an error message providing information on where the error occurred.

The second difficulty is due to the possibility of using the wrong instance in an equation. Lsd offers simple and intuitive default behaviour ensuring that the correct computations are executed at the correct time. That is, modellers don't need to worry about the possible existence in their model of many elements with the same label. A simple notation will allow using the most obvious choice.

In both cases, modellers are anyway offered instruments to overrule the default behaviour in order to build any computational construction for their model.

It is important that, while writing an equation, the modeller remember that there is a reference to the actual Lsd Variable, which is computed with that equation. Note that if there are many Objects if the same type, they contain Variables with the same labels. The same code applies for each of these Variables, but Lsd manages to differentiated the behaviour (i.e. use different values) because each time the reference to the specific instance computed is modified.

To request the value of an element of the model the command is the following:

```
p->cal("Var", 0);
```

The meaning of the above command is rather simple. **p->** is the 'physical' address of the Object containing the Variable whose equation is computed. Therefore, the same symbol **p->** actually refers

to many different entities of the model, if the computed Variable is contained in Object replicated with many instances. **`Var`** is a Variable (or a Parameter, there is no difference in the notation) that can be contained in any place in the model. Lsd searches **`Var`** in the most obvious Objects: first in the same Object as the one containing the Variable computed; secondly in the descending Objects; eventually on ascending Objects. The search continues recursively (avoiding to return to the same Objects), so that the whole model may be explored if necessary. Most of times, the equations for Variables require only values in the same Objects or in ones in the immediate 'neighbourhood', and therefore the above command.

The last element, 0, is the lag. If 0, it means that the value of Var must be the one computed at the very same time step as the Variable under computation. Therefore, the equation for **`Var`** must be computed before the current equation. If, instead, the lag is 1, then the equation uses the previous value of Var. In this case, it is indifferent whether Var is computed before or after the Variable of the equation: Lsd will return anyway the previous period value. Of course the lag may be higher than 1.

A3 Local C++ variables in Lsd equations

The typical code for an equation consists in requesting the values of Variables and Parameters, then elaborate them and return the resulting value to the variable **`res`** (which will take care to deliver the value in the correct position). **`res`** is not a Lsd Variable, but it is a 'local' C++ variable that is created for each individual equation. By 'local' is meant that, for example, the value that **`res`** assumes during the equation for one Variable, does not affect its value when another equation is computed, not even when it is another Variable of the same type, using the same equation.

Like **`res`**, there are many other local variables that modellers can use. The most frequently used is a vector of real values, **`v[0], v[1], v[2]`** etc.. In fact, the commands to obtain values are relatively long, and it is not very readable to use them in logical or mathematical elaboration. Therefore, the best way is to compute the required values and assign them to local variables, and later these are used to write the elaboration. For example, it may be possible to write (it should be one single line)

```
res = p->cal("Price", 1)* p->cal("A",1) - p->cal("C",0) -
      p->cal("RIM", 0) - p->cal("RIN", 0)* p->cal("Inn",0);
```

but it is far clearer to read:

```
v[0] = p->cal("Price", 0);
v[1] = p->cal("A", 1);
v[2] = p->cal("RIM", 0);
v[3] = p->cal("RIN", 0);
v[4] = p->cal("Inn", 0);
v[5] = p->cal("C", 0);
res = (v[0]*v[1] - v[5] - v[2] - v[3]*v[4]);
```

Another special type of local variables is also available to modellers. It is 'pointers', i.e., variables to

which some Lsd functions can assign particular type of Objects. These variables are conventionally called **cur, cur1, cur2**, etc. As example of the use of these pointers, consider the case where each **Firm** where assigned a Parameter called **IdFirm**. Suppose also that in an equation you need to compute the value of, say **Q**, for one specific **Firm**, the one with **IdFirm** equal to 3. Then, your equation will read:

```
cur = p->search_var_cond("IdFirm", 3, 0);
v[0] = cur->cal("Q", 0);
```

Some of the Lsd functions do not provide values, but operate on the model in different ways, or return other elements of the model. In the following are listed all the Lsd functions, in alphabetical order. As a matter of simplicity, we will always refer to Lsd functions using the **p->** address. However, they can be applied using any Object of the model, stored in one of the local pointers to Object. In fact, as we will see, some of the functions provide the address of Objects, that can be used with Lsd functions. That is, if instead of **p->cal** the equation contains **cur->cal**, then the same function is applied, with the difference that instead of looking the value in **p->**, it searched in **cur->**, then its descendants, ancestors etc. Of course, **cur->** needs to be previously defined with a Lsd function.

A4 Lsd global C++ Variable: **p->** and **c->**

Any equation has available two references to the currently computed Variable. We have already seen **p->** (for *parent* Object of the computed Variable), referring to the Object which contains the Variable computed. The other, less frequently used, is the **c->**, for *caller*. This is the Object, which activated the computation for the Variable. For example, in the example seen above:

```
cur = p->search_var_cond("IdFirm", 3, 0);
v[0] = cur->cal("Q", 0);
```

When the equation for **Q** is computed in the Firm with IdFirm 3, the object here is referred to as **cur** will be available as **c->**, and, for example, it is possible to ask values to the caller Object. Modeller must consider that the **c->** Object may also be non-existent (assuming the conventional value NULL), when the equation is not computed because some other equation requested its value, but is the system that is proceeding to automatically update the Variable. **c->** is used only in very special occasion.

A5 List of Lsd functions

p->add_an_object("ObjLabel")

Creates a new instance of Object of type **ObjLabel** and attaches it as descendant of **p->**. The modeller needs to care that type of Objects **p->** can accept **ObjLabel** Objects as descendants. The newly created Object is placed at the end of the group of Object **ObjLabel**. The new Object is initialized according to the values stored in the Lsd model file, and, if the initialization entails many different instances of **ObjLabel**, the very first one is used. The function returns the address of the

newly created Object, to be used to overrule the default initialization.

`p->add_an_object("ObjLabel", ex)`

Creates a new instance of Object of type **`ObjLabel`** and attaches it as descendant of **`p->`**. The modeller needs to care that type of Objects **`p->`** can accept **`ObjLabel`** Objects as descendants. The newly created Object is placed at the end of the group of Object **`ObjLabel`**. The new Object is initialized according to the same values stored in the Object address **`ex`**. The function returns the address of the newly created Object, to be used to overrule the default initialization.

`p->cal("Lab", lag)`

Returns the value of the first instance of **`Lab`** found, computed with lag **`lag`**. Lab may be either a Variable or a Parameter, and in this latter case the lag is ignored. The search is made firstly in **`p->`**, then in its descendant, and then in the ancestor, recursively.

`cur->delete_obj()`

Remove the Object whose address `o->` from the model. Data tagged to be saved are stored for later analysis. Attempt to delete an object that is under use cause an error (for example:

`p->delete_obj()`

This formulation of the use of delete object is not correct since **`p->`** contains the very Variable to be computed, and therefore cannot be deleted. But other objects can be deleted in this way.

`p->draw_rnd("ObjLabel", "VarLabel", lag)`

This function searches for a group of Objects **`ObjLabel`**, then computes the values of **`VarLabel`** for each of them. It returns the address of one of the Objects of the group chosen randomly with probability linearly dependent on the value of **`VarLabel`**.

`p->draw_rnd("ObjLabel", "VarLabel", lag, total)`

This function works exactly like **`p->draw_rnd`**, but assumes that the sum of all the **`VarLabel`** in the group of Objects **`ObjLabel`** is equal to **`total`**. This assumption permits to speed up the computation.

`go_brother(cur)`

This function returns the Object subsequent to **`cur`**, which is a pointer to an Object, only if it is of the same type as **`cur`** (that is, they have the same label). Otherwise, it returns the conventional value **NULL**.

`p->increment("VarLabel", value)`

This function works only if **`VarLabel`** is contained in **`p->`**, otherwise produces an error. It adds to the current value of **`VarLabel`** the value of **`value`**. The function returns the new value after the increment.

p->lsdqsort("ObjLabel", "VarLabel", "direction")

Search a group of Objects labelled **ObjLabel** descending from **p->.** For each of them computes the value of **Var**, and sort the group of Object according to increasing (direction=UP) or decreasing (direction=DOWN) values.

p->overall_max("VarLabel", lag)

Searches for a group of Objects containing **VarLabel**, and returns the higher value of Var with lag **lag** from that group.

p->search("ObjLabel")

This function returns the first Object **ObjLabel** found as descendant from **p->**. The returned value can be assigned to a pointer (the equation provides several pointers for this purposes: cur, cur1, cur2 etc.), on which is later possible to apply the Lsd functions.

p->search_var_cond("VarLabel",value,lag)

Search all over the model, starting from **p->,** for an Object containing Variable **VarLabel** that, with lag **lag**, assumed the value of **value**. The function returns the address of the object, which can be assigned to a pointer for later applying a Lsd function.

p->stat("VarLabel", vector)

Computes statistics over the values of Variables **VarLabel** contained in Objects that are descending from **p->.** The statistics are placed in the vector **vector** with the following values:

vector [0] = number of elements

vector [1] = average

vector [2] = variance

vector [3] = maximum

vector [4] = minimum

p->sum("VarLabel")

Using the same search procedure as for **p->cal** to locate **VarLabel**, this function uses the whole groups of Objects containing **VarLabel** and returns the sum their values.

p->write("VarLabel", newvalue, time)

This function works only if **VarLabel** is a Variable contained in **p->,** otherwise an error is issued. The value in **VarLabel** is replaced by **newvalue**, and appears as computed at time **time**. This function is used to replace the value of Parameters, or to initialise newly created Objects.