IIASA

International Institute for Applied Systems Analysis • A-2361 Laxenburg • Austria
Tel: +43 2236 807 • Fax: +43 2236 71313 • E-mail: info@iiasa.ac.at • Web: www.iiasa.ac.at

INTERIM REPORT IR-97-020/May

Laboratory for Simulation Development User Manual

Marco Valente (mv@business.auc.dk)

Approved by

Giovanni Dosi (dosi@iiasa.ac.at)
Leader, TED Project

Contents

Chapter 1 - Introduction to Lsd 1
1.1 Lsd Models 2
1.1.1 Structure 2
1.1.2 Equations 3
1.1.3 Initial Data 3
1.2 Lsd Models' Language 3
1.2.1 Objects' Content 3
1.2.2 Models' Structure 5
Chapter 2 - Lsd 0.23 Tutorial 6
2.1 Installation 6
2.1.1 Unix 6
2.1.2 Windows 7
2.2 Compiling the Nelson and Winter Example Model 7
2.2.1 Unix 8
2.2.2 Windows 10
2.3 Running the First Simulation 11
2.4 Browsing through the Model 15
2.5 Run Time Settings 16
2.5.1 Variables' Run Time Settings 16
2.5.2 Simulation Manager 17
2.6 Initial Data Settings 18
2.6.1 Setting Objects' Number 19
2.6.2 Setting Internal Object Initial Data 21
2.7 Code for Identification of Multiple Objects 23
2.8 The Equations of a Model 24
2.9 Simultaneous Equations 28

2.10 Adding a Parameter to a Model 29
2.11 Adding New Types of Object 32
2.12 Debugger 33
2.12.1 Debugger Window 34
2.12.2 Conditional Breaks 36
2.12.3 Scheduling Debugger 38
Chapter 3 - Nelson and Winter Model 41
3.1 Model Contents and Initial Values 41
3.2 Equations 42
Chapter 4 - Dosi-Kaniovski-Winter Model 46
4.1 Model Contents and Initial Values 46
4.2 Equations 50
Chapter 5 - Paillard Model 56
5.1 Model Contents and Initial Values 56
5.2 Equations 60

Index of Figures

Lsd Object 4
Figure 1 - Empty Lsd 10
Figure 2 - Setting Files’ Name 12
Figure 3 - Structure of The Model 13
Figure 4 - Simulation Summary Window 13
Figure 5 - Browser 15
Figure 6 - Modifying Variables’ Properties 17
Figure 7 - Simulation Manager 17
Figure 8 - Object Editor 19
Figure 9 - Multiple Branching Models 20
Figure 10 - Data Editor 22
Figure 11 - Setting All Data 22
Figure 12 - Data Editor with Two-level Code 24
Figure 13 - Adding a Variable 31
Figure 14 - Adding Descendants 32
Figure 15 - Lsd Debugger 34
Figure 16 - Setting a Variable for the Debugger 36
Figure 17 - Conditional Breaks 37
Figure 18 - Conditional Break 37
Figure 19 - Stack Level 39

Preface

This new research project at IIASA is concerned with modeling technological and organisational change; the broader economic developments that are associated with technological change, both as cause and effect; the processes by which economic agents -- first of all, business firms -- acquire and develop the capabilities to generate, imitate and adopt technological and organisational innovations; and the aggregate dynamics -- at the levels of single industries and whole economies -- engendered by the interactions among agents which are heterogeneous in their innovative abilities, behavioural rules and expectations. The central purpose is to develop stronger theory and better modeling techniques. However, the basic philosophy is that such theoretical and modeling work is most fruitful when attention is paid to the known empirical details of the phenomena the work aims to address: therefore, a considerable effort is put into a better understanding of the `stylized facts' concerning corporate organisation routines and strategy; industrial evolution and the `demography' of firms; patterns of macroeconomic growth and trade.

From a modeling perspective, over the last decade considerable progress has been made on various techniques of dynamic modeling. Some of this work has employed ordinary differential and difference equations, and some of it stochastic equations. A number of efforts have taken advantage of the growing power of simulation techniques. Others have employed more traditional mathematics. As a result of this theoretical work, the toolkit for modeling technological and economic dynamics is significantly richer than it was a decade ago.

During the same period, there have been major advances in the empirical understanding. There are now many more detailed technological histories available. Much more is known about the similarities and differencers of technical advance in different fields and industries and there is some understanding of the key variables that lie behind those differences. A number of studies have provided rich information about how industry structure co-evolves with technology. In addition to empirical work at the technology or sector level, the last decade has also seen a great deal of empirical research on productivity growth and measured technical advance at the level of whole economies. A considerable body of empirical research now exists on the facts that seem associated with different rates of productivity growth across the range of nations, with the dynamics of convergence and divergence in the levels and rates of growth of income in different countries, with the diverse national institutional arrangements in which technological change is embedded.

As a result of this recent empirical work, the questions that successful theory and useful modeling techniques ought to address now are much more clearly defined. The theoretical work described above often has been undertaken in appreciation of certain stylized facts that needed to be explained. The list of these `facts' is indeed very long, ranging from the microeconomic evidence concerning for example dynamic increasing returns in learning activities or the persistence of particular sets of problem-solving routines within business firms; the industry-level evidence on entry, exit and size-distributions -- approximately log-normal; all the way to the evidence regarding the time-series properties of major economic aggregates. However, the connection between the theoretical work and the empirical phenomena has so far not been very close. The philosophy of this project is that the chances of developing powerful new theory and useful new analytical techniques can be greatly enhanced by performing the work in an environment where scholars who understand the empirical phenomena provide questions and challenges for the theorists and their work.

In particular, the project is meant to pursue an `evolutionary' interpretation of technological and economic dynamics modeling, first, the processes by which individual agents and organisations learn, search, adapt; second, the economic analogues of `natural selection' by which interactive environments -- often markets -- winnow out a population whose members have different attributes and behavioural traits; and, third, the collective emergence of statistical patterns, regularities and higher-level structures as the aggregate outcomes of the two former processes.

Together with a group of researchers located permanently at IIASA, the project coordinates multiple research efforts undertaken in several institutions around the world, organises workshops and provides a venue of scientific discussion among scholars working on evolutionary modeling, computer simulation and non-linear dynamical systems. The research will focus upon the following three major areas:

1. Learning Processes and Organisational Competence.

2. Technological and Industrial Dynamics

3. Innovation, Competition and Macrodynamics

Abstract

Lsd is a computer package aiming at facilitating the use of simulation models in Economics. This document is meant to explain the concept of *simulation models* used in Lsd, describe how Lsd manage models, and give instructions for its use.

The document is structured in five chapters. The first introduces Lsd and the concept of simulation model. The second chapter is a tutorial for the use of the package. It describes Lsd interfaces for running simulations of existing models and produce modified versions of them. The last three chapters describe three models implemented in Lsd. They are meant to present the functions available in Lsd to write complex models.

Readers are not requested to possess a deep knowledge of computer programming, but are supposed to be concerned, if not actually involved, with the issues of using computers for simulation modelling.

Acknowledgments

The author wishes to thank IIASA and particularly the TED project for the support received during his stay in Laxenburg. In particular, the discussions with and comments by Giovanni Dosi and Francesca Chiaromonte were crucial for the success of the work presented here.

About the Authors

Marco Valente has been working in IIASA from 1995 to 1997 to develop a software to facilitate simulations for evolutionary models. Since March 1997 he is working in the University of Aalborg on a project based on the study of the co-evolution between demand and supply.

Laboratory for Simulation Development User Manual

Marco Valente

Chapter 1 - Introduction to Lsd

Lsd is meant to facilitate the use of simulation models, both for modellers and for non experienced computer users running computer simulations. To modellers, it allows to concentrate exclusively on the theoretical contents of the model, providing a library of ready-to-use functions dealing with the technical details of a simulation model. To final users, it provides graphical interfaces allowing an easy exploration of the model and the setting of the parameters for a simulation run.

Lsd is composed by two parts: a model manager and a model interpreter. The model manager allows to load and modify a model while the model interpreter runs the actual simulation. Both the model manager and interpreter are model independent: they adapt a set of standard graphical interfaces and a computational engine to any model. After a model is loaded, Lsd behaves as a stand alone program, specifically written for the simulation of that model.

The Lsd model language frees modellers by any technical detail necessary to run the simulation. The approach used was inspired by the Object Oriented programming paradigm. This approach, which is widely used in computer science, allows programmers to use a sort of top-down strategy when designing and implementing software. The programmers define a number of abstract entities without any specification of the actual computational contents. A gradual specification of the behaviour of such entities adapts the abstract definitions to more and more specific instances. As an example of the Object Oriented approach, consider a program to manage geometrical figures. The first step is the definition of an object as a “Figure” whose only properties are, for example, Perimeter and Area. This definition is useless, since it does not have any computational content, but it opens the way to define derived objects, e.g., a “Square”, which contains an actual definition for the computation of Area and Perimeter. Another derived object could be a “Circle” and so on. Other objects can make use of objects “Figure” without knowing in advance which specific instance of “Figure” is used. An Object Oriented language is particularly suited to write programs involving frequent revisions and expansions: adding new objects derived from “Figure” will expand the previous code without altering the behaviour of the other parts of the program.

In Lsd, programmers-modellers are supposed to specify Objects to define the entities of their models. Lsd provides an abstract definition of Object containing all the machinery to run the implementation of the Object itself. Hence, programmers-modellers can concentrate entirely on the “creative” part of building a simulation model by defining Objects derived from the abstract one. To derive a new instance, the modeller needs just to give a name to the Object and to its elements (that is, variables or parameters). The abstract definition of Lsd Objects provides also connections which can be used to link Objects to each other, so to create the structure of a model.

The computational content of a model (i.e. the equations) is expressed as a list of functions, one for each variable label; hence, modifying an equation does not involve to work inside the core of the program. Moreover, Lsd provides functions which facilitate the writing of equations, so to allow modellers to express computations as if they were using an equation editor. For example, Lsd offers solutions to retrieve variables by using only their labels (no vectors are used), and deals with the scheduling of computations by using a simple lag notation. The simplicity of use reduces the possibility of errors.

A model run with Lsd ensures high efficiency of computation, hence allowing to express heavy computing models, because the model interpreter has very few computational overheads, producing the equivalent of the output from C++ code.

The models written with Lsd can be decomposed in their fundamental components, which can then be re-used in other models. The components are instances of Objects, possibly entire “branches” of connected Objects, and their equations. There is no problem to single out components from a model and to plug them in other models, since there is no other dependency between Objects than the logical consistency of the equations for their variables.

1.1 Lsd Models

Lsd distinguishes three types of components of a model: structure, equations and initial data. It is possible to define and/or modify at different times the components, so to test parts of the model or run reduced forms.

1.1.1 Structure

The **structure** of a model in Lsd is defined in terms of its Objects and the relations among them. The model is defined in order to be highly modular; that is, to be easily modified and expanded. For example, having a model with a number of Firms operating in a Market, a modeller can be interested in expanding this model adding the possibility for Firm to use differentiated types of Capital. He will have to define new Objects (and possibly add equations for the variables in the new Objects) but will be able to re-use the unaltered parts of the structure and the existing equations.

The Lsd models are shaped in a tree-like hierarchical structure of connected Objects. This provides a default search pattern for the exchange of information among Objects. For example, to define Firm as descending from Market means that any request of a variable in Market by an equation in Firm will provide, by default, one specific instance of the variable, even though the models contains many Objects Market. This default system allows to write the equations for models with an extremely simple syntax. In any case, the default system can be overruled with several Lsd functions allowing modellers to express any possible equation.

1.1.2 Equations

The **equations** of a model must be written as C++ code, extended with the library of Lsd functions. Any information related to the model is treated by the Lsd functions, allowing modellers to use a syntax similar to writing difference equations on paper. The elaboration of this information at run time enjoys the reliability and the speed of C++ code. The final result is an easy to use “equation language” producing very fast code, and hence suited for heavy computing models.

1.1.3 Initial Data

The **initial data** are used to set up a simulation model before running an actual simulation. They are stored in a text file, together with other technical information (number of steps, variables to save, etc.). Users of Lsd models can use a simple and effective graphical interface to modify the initialisation of a model. Every initial value can be modified: the number of entities in the model (e.g. one Market and eight Firms), the parameters values (e.g. coefficients of a demand curve), and lagged variables values used by the equations during the first time step (e.g. past values of demand for some adaptive expectations rule).

1.2 Lsd Models’ Language

1.2.1 Objects’ Content

The basic elements of a Lsd model are Objects. An Object contains variables, defining its state, and equations, which update variables values at each time step. Typically, an equation is made of the logical-mathematical elaboration of the values of other variables of the model. The elaboration can be any legal C++ expression, while the values of the variables are expressed in the equation by their names. The Lsd model language provides modellers with a set of functions (implemented as methods of the class Object) to retrieve values of other variables or some standard elaboration on them.

A simulation run consist in a sequence of time steps, during which every variable in every Object of the model is computed using its equation. The computations for the variables takes place virtually in parallel, unless the modeller decides a precise scheduling for some of them. The scheduling definition is done by using the same notation used to write difference equations:

$$X_t=f(Y_t)$$

implies that the most recent value of Y must be computed before the computation of X, since its new value is used to compute the new value of X.

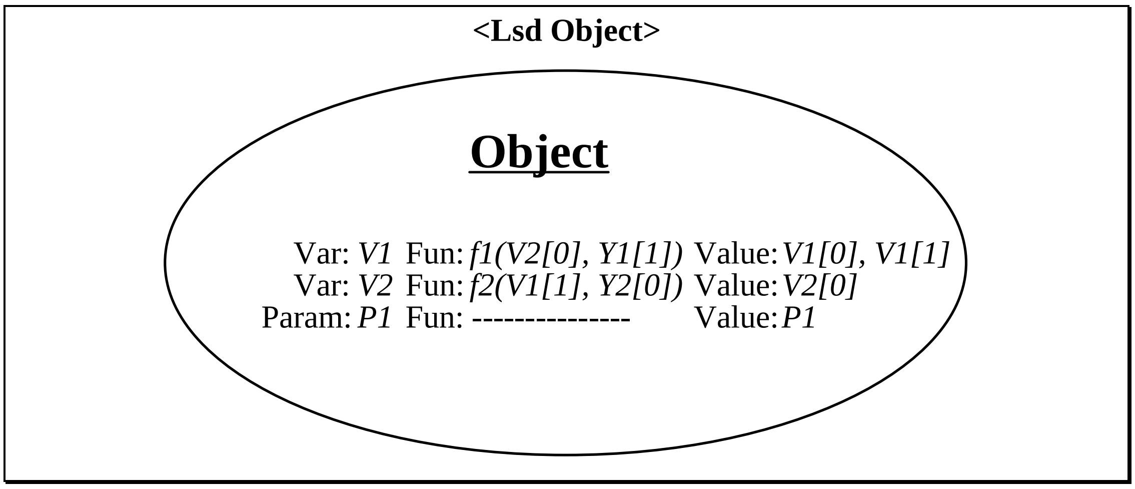
On the other hand,

$X_t=f(Y_{t-1})$
implies that the equation for Y could also be computed after the computation for X, since the latter does not need the most recent value for Y. Lsd allows to express the scheduling of the computations for a simulation using the lag notation. Hence, the

modeller is not required to prepare the complete set of steps for the simulation, but only to set the relevant precedences for the model. A set of tools allows to modify and debug the scheduling even for complex situations.

From the basic definition of an Object one can derive definitions of markets, firms, departments, universities etc. The same Object can be moved to interact with different types of Objects in different models, provided a logical consistency between the two models exists. In fact, the only interaction of an Object with the rest of a model are requests for the value of certain variables. The nature of the Object containing a requested variable does not affect the requesting Object; hence, it can be freely modified without effects on the functioning of the latter. For example, consider an Object Market designed to work in a model where some Object Firms have been defined. If the Object Market uses only the variable “Production” within each Object Firm, it can work with any other Objects having a variable with the same name. This opens the way to define libraries containing a set of Object Markets, a set of Object Firms etc., which would ultimately allow modellers to construct models without even defining new type of Objects.

An example of an Object is reported in the figure below:



<Lsd Object>
Object
Var: V1 Fun: f1(V2[0], Y1[1]) Value: V1[0], V1[1]
Var: V2 Fun: f2(V1[1], Y2[0]) Value: V2[0]
Param: P1 Fun: --------------- Value: P1

This Object contains two variables and one parameter. Corresponding to each variable there are: a label (V1 and V2 in the example); an equation, computing the value at each time step in the simulation (f1(...) and f2(...)); a vector of values (named after the variable, plus an index between square brackets).

Each equation uses, in general, the values of other variables or parameters, contained either in the same Object, or elsewhere in the model. In the example, the equation for V1 uses the value of the variables V2 and Y1, and the equation for V2 uses V1 and Y2. There is no difference in using variables external (Y1 and Y2) or internal (V1 and V2) to the Object: the Lsd interpreter uses the variable label to trace them in the model. The scheduling for the computations at each time step is determined by the use of the index attached to the variables’ labels in the equations. In the example above, the equation for V1 uses the value of V2 with the index “0” and the value of Y1 with the index “1”; the index is interpreted as a time lag in a normal difference equation. In other words, the equation for V1 could be re-written as:

$V1_t=f_1\{V2_t, Y1_{t-1}\}$

This implies that the value for V2 must be computed before the system can compute the value for V1, at each time step. The computation for V2, instead, does not require the most updated value for V1 (its equation uses V1[1]: the value computed at the previous step). The modeller does not need to explicitly specifying the scheduling of the computations; he just needs to correctly set the lags for every variable used in the equations. In the example, a deadlock error would occur, in case both variables needed the other variable most recent value, that is, the equations were:

V1: f_1\{V2[0], Y1[1]\}

V2: f_2\{V1[0], Y2[0]\}

since both V1 and V2 request the most recent value for the other.

The parameters in an Object are "variables with a constant value". As such, they don't have an equation, nor past values; they simply keep their value for the entire run of the simulation. Note that, in order to run a simulation, the user needs not only to specify the initial and constant value of the parameters, but also to provide the values for the V1 and Y1. In fact, during the first time step, these variables are requested to provide the previous value, and hence, they are part of the initialisation of the simulation.

1.2.2 Models' Structure

In a model, the Objects are connected to each other to form a hierarchical tree. The hierarchical relation can be considered as a relation of "part of": any Object is composed by the set of Objects descending from it. For example, an Object Market can be considered as composed by a set of Objects Firm. Hence, Market will be represented as hierarchically superior to Firm. The Object Firm, in turn, can be considered as composed by a set of Objects Capital, and so on. The hierarchical relation is used to trace which Objects are explored when other Objects request the value of a variable. For example, if a model has two Objects Market, then a Firm using a variable contained in Market needs to discriminate which Market it must refer to. If the modeller placed the Objects Firm as descendants of Objects Market, any Firm will use by default the Market immediately above it. The modeller, in this case, does not need to use any index or other means to discriminate between the two Markets. Of course, if the modeller wants Firms to decide in which Market to participate, he must explicitly determine how to choose among them, and will build a different structure (e.g. Firms descending from an Object Market_Exploration containing the equation to choose among the Markets).

The hierarchical structure does not limit in any way the type of models representable with Lsd. It is used internally by the system as a default search path for the values needed by the equations for the variables. The modeller can overrule the default system in many ways. The philosophy used in Lsd is to allow modellers to express simple situations in a simple way, but it is always possible to apply complex functions to represent complex situations.

Chapter 2 - Lsd 0.23 Tutorial

This tutorial introduces the Lsd system, and explains its basic functions. It explains how to compile and run a simulation, how to observe and set the initial data, and how to modify a model. It uses as example the model presented in Chapter 12 of the book "An Evolutionary Theory of Economic Change", 1982, by Nelson and Winter. The reader is invited to install the system and execute the commands while reading. We don't describe the example model in detail; it would be useful to have the Nelson and Winter book at hand in order to be able to check the verbal description of the model, its equations and its initial values, while going through the Lsd implementation.

Other two example models are provided with the present version of Lsd. Even though they refer to still unpublished models, they are used to introduce some properties offered by Lsd for complex models. For example, they modify endogenously their structure creating, while the simulation is running, new instances of Objects. After the reader has familiarised with the Lsd interfaces, he can use these other models to explore the full potentiality of the system.

We tried to keep the graphical interfaces as simple as possible, avoiding the use of many buttons, labels etc. Many functions to modify elements of the model, or their names, are activated by clicking on the labels showing the current names, so to avoid to have too many buttons in the windows.

The use of the button **Ok** in the windows is generally equivalent to the use of the return key on the keyboard, except when they are explicitly described as performing different actions.

2.1 Installation

Lsd has been developed under Linux using the gnu C++ compiler, gcc. It makes use of the Tcl/Tk language (respectively, versions 7.6 and 4.2) to provide graphical facilities. The same code has been compiled and run on a Sun workstation using OpenWindows and has been ported under Windows 95.

2.1.1 Unix

We are assuming that Lsd 0.23 has been successfully unpacked and that the operative system has already installed Tcl 7.6 and Tk 4.2 (available in any Sunsite). Other requirements are the libraries X11, required by Tk, and the standard libraries for the gcc compiler. Regarding this libraries, there should be not compatibility problems for any decently recent versions.

The Lsd package is composed by a directory, called "lsd0.23", and by 5 descending directories:

- lsd0.23/src : source files for Lsd

- lsd0.23/nw82 : example model by Nelson and Winter

- lsd0.23/dkw : example model by Dosi-Kaniovski-Winter

- lsd0.23/paillard : example model by S.Paillard
- lsd0.23/a_model: a “framework” model, to begin the construction of a new model
- lsd0.23/doc : documentation

Any example model directory contains a makefile, set according to the default system structure on a Linux system. In the lsd0.23 directory a README file contains the detailed description of the system requirements, and the instructions to customise the compilation of Lsd to your system environment.

2.1.2 Windows

To install the Tcl/Tk graphical language, you need to run its installation program (win76.exe) available in any Sunsite. You can choose the default installation options. Just note the directory where Tcl is installed (by default it should be c:\tcl). Place the lsd023.zip file in the directory from which you want to create the lsd directory structure.

Unpacking the lsd023.zip file using the command:

pkunzip -d lsd023.zip

from the dos window or allowing for the directory creation using the winzip program, will create a directory lsd0.23 containing the following subdirectory:

- lsd0.23/src : source files for Lsd
- lsd0.23/nw82 : example model by Nelson and Winter
- lsd0.23/dkw : example model by Dosi-Kaniovski-Winter
- lsd0.23/paillard : example model by S.Paillard
- lsd0.23/a_model: a “framework” model, to begin the construction of a new model
- lsd0.23/doc : documentation

Each directory contains the executable for the model indicated by the name of the directory. We use a name to refer to any example: “nw82” for the Nelson and Winter model, “paillard” for the Paillard model, “dkw” for the Dosi-Kaniovski-Winter model. We will refer to the Nelson and Winter model for the tutorial. For the other examples one needs to use their names instead of “nw82”.

2.2 Compiling the Nelson and Winter Example Model

As we will see shortly, a model is composed by structure, initial data and equations. The equations are expressed by C++ code (extended to include a set of functions specific of Lsd) contained in a normal C++ source file, and therefore there is one equation file for each model. This file needs to be compiled and linked to the rest of the code in order to run a simulation. Hence, a model can be run only if the Lsd program has been compiled together with its equations file. Other functions of Lsd, e.g. the ones reading and modifying structure and initial data for a model, do not involve the equations for the model and can therefore be used independently. In other words, one can think of Lsd as composed of two subsystems: the first regards the model definition and the second the model simulation. The model definition is dynamically editable by users, that is, it is possible to load a model, modify it, save it, discard it and re-load another model using the same instance of Lsd. The model simulation, instead, needs the code for the equations to be statically linked to the system: only one model can be run with each instance of Lsd because each instance can contain one single set of equations[1]. In the following two paragraph are exposed the instructions to compile a model. Given the wide variety of dialects, the unix version does not contain the binary files, and therefore the user needs to compile the examples. The Windows version instead includes instead also the executables and the user needs only to customise the batch file.

Both versions of Lsd, accept as parameter a string like beginning with -f, for example:

```
lsd -fMY_MODEL
```

where MY_MODEL is the name of the model files to load by default. Without such parameter, the system will start with the default model name SIM1. In any case, as you will see, it is possible to change the name of the model files after the system is run.

2.2.1 Unix

If you are in the directory lsd0.23, you need to go in the directory “nw82”, that is, typing:

```
> cd nw82
```

Here, you need to compile the system and the equations of the model, which must be linked to build the executable example. These operations are defined in a makefile, that is a text file used by the “make” program. You should just need to type the command:

```
> make
```

The compiler will begin to compile the system files placed in the directory lsd0.23/src. As last step, it will compile the function file fun_nw.cpp, and will link all the objects files producing the program called “nw82”. If an error message occurs, you need to edit the makefile in the current directory. The most likely errors are provoked by missing libraries (typically the tcl, tk and X11 libraries). The makefile contains environment variables for the libraries’ paths: you need to check if the paths are correct and, if they are not, modify the content of these variables.

A library is a file used by compilers to include a set of functions, already compiled, in the code of a program. Normally, when the compiler is requested to link the library XXX, it searches for the file “libXXX.a” (or “libXXX.so”, the extensions “.a” or “.so” are equivalent) in some standard directories. If a library is not found by the compiler, it can be due to two reasons: the name of the library is misspelled or the path where the library is installed is not listed in the standard directories. The makefile allows to set variables in order to add particular paths for the libraries and to express names of the libraries different from the standard ones. The user, in case of compilation errors for missing libraries, needs to find where the libraries are, and edit the variables in the makefile to have them correctly linked to the program.

[1] The reason for keeping the equations hard-coded in C++ is twofold: it allows the maximum freedom when deciding the equations of a model and it provides very fast code. The future evolution of Lsd will contain an equation editor which will allow modellers to modify dynamically also the equations.

The libraries for tcl and tk are normally installed with the version numbers attached; that is, their names will be “libtcl7.6.a” and “libtk4.2.a”. If this is not the case, you need to set in the makefile “TCL_VERSION=“ instead of “TCL_VERSION=7.6” and “TK_VERSION=“ instead “TK_VERSION=4.2”.

The path for these libraries is usually “/usr/local/lib”. If this is not the case, you need to find where they have been stored on your system and change the variables TCL_LIB_PATH and TK_LIB_PATH replacing the default path with the one on your system.

If the X11 library cannot be found by the compiler, you need to find the file “libX11.so” on your system and replace the default path in “X11_LIB_PATH=/usr/X11R6/lib”.

The last customisation you can need to make regards the header files for tcl and tk. Many files containing parts of Lsd code need to use the headers “tcl.h” and “tk.h”. The variables PATH_TCL_ HEADER and PATH_TK_ HEADER must be set to the path containing respectively the “tcl.h” and the “tk.h” header files.

The Lsd system needs to use the Tk library, which is built on the Tcl and the X11 libraries. On some systems (not on Linux, but on the Sun, for example) the X11 library needs also other libraries. If you cannot compile, the easiest trick is to check which libraries are needed is to run “ldd” on the X11 library. This program produces a list of the shared libraries necessary for the library specified in the argument. For example, typing:

```
> ldd /usr/local/openwin/lib/libX11.so
```

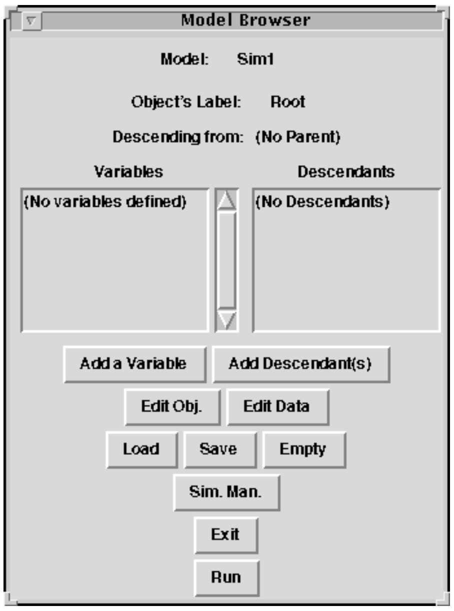
produces a list the libraries used by X11 (placed in the directory /usr/local/openwin/lib). Hence, you must link the libraries in the list in your program.

If there are no error messages during the compilation, you will find a new file, called “lsd” in the lsd0.23/src directory. This is an executable file containing the Lsd system linked with the equations for the Nelson and Winter model. Just type:

```
> lsd
```

and you will have the Lsd system running.

<Figure 1 - Empty Lsd>



2.2.2 Windows

If the Tcl/Tk package has been installed in the directory:

c:\tcl

it is sufficient to click on Lsd_nw.bat to run the model. This file starts the system passing the directory c:\tcl as the address for the Tcl/Tk language. If you chose a different directory an error messages, signalling that the Tcl/Tk directory was not found will appear. In this case you need to edit the file Lsd_nw.bat, whose content is the following line:

lsd_nw.exe -ic:/tcl -fnw82

replacing the parameter -ic:/tcl with -iTCL_DIRECTORY, where TCL_DIRECTORY is the one where Tcl/Tk was installed. Note that you must use the unix-style forward slash / instead of the normal dos slash \ (the Tcl/Tk, being originally developed under unix, has some problems in dealing with the \ character).

Each model directory contains also an IDE file, containing the settings for its compilation using the Borland C++ compiler, ver. 5.0. It is a project indicating to compile every source file in the \src directory plus one source file for the equation of a model (normally called fun_XXX.cpp). As explained above, the compilation is necessary to modify the equations in a model.

2.3 Running the First Simulation

By default, the system starts with no model loaded[2]. You can see three windows: a large graphical window (currently empty) will show the graphical representation of the model, once you will load it. A smaller window (Model Browser, represented in the Figure 1) will show the contents of the Objects, and is also used as control centre to communicate with the system. The third window, titled Log, will contain the messages from the system. Since there is no model loaded, the graphical window is empty and the browser shows no Variables or Descendants in the respective lists.

Besides the equation file, the Nelson and Winter model is defined in two separate files: “nw82.str” and “nw82.par”. The first contains the structure of the model while the second contains the numerical values for the initialisation (i.e. the initial data). You can observe these files with an editor[3]: the structure file contains the “declaration” of the Objects, their relative position in the model and their content in terms of variables and parameters; the initial data file contains the numerical setting for a simulation: how many instances for each type of Object, the parameters values, variables lagged values and other technical information. Lsd uses the extension of the files to discriminate their content. You can use the name of the files to distinguish different models or different initialisation settings of the same model. Hence, in order to have Lsd load a model, you need to name for it. This is done by clicking as indicated in Figure 2. Type in the resulting window the new files’ name “nw82”[4].

[2] This instance of Lsd can run only the Nelson and Winter model because its equations are linked in the binary file. But it can be used to define and edit the structure and the initial data for any model. This is why you have to explicitly load the model. Windows users do not need to change the name, since the system was run using the -f option.

[3] Do not modify these files using the editor. They are automatically generated by Lsd and can be modified only respecting the Lsd format for these files.

[4] When you ran Lsd by typing “lsd” you could have given a parameter containing the default name for the files instead of “Sim1”. For example, you could have typed:

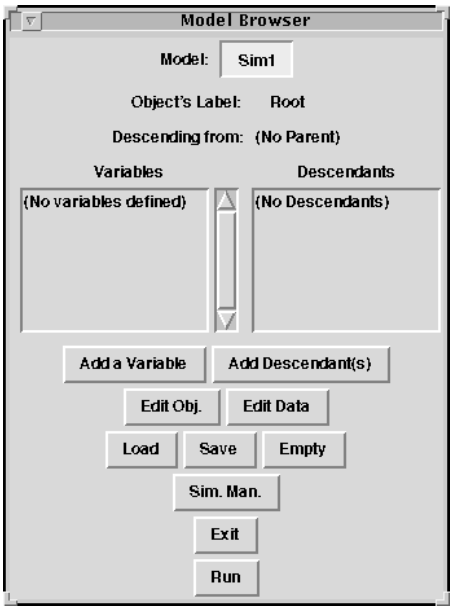
```
> lsd My_Model
```

replacing “Sim1” with “My_Model”. Next time you run the Nelson and Winter model, you can type

```
> lsd nw82
```

so to skip this step.

<Figure 2 - Setting Files' Name>

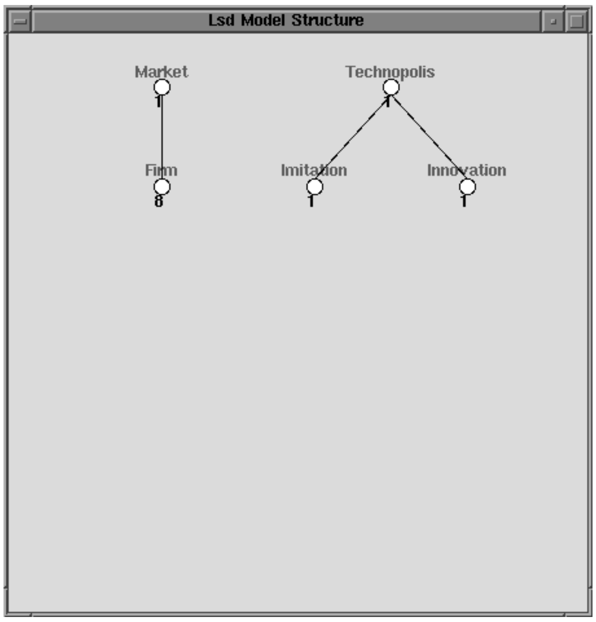


At this point, clicking on the **Load** button the system will load the model structure and the initial data contained in the specified files. If an error occurs, it is likely that you misspelled the files' name. Remember that Unix is case-sensitive regarding the files' name. If you want to load the model files from a different directory in respect of the current one, you must use the unix-like slash / also if you are running the windows version.

Now the graphical window (Lsd Model Structure) shows the structure of the model. It shows the type of the Objects descending from the root drawing an circle for each type Object of object. The lines connect the parent Object (upper circle) to the descendants (lower circle). The labels indicate the names of the Objects. The numbers below the circles indicate the how many Objects of that type are used in the model. The Model Structure windows provides several shortcuts to browse through the model and perform the most frequent operations. Namely, it shows the content of the Objects (when the pointer passes over one Object); positions the Browser on one Object (by clicking on the left button); shows the initial data window for the Object (clicking on the right button). Such shortcuts will be mentioned later when these operations will be explained.

The model represented in Figure 3 shows two Objects descending from the Root (which is not depicted): an Object Market and an Object Technopolis. From the Object Market descend the Objects Firm. As indicated by the number below the symbol for the Firm, there are 8 instances of this Object. From the Object Technopolis descend two Objects (Imitation and Innovation) both represented with one instance each.

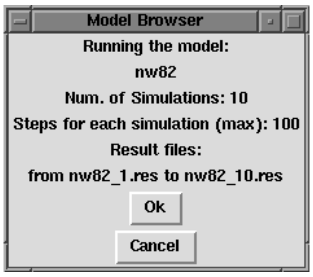
<Figure 3 - Structure of The Model>



Once the loading is completed, you can run the simulation by clicking on the **Run** button. Lsd allows to run sets of simulations with the same model and the same initial data. For each simulation run in the set, a different seed for the pseudo-random number generator is set by the Lsd interpreter, and the results of each run are saved in a file, named after the model's name and the seed used. Moreover, for each simulation set, the system creates a result file, storing, in each row, the final values of the variables for each simulation run.

Before the simulation is run, the system shows a summary window declaring what it is going to do and the names of the result files it is going to create (and overwrite, if they already exist). The window, shown in Figure 4, reports the name of the model; the number of simulation runs in the set, with the maximum number of steps for each run; the name of the result files which are going to be created to store the results for each run. The system creates as many result files as many simulation runs have been indicated. There names are obtained by the model name, an underscore and an integer number, while the extension is always ".res". This number refers to the seed for the random number generator used for the simulation run. Since the default settings ask for a 10 simulation runs, the result files produced goes from nw82_1.res to nw82_10.res. See the Run Time Settings section for instruction about these options.

<Figure 4 - Simulation Summary Window>


Model Browser
Running the model:
nw82
Num. of Simulations: 10
Steps for each simulation (max): 100
Result files:
from nw82_1.res to nw82_10.res

It is important to check the information reported by this window because the system is going to overwrite, without further warning, every result and model file having the same name. In particular, the model files (here “nw82.par” and “nw82.str”) will be overwritten with the current content of the system. Also any result file, possibly obtained by previous experiments, will be overwritten, if they have the same name of the ones which the system is going to use, that is, if the same model is run using the same random number generators. Clicking on **Cancel**, the user can come back to the Browser window to modify, e.g., the model’s name, to avoid unwanted overwriting of existing files. If you want to keep two different settings for the same model, you can load one model, make the changes in the settings, change the model name and save it. Thus, there will be two couples of “.par” and “.str” files and the result files will have different names.

In order to start the simulation, click on **Ok**. During the simulation the Log window contains the messages of the simulation steps terminated. In case of multiple simulations, another message signals the end of each simulation run. The windows are refreshed at the end of each simulation step. To further increase the simulation, it is possible to click on the **Fast** button, which will cause the step messages to be suppressed and the refreshing of the window only at the end of each simulation run. This can be used for simple models where the time of writing a line is relatively long in respect of the whole simulation and you are running many simulation runs. For example, if you try to click on **Fast** with the Nelson and Winter model, the global time for the simulation will be sensibly reduced. Clicking on **Observe** the step messages are restored. Clicking on **Stop**, the simulation will abort.

The initial data for the Nelson and Winter model correspond to the model with 8 firms and parameter BANK equals 1, using the values reported at page 302 of the book. Each simulation run is set to stop after 100 steps and are set by default 10 simulation runs.

You can observe the results produced by the simulation opening the file “nw82_1.res” which has been created during the simulation. It is a tab-delimited text file containing a column for each variable saved. Results can be plotted, for example, by loading the file with Excel in windows or with gnuplot, in unix. A summary result files (named after the model with the “.tot” extension) contains the same variable as the other result files, but contains only one line for each simulation. This file allows to compare the final values for the different simulation runs.

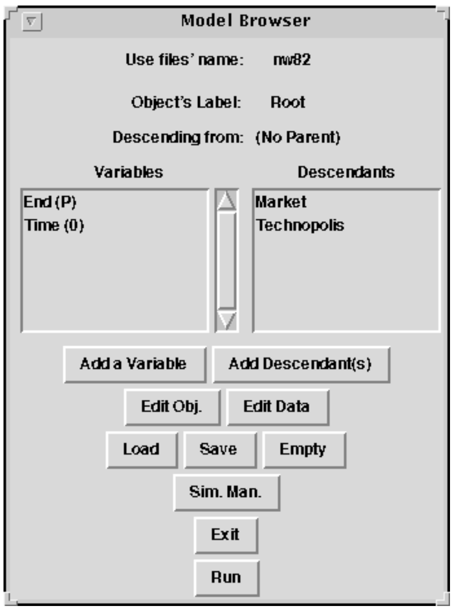
Note that the columns’ titles replicate the names of the variables. The variables for Firms have attached a number referring to their Firm instance: A_1 is the productivity for the first Firm, A_2 for the second etc. This is a code used to differentiate the same variables contained in different instances of the same types of Objects. The code is automatically generated by the system any time there is the possibility of confusing different instances of variables with the same names (that is, the same variables in different instances of the same type of Object). The general definition for this code will be described in detail below.

The variables saved in the result file are determined by the user. You just accepted the choice defined in the data file “nw82.par”, but you could have modified this choice. In the section “Run Time Settings” we will describe how to set the variables to be saved.

2.4 Browsing through the Model

After a set of simulation runs (we just ran a one element set, that is one single run), the system discharges automatically the model and re-loads it from the model files; Lsd contains now the model as it was before running the simulation. Let's see what the Browser window shows and allows to do.

<Figure 5 - Browser>



Model Browser

Use files' name: nw82

Object's Label: Root

Descending from: (No Parent)

Variables: End (P), Time (0)

Descendants: Market, Technopolis

The Browser window shows the content of an Object in two lists, one for the variables (showing both variables and parameters) and one for the descending Objects[5]. In the beginning, the browser points to the root of the model. It indicates the name of the current Object (Root), on the right of **Object's Label:,** and the name of the Object it descends from (none for the Root), on the right of **Descending From:**. The variables and parameters contained in the Root are used for controlling technical aspects of the simulation; they do not have a specific economic meaning. In the list of variables, the labels having a (P) appended refer to parameters while the labels with an integer refers to variables. The integer indicates the number of lagged values used in the model for the variables: it means that there is an equation in the model which requests the value of that variable with that lag. Consequently, in order to run a simulation, it is necessary to define the lagged values which will be used during the first time step of the simulation. A variable with "(0)" after its label does not need any initial value, since no equation requests any lagged values of it.

Root contains the variable Time, representing the internal time values, and the parameter End, indicating the last step of the simulation[6]. The descendant of the Root

[5] The Browser window shows the definition of an Object, but not the numerical contents of it. Hence, it shows one single line for each type of descendants, despite their actual number of instances. Another interface, described below, allows to observe and modify the number of Objects in the model.

[6] An Lsd model can determine the end of a simulation in two ways: one is determined by the interpreter, which can be set to run no more that n steps; the other is via the equation, determining the conditions under which the simulation must finish.

are the Object Market and the Object Technopolis. It is possible to visit an Object by double-clicking on its name in the **Descendants** list. To come back to observe the Object parent of the one currently observed, you need to click on the parents name, indicated on the right of the label **Descending from:**.

The core of the model is formed by the Objects Market and Firm. The Object Market contains only two variables: P for the price and Q_TOT for the total quantity produced by the industry. The Object Firm is defined as descending from the Object Market. It contains all the variables and parameters used in the book to describe the status of a firm. A parameter is used to differentiate between innovative and non-innovative firms: "Inn" is set to 1 if the Firm is innovative and to 0 otherwise.

Descending from the Root, and therefore "parallel" to the Object Market, the model contains the Object Technopolis, which provides new technologies to the Firms (that is, new values for the productivity of Capital). This Object does not have any variable, but contains two Objects, Innovation and Imitation, expressing two different means to obtain new technologies.

Clearly, there are many possible ways to represent the same model with Lsd. The main criterion in forming the Objects should be to group together the variables of the model relating to a clearly identifiable entity. If this is properly done, a model can be very easily extended without any modification to the existing code. For example, it could have been possible to move the function done by the Object Imitation within the Firms, defining a variable in Firm as "A_IM", with the same equation. This would have implied defining an instance of this Object for each Firm, thus replicating many times the same code. Notice that the model could be run using exactly the same equations even moving Imitation as descendant of Firm; in fact, the language for the equations does not contain any reference to the Objects owing the variables. Objects are used only to retrieve a variable in the model, hence, if A_IM were moved in Firm, the only effect during a simulation run would be a different search paths when its values are requested. See the Programmers' Manual for more details on the internal functioning of Lsd.

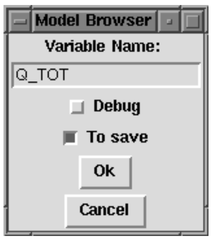
2.5 Run Time Settings

The run time settings are the options users can choose to run simulations of the same model with different seed random number generators, saving different variables in the result files, allowing a different number of steps for each simulation run and debugging different aspects of the model.

2.5.1 Variables' Run Time Settings

As you saw when you ran the simulation, some variables were saved in the results file. But there were also variables (e.g. A_IN and A_IM) that were not saved. The system stores in the initial data file also the information relative to which variables have to be saved during the simulation run. Such information can be modified double-clicking on the name of a variable in the Browser window. This produces a window setting the run time status of the any instance of that variable in the model; Figure 6 shows the one for Q_TOT.

<Figure 6 - Modifying Variables' Properties>



Q_TOT

Any modification operated through this window will be replicated on all instances of the variable in the model, that is, for any variable in the different instances of the same types of Objects.

The window allows to decide whether to save the variable in the result file. Clicking on the **To save** label, you will switch the option to save it or not. The **Debug** option determines whether this variable will cause the debugger to stop when it changes value, when the simulation is run with debug mode (see the Debugger section).

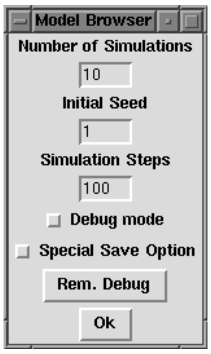
The window in Figure 6 can be also used to modify or delete variables (or parameters) in the model by typing a new name in the entry box or deleting the existing and clicking on **Ok**. The modification will affect every instances of the Objects containing the variable. Do not modify the names because the equations are "linked" to their variable only by using the names. If you try to modify the name without modifying the equation file (and recompiling the system) the simulation will abort, since it is not able to find the correct equation for the variable with the new name.

Clicking on **Cancel**, you will come back to the main Browser window without modifying the status of the variable.

2.5.2 Simulation Manager

Lsd offers the possibility to run a set of simulation runs of the same model, using different seed generators. Click on **Sim. Man**. to obtain the window shown in Figure 7.

<Figure 7 - Simulation Manager>



Number of Simulations: 10
Initial Seed: 1
Simulation Steps: 100

The first entry box, from the top, allows to decide the number of simulation runs. Setting a number larger than one, tells Lsd to make a simulation run, then to re-load the model (always using the same initial data), to set a new seed number (and a new result file name), and to make a new run.

The initial seed defined in the second entry box initialise the random number generator for the first run. The subsequent runs (if any) will use the seeds obtaining summing one to the seed used in the previous run. Allowing to set the initial seed for a set of simulation runs, Lsd allows to extend the number of experiments made with a model, in case the simulation runs already performed did not provide enough results. The seeds are also used to name the result files for each simulation run, as explained above.

The last entry box determines the number of steps for each simulation run. This value represent the maximum number of steps for a simulation, but the simulation can also be stopped earlier. The modeller can define a variable in the model (and therefore an equation) which determines the conditions under which a simulation run should stop. In this example model, the termination is determined in a very simple way: the variable Time in Root quits the simulation when its value (i.e. the number of steps done) equals the value of the parameter End. In general, the stopping equation could be more sophisticated. For example, it could stop the simulation if the total production falls to zero or if the trend of a variables in the last n iterations is constant. In this example, both the “internal” condition and the “external” condition determine the simulation to stop at the 100th step.

A check box in the Simulation Manager window allows to set the Debug Mode for the simulation run. If this box is checked, the interpreter will provide a set of information about the model. See in the Debugger section for further information.

The last check box, called Save Special Option, activates a different way to save the result. It is still under testing, and should not be used.

The button **Rem. Debug** removes all the debugging options for any variable. See below in the Debugging section for further information.

2.6 Initial Data Settings

The initial data for a simulation run are of two types: number of elements (e.g. how many Firms) and the initialisation values (e.g. initial capital stock for each Firm). The first type of values affects the number of values of the second type: changing the number of Firms will also change the number of capital stocks to be initialised. Lsd re-computes, at each action of the user, what values are needed to initialise a simulation run. Two editors, described in the next two paragraphs, allow users to define the two types of data. These editors, following the general philosophy, are completely model independent; they “read” the model currently loaded in the system and prepare the window to present and edit the relevant information relative to that model. The same interfaces are used for any model, since they adapt dynamically to any Lsd model.

The approach used to design the interfaces is to facilitate as much as possible the insertion of data. This means that, for any action of the user, Lsd tries to “guess” the necessary consequences, rather than asking the user to fill the missing information. Before the actual simulation is run, however, the user must anyway explicitly confirm these guesses.

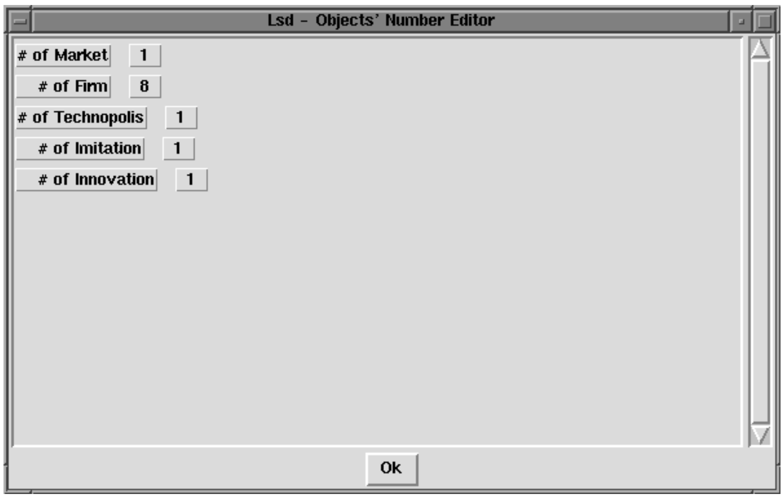
Any modification made to the model, included the modifications made to the initial values, will be saved in the files named after the model's name, when the simulation is run. If you made some modifications but you don't want to overwrite the original files, you need to change the model name, before running the simulation.

2.6.1 Setting Objects' Number

In general, Lsd allows to modify the number of any type of Object in the model (but the Root). For example, you can run concurrently the simulation for every setting presented in the book: just create as many Objects Market as necessary and set the parameters for each Market (and Firms descending from them) to the proper values.

To observe and modify the number of instances of the Objects in the model, you need to enter the Objects' Editor by clicking on the button **Edit Obj.**. For the data set in the "nw82.par" file, the window will appear as in Figure 8.

<Figure 8 - Object Editor>



# of Market	1
# of Firm	8
# of Technopolis	1
# of Imitation	1
# of Innovation	1

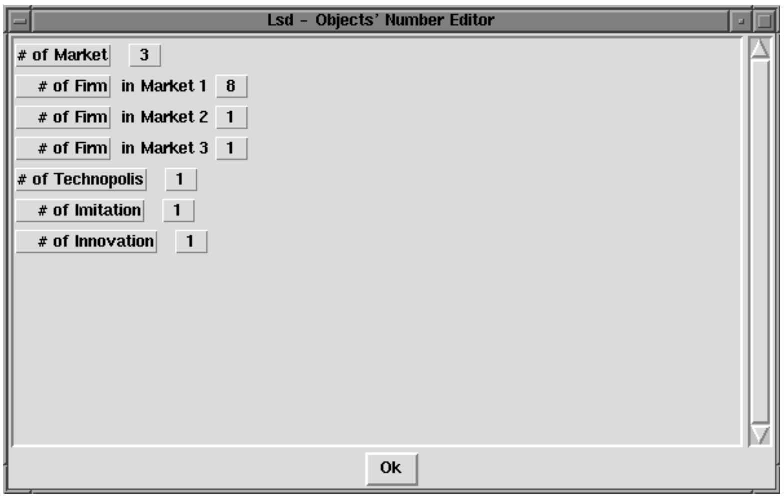
This window reports the number of instances of all Objects in model. The indentation indicates the level in the hierarchy: Firm, Imitation and Innovation are in a lower level than Market and Technopolis and hence they are indented. You can modify the number of instances of an Object clicking on it. For example, click on the number 8 corresponding on the Object Firm and enter 16 to increase the number of Firms.

The system behaves differently when the user increases or decreases the number of instances of an Object. If the user increases the number, the system appends the new instances to the end of the list of the existing ones. Instead, if the user decreases the number, the system allows to choose two different strategies for deletion: to delete instances starting from the bottom of the current list, or to specify individually the instances to be deleted. In the second case, one must insert the position numbers of the candidates to the deletion in increasing order[7]. Try to decrease and increase the number of Firms using different methods to see how it works.

[7] The system shows, in this case, a sequence windows, one for each instance to delete. Users are supposed to type, in each window, the number of the instances they want to delete. For example, inserting 2, 5, 6, Lsd will delete the second, fifth and sixth instance. The system forces users to insert the numbers in increasing order, that is, you cannot type 5, 2, 6. In case of mistakes, you can click on

In general, if an Object A has a descending Object B, the numbers of instances of B might differ across instances of A. As an exercise (this is an Object with descendants) try to modify the number of Markets increasing it to 3 and you will obtain what is shown in Figure 9.

<Figure 9 - Multiple Branching Models>



# of Market		3
# of Firm	in Market 1	8
# of Firm	in Market 2	1
# of Firm	in Market 3	1
# of Technopolis		1
# of Imitation		1
# of Innovation		1

You can see that the lines reporting the number of Firms have become three, referring to the three different sets of descendants from Markets. Each line for Firm has attached another label and a number referring to the Object they are descending from (Market) and its instance number (1, 2 or 3). The first line for Firm (indicating the number of Firms descending from the first Market) refers to the old set of Firms (it contains 8 instances). A single new Firm has been generated automatically within each of the new Markets. The system always assumes that the newly created instances of an Object have a single instance of any descending Object. Of course, you can now modify the number of descendants[8]. Note that Lsd does not allow to set to zero the number descendants, at this stage, because this would make confusion between the definition of a model where that type of descendant exist and another model where they do not exist. The Lsd language for writing the equations, however, allows to add and delete Objects, and hence can happen that, during a simulation run, some Objects have zero descendants.

The interface for editing the number of instances of an Object creates a code (here the numbers 1, 2 and 3) which is attached to the names of the descending Objects (i.e. Firm) in order to identify which instance they descend from. We will see later how this code is used for more complex situations. Delete now the two new Markets, by setting the number of Markets to 1 and selecting **Last** when asked which Markets to

Cancel to exit from the window. In case the user insert illegal values (for example, larger that the existing number of instances) the windows exits automatically, deleting only the ones indicated with a legal number.

[8] Actually, you need to change the number of Firms for this model because the equations of the model cannot be computed with only one Firm in a Market: they use the complement to the market share as denominator in the equation for the computation of new capital and hence there will be an error when trying to run the simulation.

delete. To exit from the Object Editor and return to the Browser window, just click on the **Ok** button.

When new instances of an Object are created, the system initialises variables and parameters for them to default values (it copies the value of the very first previously existing instance of the new Object). All variables have a switch indicating whether their values have been placed as defaults by the system. If there are any variable with the default values, the system will block an attempt to run a simulation, signalling the default initialised variables. The user must always “set” the values for new Objects. This does not necessarily mean changing them; if one simply observes the default values with the Data Editor (described below) the system will confirm them and allows to run the simulation.

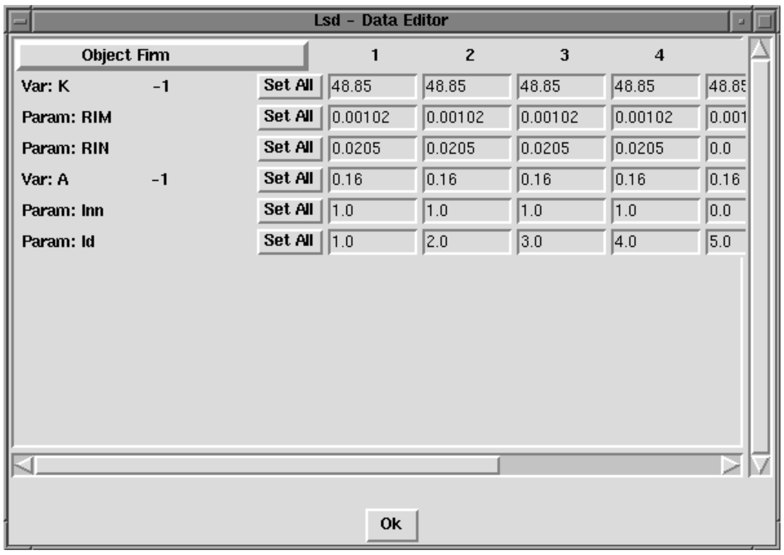
2.6.2 Setting Internal Object Initial Data

A simulation is a sequential updating of the values of every variable in the model, at each time step. The variables values are computed as functions of parameters (i.e. constant variables), present time variables and lagged variables values. A model whose equations use only present time variables is either uncomputable (each variable depending circularly on itself) or depending only on the constants of the model (the parameters). Hence, in general, there are variables used in the equation with a lag. To compute the first step of the simulation, the system needs to have the values for every parameter and lagged variable. Clearly, the number of initial values to set depends on the number of instances of the various Objects. For example, if you loaded a model ready to be run but you increased the number of Firms, you need to set the initial values for the new instances, before you can run a simulation.

Lsd provides a Data Editor which analyses the model and prepares an interface to observe, and possibly modify, the necessary initial values (and only them: the interface does not show variables which are not needed as starting values for the simulation). This interface can be used for one single type of Object at a time: you need to reach the desired Object, using the Browser, and then click on **Edit Data**. For example, go to the Object Firm and open the Data Editor. The window is reported in Figure 10. You can show the same window by clicking with the right button of the mouse when the pointer is on Object Firm.

The window indicates (label on the top left corner) the Object shown and the initial values, by rows, for all instances, by columns.. The labels before the rows refer to the variables/parameters. If the row refers to a lagged variable, the lag is indicated after the label of the variable.

<Figure 10 - Data Editor>



Lsd - Data Editor

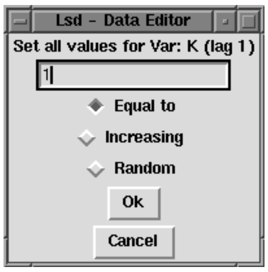
Object Firm			1	2	3	4	
Var: K	-1	Set All	48.85	48.85	48.85	48.85	48.8
Param: RIM		Set All	0.00102	0.00102	0.00102	0.00102	0.001
Param: RIN		Set All	0.0205	0.0205	0.0205	0.0205	0.0
Var: A	-1	Set All	0.16	0.16	0.16	0.16	0.16
Param: Inn		Set All	1.0	1.0	1.0	1.0	0.0
Param: Id		Set All	1.0	2.0	3.0	4.0	5.0

Ok

The numbers above the columns refer to the instances of the Object Firm. This window has some properties to facilitate data entry: when the focus is on a cell, a message (shown just above the button **Ok**) specifies the variable label and the code of the Object instance to which the cell refers; pressing the Return key, the focus is moved to the next cell (by row; that is, the cell of the next instance, for the same variable or parameter).

If there are many instances of an Object, inserting the data one by one can be very tedious and error prone. For this reason, a function to insert a sequence of values for a parameter or a variable in the various instances is provided. For example, suppose you want to set the lagged value of K to the same value, say 1, for all Firms. Instead of repeatedly inserting 1 in each cell, you can click the button **Set All** corresponding to the line of **K-1** row. A window like the one in Figure 11 will appear.

<Figure 11 - Setting All Data >



Lsd - Data Editor

Set all values for Var: K (lag 1)

1

[✓ Equal to]

Increasing

Random

Ok

Cancel

It shows the variable (or parameter) being edited and provides three choices: to set the same value in every instance; to set increasing values; to set a random values in every instance. You need to enter a value in the entry box, to choose an option and to press **Ok**.

The different options use the value in the entry box in different ways. Choosing **Equal to** the entered value is assigned to all instances. Choosing **Increasing**, the entered value is used as the step of an increasing sequence along the instances. The starting value of the sequence is taken to be the value of the very first instance (the one in the first cell, that is, the cell immediately on the right of the button **Set All** you pressed). Choosing **Random** the system assigns to each instance a random value drawn from a uniform distribution whose lower bound is the value of the first instance and the upper bound is the entered value. Make some trials to see how the different options work. The values will not be modified if you press **Cancel** to close the window or if you don't specify any choice option.

2.7 Code for Identification of Multiple Objects

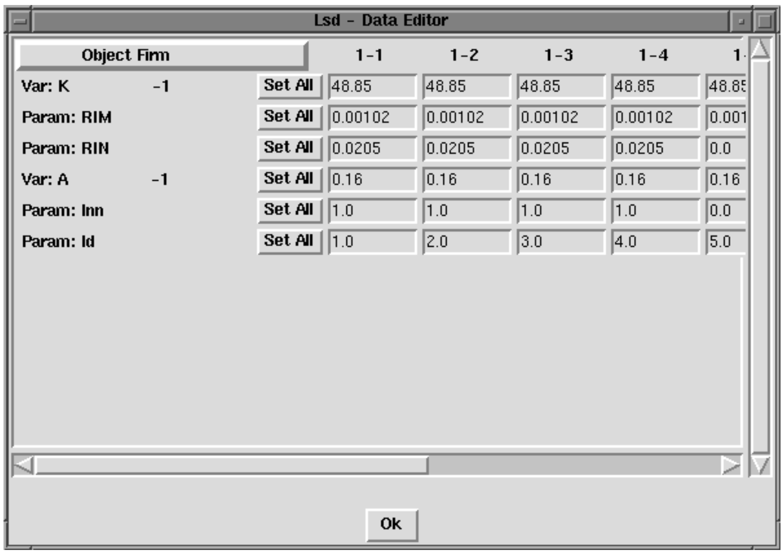
In the original data file, there is only one instance of Market; a Firm can then be identified by a single number referring to its position in the set of Firms descending from the unique Market. But if we had many Markets, how could we distinguish, for example, the variables of the first Firm in the first Market from the ones of the first Firm in the second Market? To see how Lsd manages this problem, increases again the number of Markets to 3. As we saw before, the two new Markets will have a single instance of the descendant Firm, while the existing Market will keep the same number of descendants.

HINT

The two editor interfaces (Data and Object Editors) have a shortcut to pass from one another. While in the Data Editor window, clicking on the Object label (upper left corner) will show the Object editor. While in the Object Editor, clicking on the name of an Object will show the Data Editor for that Object. Try to use these shortcuts: you always go back to the previous window by clicking on **Ok** in the current one. The shortcuts are recursive calls; you must always follow the same pattern to come back after a sequence of them. Hence, if you “shortcut” too many times in a row, “popping” back will take quite a while.

If you successfully increased the number of Markets, the Data Editor for the data in Firms will appear as in Figure 12.

<Figure 12 - Data Editor with Two-level Code>



Lsd - Data Editor

Object Firm			1-1	1-2	1-3	1-4	1
Var: K	-1	Set All	48.85	48.85	48.85	48.85	48.85
Param: RIM		Set All	0.00102	0.00102	0.00102	0.00102	0.001
Param: RIN		Set All	0.0205	0.0205	0.0205	0.0205	0.0
Var: A	-1	Set All	0.16	0.16	0.16	0.16	0.16
Param: Inn		Set All	1.0	1.0	1.0	1.0	0.0
Param: Id		Set All	1.0	2.0	3.0	4.0	5.0

As you can see, the columns are not headed by a single integer referring to instances of Firm, but by two digits, separated by a hyphen: the first digit refers to Markets and the second to Firms. The code is automatically generated by parsing the tree of the model: at any node in the tree a new digit is generated if, and only if, there are many instances of the same Object type, and increasing integers are assigned to the instances. The code avoids the use of redundant digits: scroll horizontally to observe how the last two columns are coded. A single digit is sufficient to identify the columns relative to the single Firm in the second and third Market.

The same code is used in the result file to distinguish variables in multiple instances of the same type of Objects (the hyphen "-" is replaced with an underscore "_" to avoid possible misinterpretations by some spreadsheets).

At this point, you can replicate every simulation presented in the book for the BANK=1 case. As an exercise, you can simulate the 2, 4, 16 and 32 Firms versions, taking care of setting the proper initial values (reported at page 302 of the book). You can also modify the seed random number generator[9] and the number of steps for each simulation, by using the Simulation Manager. Take care of assigning a different name to each model, so to not overwrite the result file models and results files.

2.8 The Equations of a Model

We have seen how to make "run time" changes to some parts of a model. Those regard what the Lsd system reads (and hence allows to modify) before each simulation is run with a given model. Another crucial part of a model is constituted by its equations, expressing how the value of each variable is computed as a function of

[9]Lsd changes automatically the seed after a simulation run. If you change model (i.e. 2, 4, 8, etc. Firms), you may want to use always the same seed.

other variables. They are coded in C++ by the modeller and linked to an instance of the Lsd system.

The equations of a model are written in a source file designed in such a way that anytime a variable updating is requested[10], the entire file is scanned. Hence, the order in which the equations are inserted in source file does not matter. The code for the equation is generally used by each variable once for each time step of the simulation. If the value of the variable is requested many times during the same time step, the system returns the previously computed value, without re-computing the equation. It is possible to explicitly write an equation such that it recomputed anytime it is requested, even during the same time step. This is necessary when writing, for example, the equations using random variables: any request must draw a new value. In the Nelson and Winter example, the variables A_IN and A_IM have equations which are re-computed at any request by the variables A in Firms.

The actual order in which the equations are computed during a time step is not relevant, in Lsd. The interpreter scans the model, starting from Root, and asking for the values of every variable in every Object of the model. Of course, given the way the equations are expressed, the value of some variables cannot be computed unless other values are computed first. For example, the interpreter requests the value for Q_TOT in Market before the values of variables Q in Firms, because Market is "higher" than Firm in the model structure. The system begins to compute the equation for Q_TOT, and this triggers the computations for every variable Q. Hence, when the equation for Q_TOT is completed, the variables Q have been updated; in the following, when the interpreter will request the value for each Q, they will return the already computed values, without computing again the equation.

To modify and add equations you don't need to be an experienced C++ programmer. An equation is an elaboration of numerical values (Lsd uses only real numbers). The elaboration are expressed with the C syntax, while the values to be elaborated are provided by the Lsd functions. The C code, for most of the equations, makes use of the set of mathematical operations (e.g. +, -, log, exp, etc.) and the standard programming tools (e.g. if-then-else). Here I will describe the most common Lsd function used to retrieve the values of other variables in the model. More sophisticated functions (mainly based on the one described here) are described in the Programmers' Manual, together with a brief summary of the C++ grammar.

You can observe the file of the equations (here we use the "fun_nw.cpp") with a text editor. The file contains a sequence of blocks of the type:

```
if(!strcmp(label, "XXX"))
{
// here is the code for the equation of variable XXX
res=...;
done=1;
}
```

each devoted to the computation of a single variable of the model, in the example the variable XXX. In each block, `res` is assigned the result of the equation, and `done` is

[10] There are two possible ways to trigger the computation of the equation for a variable: either it was requested by the Lsd interpreter or it was requested by another variable in order to compute its own equation.

a flag indicating that the variable has been computed. If you forget to use the flag in the end of a block, the system will give an error as if there was no equation coded for the variable.

For the actual computation (in the lines of the block before `res=...`), you can use any C++ legal code, plus the functions provided by Lsd, as parts of the definition of Object. Among those, the most frequently used is

```
cal(“YYY”, 0);
```

which returns the value of the variable called “YYY”, with its present time value (i.e. lag 0, indicated by the second element of the `cal(...)` function).

To express the equation for a variable XXX as the sum of the present value of variable YYY and the lagged value of variable ZZZ, the block can be written as:

```
if(!strcmp(label, “XXX”))
{

res= p->cal(“YYY”, 0)+ p->cal(“ZZZ”, 1);
done=1;
}
```

The code above corresponds to the equation $XXX_t=YYY_t+ZZZ_{t-1}$.

For the sake of clarity, we generally used temporary variables (the elements of a vector `v[…],` each one used as a temporary variable) to place the results of partial computations. We prefer to write the block above as:

```
if(!strcmp(label, “XXX”))
{
v[0]=p->cal(“YYY”, 0);
v[1]=p->cal(“ZZZ”, 1);

res=v[0]+v[1];
done=1;
}
```

It is important to keep in mind that the equations are part of the variables and that, in a Lsd model, variables are connected to the rest of the model through the Objects owning them. Hence, whenever in an equation the value of another variable is the needed, it is necessary use an Object which, through its Lsd functions, will explore the model providing with the desired value. Using the C++ notation, the way to express the function using an Object called `object` is

`object->cal(...)`[11].

There are two Objects that modellers can use in the code of the equation for a variable: the Object owning the variable itself and the Object owning the variable

[11] We actually use pointers to Objects, that is the memory address of an Object.

which triggered the computation of the equation. Depending on the nature of the variable, the code for the equation can use one or the other Object.

The file for the equation provides by default two Objects which modellers can use to call the `cal(..)` function. The most commonly used is the Object called `p`, hence the function must be used as

```
p->cal(...)
```

The Object `p` refers to the Object containing the variable whose equation is computed. For example, in the equation for Q, contained in Firm, uses:

```
p->cal("K", 1)
```

to obtain the value of the variable K (with one lag) contained in the same Firm of the computing Q.

A variable requested in an equation is not necessarily contained in the same Object. For example, consider the equation for PROF (profits), which is a variable contained in Firm. It uses the value of P (price) contained in Market. The modeller does not need to notify in the equation the different positions of the requested and of the requesting variable. The `cal(...)` function, in fact, will start a search of the requested variable through the entire model, returning the "closest" one, in case of multiple instances of the desired one. Hence, if we set the model so to have many Markets, the equations for PROF in each Firm will provide the value of the variable P in their own Market.

The function `cal(...)` uses a strategy to explore the model searching for a variable. First, it explores the variables contained in the Object indicate in the equation file (in the example below, p). If the search fails, it asks to continue the search to the first descendant, then to the second and so on. If there are no descendants, or they don't find the requested variable, it sends the same request to its parent Object. This strategy ensures that the entire model is scanned by following the strategy:

- search here; if not found
- search down; if not found
- start a new search up

But there are cases when this strategy does not provide the "right" result, that is, the variable found is not the one the modeller wants to use. For example, consider the equation for A_IM, providing Firms with the new productivity obtained by imitation. This equation uses the values of some variables in the Firm which requested its value (K, RIM). But the Object Imitation, containing A_IM, is in a branch of the model "parallel" to the one where the Firms are. Hence, if A_IM would request a variable in Firm, it cannot individuate the "right" Firm, but will always use the variables in the first Firm descending from Market. In fact, if the strategy of search started from the Object Imitation, it would (you can follow on the Structure window):

1. fail in the first Object, that is Imitation; not finding any descendant would go up in Technopolis;

2. fail in Technopolis and in both descendants Imitation and Innovation
3. go up in Root, and fail
4. search in Market, and fail
5. search in the first Firm and succeeds

The problem is, of course, that also when Imitation is requested by the second, or other Firms, it would incorrectly use always the K value of the first Firm.

In this case, the search must start from the Firm which originated the request for A_IM. Instead of using the line

p->cal(“K”, 0)

which indicates to start from Object Imitation (remember that p is always the Object containing the variable of the equation, where p is for parent of the variable), the function `cal(...)`will be expressed as:

```
c->cal(“K”, 0)
```

where c-> is the “caller” Object, that is, the instance of Firm which originated the request for A_IM.

In general, the `p->` Object is used whenever the variable is located either in the same Object as the computing variable or “up” in the hierarchy. It can be used also when the searched variable is unique in the model, since the search strategy is designed to pass through every Object in the model. `c->` is instead used when there are many possible Objects having the desired variable and these can be reached only starting from the Object owning the computing variable (see the graphical representation of the model).

Besides the `cal(...)` function, there are a series of other functions in Lsd which exploit the tree-like structure of a model to perform special computations. For example, you can see the `sum(...)` function used in the computation of Q_TOT. See the Programmers’ Manual for the list of these functions and to get more details about the code for the equation of a model.

Remember that any modification made to the equation files does not affect the model, unless you link the new equation file to the rest of the system. If you wants to modify (or add) an equation, you need (after having saved the “fun_nw.cpp” file) to: exit from Lsd, if you are running it; compile the system, using “make” under unix or producing a new executable with the window compiler; re-run the new Lsd, containing the modified version of the equations; reload the model.

2.9 Simultaneous Equations

Lsd is thought to run discrete time models, since they are the closest to the actual functioning of a computer program. This implies that two variables cannot be computed at the same virtual moment within a simulation step, if they depend on one another. This facilitates enormously the implementation of discrete time models, but it constitutes a severe problem for (pseudo) continuos models. Lsd offers the modeller a way to overcome this problem: the computation of each variable must always be separated from the one of others variables, but the actual value of a variable can be stored during the computation of another variable. For example, consider two variables needing to be computed at the same time, say XXX and YYY. The modeller will write their equations as follows:

```
if(!strcmp(label, “XXX”))
{
// Here is the computation of both variables
// whose results are stored in v[0] for XXX and in v[1] for YYY
res=v[0];
p->write(“YYY”, v[1]);
done=1;
}

if(!strcmp(label, “YYY”))
{
// Does nothing but ensuring that XXX makes all the job
// before the end of the present block
p->cal(“XXX”, 0);
// Now the value for this variable is stored in val[0]
// and the block can safely return its own value
res=val[0];
done=1;
}
```

The first block makes the necessary computation and assigns the results for XXX to `res`, but it also writes the value for YYY using the Lsd function `write(...)`. The second block requests the value of the first variable: this triggers the assignment to YYY made in the first block, hence, before the next line is executed, the content of `val[0]` for YYY is ensured. The result of this second block is simply its current value for XXX (in any variable, `val[0]` stores the most recently computed value, `val[1]` the previous time value and so on).

With this “trick”, Lsd can perform the computations of a set of variable whose values depends on one another, as, for example, the numerical solution of a set of differential equations.

2.10 Adding a Parameter to a Model

The current implementation of the Nelson and Winter model represents the case reported in the book as BANK=1 (that is, firms can borrow up to one time their profits in order to invest in new capital). Let’s see how to modify the model in order to insert a parameter, call it BANK, which allows users to freely set the percentage of profits Firms can borrow from the bank.

The equation which uses this parameter is the one computing the new capital; hence you need to search in the file fun_nw.cpp the block:

```
if(!strcmp(label, "K"))
{
  v[0]=p->cal("K", 1);
  v[1]=p->cal("P", 0);
  v[2]=p->cal("PROF", 0);
  v[3]=p->cal("A", 1);
  v[4]=p->cal("Q", 0);
  v[5]=p->cal("Q_TOT", 0);
  if(v[2]<=0)
   v[6]=v[2]+0.03;
  else
   v[6]=v[2]*2+0.03;

  v[7]=v[1]*v[3]/0.16;
```

```
      v[8]=v[4]/v[5];
      v[9]=max(0, min( (1.03 - (2-v[8])/(v[7]*(2-2*v[8]))), v[6] ));
      res=(v[0]*0.97+v[9]);
      done=1;
    }
```

This block contains the equations on pages 302-303 of the book. The first operation is to write the command to retrieve the new parameter BANK. Thence, we must insert a line like:

```
    v[10]=p->cal(“BANK”, 0);
```

The value obtained must be placed in the computation for v[6] (f(π) in the book). The resulting function is:

```
    if(!strcmp(label, "K"))
    {
      v[0]=p->cal("K", 1);
      v[1]=p->cal("P", 0);
      v[2]=p->cal("PROF", 0);
      v[3]=p->cal("A", 1);
      v[4]=p->cal("Q", 0);
      v[5]=p->cal("Q_TOT", 0);
      v[10]=p->cal(“BANK”, 0); // Here is the new parameter

      if(v[2]<=0)
       v[6]=v[2]+0.3;
      else
       v[6]=v[2]*(1+v[10]) +0.3; // Here is where BANK is used

      v[7]=v[1]*v[3]/0.16;
      v[8]=v[4]/v[5];
      v[9]=max(0, min( (1.03 - (2-v[8])/(v[7]*(2-2*v[8]))), v[6] ));
      res=(v[0]*0.97+v[9]);
      done=1;
    }
```

At this point, we need to recompile the example model. Use again the “make” command (see the section “Compiling the Nelson and Winter Example” above), which will recognise automatically the modification to the fun_nw.cpp file and will produce a new version of the model.

Run again the example model and load the data as before. Try to run the simulation. As you should have expected, the system tries to compute the equation for K and, hence, request the new parameter BANK. But the structure of the model is still the old one, without any parameter called BANK, and hence the interpreter exits with a message communicating the name of the variable (or parameter) not found.

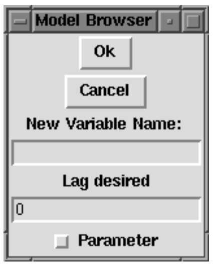
Restart the system and load the model again. To insert a new variable or parameter in the structure of a model you need to click on the “Add a Variable” button, after placing the browser window on the Object in which you want to make the insertion. We need to decide where to place the new parameter. The Lsd interpreter does not care where it is: when the program reaches the line requesting BANK, the interpreter begins a search which is stopped only when a parameter or a variable with that label is found. Given the search strategy, if the parameter is in the same Object of the calling variable (K in Object Firm), it will be the one returned without further search. Otherwise, the search continues up in the tree.

We could set the parameter to be equal for all Firms (as in the original model), or Firm specific (implying a varying reliability of Firms, and hence a differentiated access to credit). In the first case we can place the parameter in Market, so that it can be accessed by every Firm. In the latter case, we must place the parameter in Firm, so that there will be a different instance of BANK for each Firm, and it will be possible to attribute different values for different instances. In a further version of the model, the fixed parameter could be transformed in a variable, whose value could be computed as a function of, e.g., the past profits of the Firm. In all this cases, the use of BANK will not change, and we will never need to modify the equation for K discussed above. Of course, we will need to add an equation for the computation of BANK.

Clearly, where to position parameters and variables depends, besides the technical point, on the economic interpretation. Notice that, technically speaking, if the parameter/variable appears only once, it can potentially be located in any Object instantiated with only one copy in the model (the interpreter will eventually find it).

Go in Market (or whatever Object you decided to assign the parameter) and then press **Add a Variable**. The window will become as in Figure 13 (the window is the same to insert either a variable or a parameter). Since we need to insert a parameter, check the corresponding box on the bottom of the window. Write the name of the parameter and press **Ok**.

<Figure 13 - Adding a Variable>



If you had written “Bank”, instead of BANK, the interpreter will exit as it did before, since the labels are case sensitive. To modify the label of a variable, you can double-click on it an change it, as we saw before. In case you had forgotten to check the parameter box, and hence you have created a variable, you cannot modify this. The same applies for the number of lags in the case of creating a variable. Then, you need to delete the wrong variable (by assigning to it an empty string as a name) and to create a new one, with the correct characteristics.

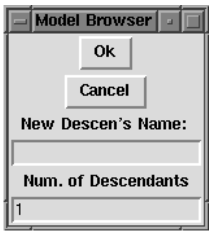
Before running the simulation, we still need to set the parameter value. When a new variable or parameter is created, the system sets its value to 0 by default, but (as we have seen) does not allow to use it for an actual simulation run. You must enter the Data Editor in correspondence of the Object owning the new variable/parameter and set its value (or just “observe” the default one, thereby confirming it). After you have done this you can run the simulation.

2.11 Adding New Types of Object

Let us make an example of how to modify a model's structure adding Objects. The modification we will examine is only a formal one, not modifying the behaviour of the model; we will move the variable K from the Object Firm to a newly created Object Capital. In this way, we will not have to modify the equations or add other variables, but the structure will be ready for further modifications.

Go with the Browser on Object Firm and double click on the variable K. In the entry box containing the name of the variable insert an empty string (that is, delete the existing "K") and click on **Ok**. The Browser window will not show the variable K anymore. Now, click on the button **Add Descendant**. The resulting window is shown in Figure 14. It allows to insert a label for the new Object, and the desired number of instances. As you know, the Browser actions modify the entire model in the same way. This means that the descendant will be created within each Firm, and in the same number of instances. Afterwards, you can specify different numbers of descendants using the Object Editor.

<Figure 14 - Adding Descendants>



New Descen's Name:

Num. of Descendants
1

Enter "Capital" as the name of the new Object, and 1 as number of descendants. The Browser will show the Object Firm having "Capital" as descendant. Double-click on it to move the Browser on it. The newly created Objects are obviously empty. Now you need to insert here the variable K you previously deleted from Firm. Click on **Add Variable**. Since K is used in the equations with a lag[12], you need to type 1 as number of lags for the variable. The user must always specify the lags for a variable, depending on the use made of the variable by the equations for other variables. In case of errors, the interpreter stops the computation and issues a message with the indications to fix the problem. Click on **Ok** to return to the Browser window.

If you correctly set to 1 the lag for K[13], clicking on **Data Editor** you will be shown the single line for the initial values of the lagged K. As you see, the system initialised the value to 0 by default. Click **Set All** and insert the value 48.85 (this is the common initial value of K reported by the book for the case of 8 firms). Set the **Equal to** option and click on **Ok**.

[12] The capital is updated when the profits have been computed. But the equation for profits is based on production, whose equation, in turn, uses the capital. From the description of the model in the book, previous period capital is used for production, and the new capital value is computed at the end of the period. Hence, we need to store the lagged value for capital.

[13] The Data Editor scans the Object and presents only the parameters and the lagged variables which must be set to initialize a simulation. A variable with zero lags would not need an initial value and hence would not be shown by the Data Editor.

Now the model has been restructured so to have a new Object Capital as descending from Firm. The new Object contains only the variable K. We do not need to introduce an equation for K, because the model will use the previously existing one. In general, when inserting a new variable one must add an equation for it and recompile the system. In our example, though, we have moved an existing variable in a newly created Object; the only difference is that, when the simulation is running, the interpreter will have to move between the Objects Firm and Capital in order to fetch the values of the variables. The fact that the equation for K (as well as the ones for PROF, Q etc. using K) do not need to be modified, shows the power of Lsd in exploiting the structure to ease the burden for the modellers. As we saw, the modeller does not indicate a specific Object for the variable needed in the computation (e.g. “search variable K in the Object Firm”), which would make impossible to re-use the equations in different models. Rather, the Lsd functions are flexible (e.g. “search variable K in the Object closest to this one”) allowing to use the same code in different models.

In this example, after we moved K to the new Object Capital, any equation using K will still find it, because of the search strategy used by the interpreter to retrieve variables. Since the Capital descends from Firm, the instance of Capital for each Firm will be explored before the Capitals descending from the other Firms. There is no possibility for a Firm to use Objects Capital from other Firms or for a Capital to use values in other Firms than its own.

In order to save this version of the model, change the file name by clicking on the previous name (“nw82”) on the top of the Browser window. Insert a different name (e.g. “nw82a”) and click on **Ok**. Now you can save the model by clicking on **Save** without overwrite the other files. Click on **Run** to run the model.

The exercise you just went through does not modify at all the behaviour of the model. In general there are many equivalent ways to express a model with Lsd.

The choice among them, once again, is dictated by both technical convenience and economic interpretation. Our new version opens the door to interesting possibilities: we could create several instances of the Object Capital within Firms. Each such instance could contain, besides the variable K, a different productivity parameter. We could also change the investment equation (i.e. the equation for K) to differentiate new technologies from previously known one. In the first case, the investment equation will create a new Object Capital with a new value for the productivity parameter and K equal to the investment itself. In the second case, the equation will just increase the K in the Object Capital owned by the firm.

Such changes will obviously require modifications and additions to the equations relative to the variables involved. But we could re-use the rest of the model.

In the Programmers’ Manual we describe two different Lsd functions for dynamically adding new Objects. You can see them at work in the other two example models, both of which use dynamic creation of Objects.

2.12 Debugger

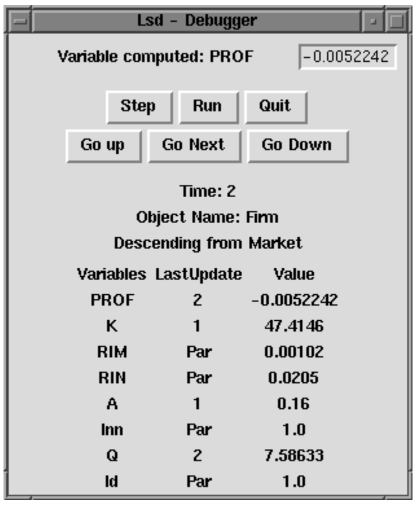
While running the first simulations of a new model, it is often useful to follow the simulation step by step, to check for possible bugs in the equations. Given the object oriented structure of the models represented in LSD, there is no global scheduling of the operations, and hence no possibility to set breaks at some step in the simulation, as in a standard debugger. The LSD debugger is based on occurrence of events changing the values of a variable: if the simulation is run in Debug Mode, the debugger stops it immediately after the computation of selected variables and allows to observe the current values of those, as well as any other one. It is also possible to set conditional breaks: the debugger will then stop the simulation only when a given variable value satisfies a certain condition.

To set the debug mode, one must check the box “Debug mode” in the Browser window before running the simulation. The user must decide what variables he wants to follow, and check on the box **Debug** of the window shown in Figure 6 (obtained by double-clicking on the variable name in the Browser before running the simulation[14]). Every instance of a selected variable will be followed (if you decide to follow the variable PROF in Firm, the debugger will stop the simulation after the computation of PROF in each Firm). During the debugging session, though, one can select a subset of instances to be followed. Remember: if the simulation is running in Debug mode but no variable has been set to be followed, the debugger will never stop the simulation until its natural end.

2.12.1 Debugger Window

After having set the debug mode and having determined what variables to follow, click as usual on the **Run** button to start the simulation. When an instance of a variable to be followed is computed, the window will automatically change, as shown in Figure 15.

<Figure 15 - Lsd Debugger>



Variable computed: PROF -0.0052242

Time: 2
Object Name: Firm
Descending from Market

Variables	LastUpdate	Value
PROF	2	-0.0052242
K	1	47.4146
RIM	Par	0.00102
RIN	Par	0.0205
A	1	0.16
Inn	Par	1.0
Q	2	7.58633
Id	Par	1.0

Unlike the Browser window used to set a model before the running of a simulation, the Debugger window does not refer to a class of Objects in general, but to one specific instance of an Object. There is no code here to differentiate between instances

[14] The information about the variables to be debugged is stored in the initial data file. You will find that some variables have already been set to be followed by the debugger. Again, you can change this, but the modification will be recorded only if the model is saved.

of the same Object; the user must deduce which instance is shown from its position in the model[15].

The window shows at the top the name of the variable whose computation caused the simulation to stop, and its computed value.

After a number of buttons (whose function is explained below), the window contains:

• the internal time step of the simulation

• the name of the Object containing the variable

• the name of the parent Object

• the status of the Object containing the variable.

The status is represented by a list of all variables and parameters contained in the Object (including the debugged one). For each variable, the time of the last computation and its value are shown[16]. For example, the variable “K” has not been computed yet: its value of last computation is in fact 1 (in the model the profits are used to compute the new capital), while the window shows that the current time is 2. Thus the value reported for “K” refers to the previous period.

The parameters of the Object, like the propensity to innovate or the flag for innovators, do not have any last update value, since they are never computed. This is indicated by the “Par” label in the last update column.

This window allows a number of actions to debug the model:

• The value of the computed variable can be change by clicking in the sunken window containing it and inserting a new one. This is useful to modify the behaviour of a model at run time and check special cases.

• Clicking on one of the **Go ...** buttons will move the browser to explore another object[17]. Browsing along the Objects in the model, the window will show the same “Time” and the same “Variable Computed”, while the Object shown depends on the direction chosen to browse through the model.

• Clicking on **Quit** will cause the simulation to abort.

[15] As an altenative, one can add a parameter,e.g., “Id” to the Object types with many instances, and assign a different parameter value to each instance. Such parameter has been included for Firms, as you can see in the debugger window.

[16] As it should be clear by now, the variables are computed as long as their values are necessary. Hence, in any moment during a simulation time step, some variables are updated to the last time step, while others still have the last period value. To keep the time consistency between variables updated at different times, the user must consider the global time and the variable specific “Last Update” field: the lagged value of a variable whose last update time is equal to the global time corresponds to the “present” value of a variable whose last update field indicates the previous global time step.

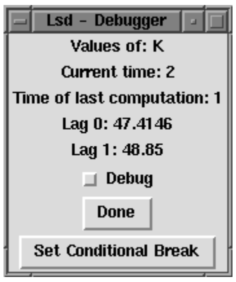
[17] It is a good exercise to familiarise with the object structure of the model. In fact, going “Up” refers clearly to the parent object, but going “Next” or “Down” is less obvious. “Next” is the sibling of the present object in the chain of descendants from the parent. It can be either another object of the same type (as in the example above) or an object of different type. “Down” means the first descendant. To reach a determined descendant starting from the parent it is necessary to go down and to move along the chain using “Next”. Going “Up” and then “Down” does not necessarily return to the starting point: if the initial object is not the first in the chain of descendants, one will need to make some steps “Next” (after “Down”) to reach it.

• Clicking on **Run** will exit the Debug mode. The simulation will run at maximum speed and it will not stop till its natural end (unless a conditional break is set; see below).

• Clicking on **Step** the simulation will proceed till the next debugger stop; that is, the computation of a variable set to be debugged, or a conditional break matched by a variable value.

It is possible to double-click on any variable in the list of elements of an Object shown in the Debugger window. This provides more information about the variable, and allows to modify its debugging behaviour. For example, clicking on the label K, the Debugger window becomes as in Figure 16.

<Figure 16 - Setting a Variable for the Debugger>



Values of: K
Current time: 2
Time of last computation: 1
Lag 0: 47.4146
Lag 1: 48.85

The window shows:

- the label of the variable you clicked on;
- the current time;
- the time of last computation of the variable;
- the list of lagged values of the variable;
- a Debug check box;
- a button to exit from this window (**Done**);
- a button to set conditional breaks.

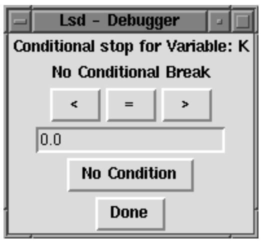
This window is used to observe old values of the variable, if they exist. In fact, the normal list of variables shows only the most recently computed value, but does not allow to observe older values. It also allows to set the variable as a debugging variable: if the Debug box is checked, the program will stop to show the variable anytime it changes values.

Selecting a variable to be checked by the debugger in the Browser window (that is, before starting the simulation) will cause to check all the instances of that variable. Instead, the Debug boxes you can access from the Debugger window (during a simulation run), affect only the instance you choose.

2.12.2 Conditional Breaks

The **Set Conditional Break** button serves to specify conditions under which the variable must be checked. Suppose, for example, that one of the variables assumes an undesired value (e.g. a negative price, a share greater than one etc.). To investigate the reason, you need to stop the simulation when this happen and to check the values of the variables used in the equation for the variable. Using a conditional break, instead of an unconditional one, avoids the tedious waiting for the bug to appear. Clicking on the **Set Conditional Breaks** button produces a window like the one shown in Figure 17.

<Figure 17 - Conditional Breaks>

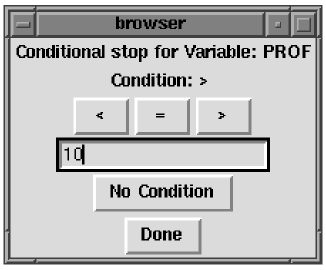


The window shows:

- the label of the variable
- the current condition set
- three buttons for a “smaller than”, “equal to” or “larger than” conditions
- an entry box for setting the threshold value
- a button to delete the condition
- the exit button, **Done**.

As an example, suppose we want to stop the simulation and control the model when this instance[18] of the variable PROF is greater than 10. We need to press on the button “>“ (causing the **Condition:** to change), and insert 10 in the numerical entry box, as shown in Figure 18.

<Figure 18 - Conditional Break>



Clicking on **Done** we will exit the conditional break window, returning to the Debugger window (Figure 15). Remember that keeping the Debug option on for the variable will cause the program to stop in any case; thus, in order to stop the simulation only if the condition is satisfied, we need to uncheck the debug option for the variable.

[18] Remember that the Debugger window accesses one individual variable at each time.

2.12.3 Scheduling Debugger

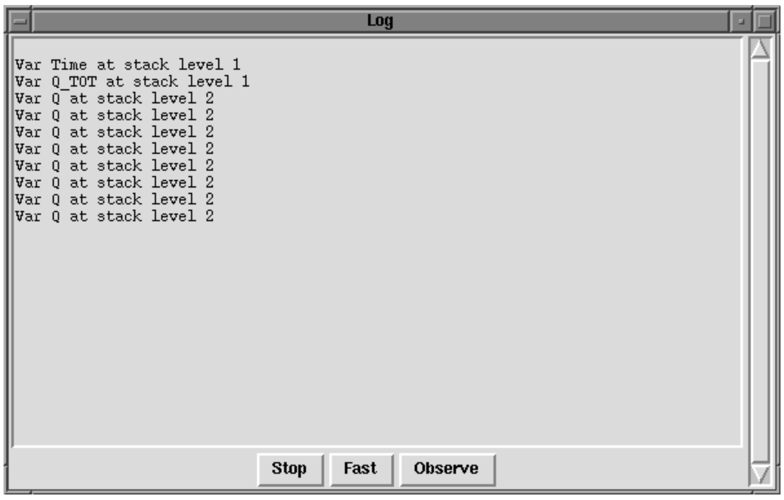
The Lsd model interpreter is able to identify some errors. When an error is encountered, the interpreter tries to finish the simulation step, assigning an error code to the equations left to be computed, and signalling the error. A typical error, is to request an Lsd function (e.g. c->cal(...) to an Object which does not exist (e.g. when c-> is NULL because the equation was not triggered by any variable). In this case, the error messages help to identify the mistake. Other errors (for example, division by zero) cannot be captured since they take place during the execution of the code in the equation file and provoke the crash of the system.

There are some type or bugs which need, to be identified, to know the actual order to execution of the equations. As said before, Lsd ensures that the priorities between the equations are respected, even though the beginning of the computations is arbitrary. This is done by using the stack: if an equation begins its computation and, during the execution of the code, recognises that other variables must be computed first, that computation is temporary suspended (placed on the stack), the other equations are executed (at a higher level of the stack), and later the original execution is continued. This nested system can also become rather complex, when many variables are iteratively placed on the stack. In case of complex errors, could be therefore important to know for any equation which other variables' computation have been suspended and are on the stack, waiting to finish the computation for their equations.

When the debug option is on, the Log window prints a line for any equation whose computation has begun, indicating the level of the stack. The stack information can be used to identify the sequence of computations triggered by the equations. Typically, this type of debugging is necessary to resolve a deadlock; that is, when two variables request each other's value in order to complete their computation, and hence an infinite loop risks to be initiated. Without the debug mode, the Lsd interpreter communicates one variable involved in the deadlock before aborting, but cannot trace back the complete sequence. Often, a deadlock appears after a long chain of triggered computations and hence it is difficult to understand how to fix the error. For example, consider having four variables to be computed. The equation for A needs the value of B, which needs the value of C, which needs the value of D, which eventually requests the value of A again, provoking the deadlock. This sequence can be mixed with other computations (any variable in the sequence can meanwhile trigger also the computation of other variables, not involved in the faulty chain, which can hide the real sequence of the deadlock). The stack level is an integer which is increased anytime a computation is triggered. If the equation for A is computed at stack level 1, then the computation for B will have the stack level 2, C 3, D 4 and A (second attempt to compute again the same variable) 4. Following the jumps in the stack levels, it is easier to discriminate the variables involved in the offending chain. In fact, suppose that the computation of C triggers the computation for another variable, say E, before triggering the computation for D. This computation will be assigned a stack level 4, but it will be not part of the deadlock chain, since the next line (for D) will have a stack level equal to 4 again, indicating that the computation for E was successfully completed and hence innocuous with respect to the deadlock.

As example, consider the Figure 19. It reports the Log window in case of setting the variable Q_TOT as to be debugged. When the debugger stops the simulation, the system have just finished to compute the Q_TOT. In order to complete the equation, it had to compute the values of Q for every Firm.

<Figure 19 - Stack Level>



```
Var Time at stack level 1
Var Q_TOT at stack level 1
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
Var Q at stack level 2
```

Variable Time was the first to be computed, since it is contained in the Root. But it did not trigger the computation of any other variable, hence the successive line is again at same stack level as the first one. The second line shows that the Lsd model interpreter requested the value of Q_TOT. Not that its stack level is 1, since Q_TOT and Time were computed because requested by the interpreter, and not because requested by other equations. In order to complete the equation of Q_TOT, the equation needs the values of every Q in the model. In the Log window you can see the 8 lines indicating the equations for Q in each Firm. As you can see, the lines for the Q's report the stack level 2, since they were computed because of the request of Q_TOT, whose computation was made at stack level 1. The lines stop there, because the figure was taken while the Debugger had suspended the simulation showing the result of Q_TOT.

In summary, there can be three different cases in two successive lines, reading the information from the stack:

• the second line has a higher levels in the stack; this means that the second variable was caused to be computed by the previous one;

• they have the same level; this means that the two variables were computed "in parallel" (as parameters to be used for the same equation of the previous stack level);

• the second has a lower level; this means that the first variable closed a cycle of computation, and the following is continuing a computation at a lower level of the stack.

Variables which are computed because requested by the interpreter (that is, whose computation is not triggered by other ones) have always stack level 1. Remember that the debugger writes a line only for computations that are actually performed; if an equation requests values of variables already computed during the same time step (or of parameters, returning values without any equation), their value will be provided without the re-computation, and no lines will appear.

The print out works only in Debug mode, and is unset by clicking on the **Run** button during the debug session. It is possible to run a simulation in Debug mode without setting the Debug option for any variable: in this case, the program will never stop till the end of the simulation (no variable to check), but the print out with the list of computations begun, and the stack level information, will be produced. This will cause the simulation to run at a sensibly lower speed.

Normally, after a debugging session, there will be many variables whose debug option is on. Instead of tracing every variable to unset the debug options, you can enter in the **Sim.Man**. window and press the **Rem. Debug**. It will “clean” all the variables, by unsetting the debug option for every variable in the model.

Chapter 3 - Nelson and Winter Model

The content of the model is divided in a number of entities representing relatively independent sections of an artificial economy. Each entity is implemented by a Lsd Object, labelled after the function each entity has in the model: Market, Firm, Technopolis, Innovation and Imitation. Another Object, called Root, contains technical elements to run the simulation; it does not have an economic interpretation, and it is used to control some technical aspects of the simulation.

3.1 Model Contents and Initial Values

An Object is defined by the variables and parameters contained. The difference between the two is that the former have an equation computing new values for each time step, while the latter have a constant value throughout the simulation.

In order to start a simulation run, it is necessary to insert in the model the values used for the first time step. They are the values for every parameter and for the variables whose values are used with a lag. The Objects' description below indicates with "(p)" the parameters and with an integer between parentheses the variables. The integer refers to the maximum lag the variable value is used by the equations in the model. In order to run a simulation of the model, the user can change the number of instances for each Object type. The user can also modify the initial values for the parameters of the model and for the past values of the variables used at the first step of the simulation. The Data Editor provides, for each Object, the list of the elements whose values are used during the first time step. They are every parameter (indicated with "(p)") in the Object and the lagged variables (the ones whose integer between parentheses is larger than zero). The tables below list the elements of the Objects, beginning with the ones which need to be initialised and provide also a short description of their role in the model.

Root

Seed (p)	Initialise the random number generator. Use the same value to have the same sequence of pseudo-random numbers or different values for having different values
End (p)	The last step of the simulation
Time (0)	It reports the time steps of the simulation. Its equation sets the flag to quit the simulation

Market

P (0)	Price.
Q_TOT (0)	Total quantity produced.

Firm

RIM (p)	Costs of research for imitation. Used for the probability to access, via imitation, a more productive technology and as a cost to determine the profits
RIN (p)	As RIM, for the research via innovation.
Inn (p)	This parameter tags the firms which can access new technology also via innovation (if Inn=1) or only via imitation (Inn=0)
Id (p)	Technical parameter used to distinguish firms.
K (1)	Capital. Its initial value is used for the computation of the quantity produced the first period
A (1)	Productivity. As the capital, it is used for the computation of the quantity produced the first period.
PROF (0)	Profits.
Q (0)	Quantity produced.

Technopolis

(no elements)	

Imitation

A_IM (0)	Provides the results of imitative research for new technologies

Innovation

A_IN (0)	Provides the results of innovative research for new technologies

3.2 Equations

The following table contains the description of the equations used in the model. For each line is reported a variable, a short description of its use in the model and a formal definition of the equation used to compute its values for each time step. If the equation makes use of the values of other variables, these are indicated by the names of the variables attached to an integer between square brackets. This integer refers to the time lag the variable's value is used in the equation: XXX[0] means that XXX's value used in the equation must be the most recent one (lag 0), XXX[1] means that the previous period value is used, etc. The formal expression of the equations corresponds exactly to the code Lsd uses to express the equations. You can observe the file fun_nw.cpp searching for blocks of code like:

```
if(!strcmp(label, “XXX”))
{ ...
res=...
done=1;
}
```

expressing the equation for the variable XXX. Remember that `cal(...)` is the Lsd function providing a value corresponding to the variable label included in the function. The value assigned to `res` is the result of the computation and hence it becomes the present time value for the variable at each time step.

The only difference between the code and the formal description is that, for clarity reasons, the variables or parameters values necessary to compute the equation are first assigned to temporary variables (`v[0], v[1],` etc) which are then used to compute the value of `res`. Then, for example, the equation for computing the quantity produced, formally expressed Q= K[1]*A[1] will be written in the code as:

```
if(!strcmp(label, “Q”))
{
v[0]=cal(“K”, 1);
v[1]=cal(“A”, 1);
res=v[0]*v[1];
done=1;
}
```

The first column indicates the Object containing the variable: F for Firms, M for Markets, In for Innovation and Im for Imitation.

F	Q	The quantity in each Firm is computed as the product productivity and capital, both with their lagged values. K[1]*A[1]
F	PROF	The unitary profits are given as the price minus fixed costs and “expenses” for innovating and imitating. P[0]*A[1]-0.16-RIM-RIN
F	K	The the capital stock is a function of an efficiency index computed comparing the ratio price and production cost versus the market share. The increase in capital stock is financially constrained by the available profits, considering also the possibility to borrow a percentage of profits from a virtual banking sector. A physical depreciation constantly reduces the capital. $MAX\left\{0, MIN\left\{\left[\left(1.03-\frac{2-\frac{Q[0]}{Q\_TOT[0]}}{\frac{P[0]*A[1]}{0.16}\left(2-2\frac{Q[0]}{Q\_TOT[0]}\right)}\right), f(PROF)\right]\right\}\right\}*K[1]+0.97*K[1]$ where f(PROF) is $f(PROF)=\begin{cases} 0.03+PROF[0] & PROF[0]\leq 0 \\ 0.03+2*PROF[0] & PROF[0]>0 and BANK=1 \\ 0.03+3.5PROF[0] & PROF[0]>0 and BANK=2.5 \end{cases}$ The function f(...) above is restricted only to BANK = 1. The extension

		for a generic value of BANK is presented as exercise in the Users' Manual.
F	A	the new productivity is the maximum among the previous productivity and the ones possibly (that is, in case of successful draw) obtained from the imitation and the innovation. MAX(A[1], A_IN[0], A_IM[0]). The same function is applied to both innovative and non-innovative Firms. The values for A_IN is zero always if the Firm is a non-innovative one.
M	Q_TOT	total quantity produced in the market, i.e. the sum of the quantities produced by each firm. ΣQ[0]
M	P	the price is determined by a demand function. 67/Q_TOT[0]
In	A_IN	This variable returns, for innovative Firms, with a certain probability a draw from a random function a new productivity value. The probability for the draw is proportional to the expenses in research for innovation (supposed as a fixed amount of the capital). The probability function used to draw the new productivity is a function whose log is a normal with constant variance and mean increasing with the time. $IF(RND < K[0] * RIN * 0.125 AND \quad Inn = 1)$ $e^{Norm(\log(0.16+t*(0.01),0.0025)}$ $ELSE$ 0 The RND returns a draw from a uniform between 0 and 1. The Norm(m, s) returns a draw from a normal with mean m and variance s). Note that the implementation of the equation uses the address of the caller Object, since the default search cannot reach the desired Object. Moreover, since this function needs to be computed anytime it is requested, it must explicitly re-set the last_update field to its previous value so to allow the next call to become effective.
Im	A_IM	This variable provide the result of the research for imitation performed by the Firms. With a certain probability it returns the value of the highest productivity currently used in the Firms. $IF(RND < K[0] * RIM * 1.25)$ $MAX_i(A[1]_i)$ $ELSE$ 0 Also this variable uses the caller address and re-sets the last_update

		field in order to be used by every Firm.

Chapter 4 - Dosi-Kaniovski-Winter Model

This model is a very early version of a still unpublished model. It is presented here only for the purpose of showing the possibilities offered by Lsd to implement simulation models. The original authors have all the credit for the model.

4.1 Model Contents and Initial Values

The model represents a market where Firms, having different productivities and different unit costs of production, produce a unique homogenous product. Since the model is meant to analyse the behavior of the market with firms having many different (but constant through the time) costs and productivities (ratio production/capital), the model contains Objects for each ratio and each costs. Firms enter the market for each combination ratio/cost. The number of firms entering is determined by a random process while the number of firms exiting from the market is determined by their dimension (i.e. the quantity of capital).

The structure of the model is made of an Object Market, from which descends a number of Objects Ratio, containing in turn a number of Objects Cost. This represent a sort of matrix having every possible combination of Costs and Ratios. The simulations tested (described by the inital values delivered with the example model) use only one single Cost and two Ratio. Of course, these settings can be changed in the initial data file. Initially, given the settings used, there is only one firm for each combination Ratio/Cost. This Firm is doomed to be eliminated at the first time step, given the values used to initialize it, so to represent initially no firms in the market.

A number of alternative probability functions are available to determine the number of Firms and their size when entering the market. The probability functions used can be selected by setting special parameters acting as switches.

The simulation cycle consist in incumbent Firms investing, new Firms entering in the market and small Firms (with capital below a threshold) dying.

For each simulation cycle a number of statistics are computed: number, age, size of Firms at different levels of aggregation (Cost, Ratio and total).

In the following tables there are the contents of each Object, the nature of their elements ("(p)" stands for parameter and "(i)" refers to variables whose maximum lag used is i). The tables are meant to be used as quick reference for determination of initial values. Their equations is described in the next paragraph.

The Object Root contains the technical elements controlling the simulation

Root

Iterations (p)	Number of iterations for the simulation
Seed (p)	Initialize the random number generator
Time (0)	Time step of the simulation

The Objects Firm are dynamically created at each time step, during the computation of the variable New_firm. When they are created, their values are initialized to default values. They do not enter in actual production immediately, but only during the subsequent time step (the parameter New_born tags the newly created Firm from the existing ones). The initial values for Object Firm to be set regard the single initial Firm(one for each Cost and Ratio: two in total, given the initial values used). This Firm is doomed to be eliminated the first step and it is used only for technical purposes.

Firm

Plate (p)	Identification parameter, used to distinguish Firms. Meaningless, used for debugging purposes.
New_born (p)	This parameter tags the newly entered firms. It is 1, if the Firm has been just created and 0 otherwise. Initially, it must be placed to 0, so that the new Firm will be deleted, given its zero capital.
K (1)	Capital of the Firm. The first Firm at the beginning of the simulation is a “dying” firm, that is, it has zero capital.
Age (1)	Age of the Firm. The initial value is ininfluent, since it the initial Firm will be deleted anyway.
I (0)	Investments per unit of capital, that is, variation of capital.
Q_firm (0)	Quantity produced by the Firm.
Treshold(0)	Minimum quantity of capital required by Firms to avoid being canceled.

The following Object’s main purpose is to contain sets of Firms, sharing the same unit cost of production and the same capital/production ratio. It contains a number of “statistical” variables whose purpose is to describe the characteristics of the set of Firms.

Cost

m (p)	Unit cost of production.
Q_cost (0)	Quantity produced by every Firm sharing the cost
Num_firm_cost (0)	Number of Firm with the same cost
Num_death (0)	Number of Firms with the same cost canceled.
New_firm (0)	Number of Firms with the same cost entered
Av_age_c (0)	Average age of Firms with the same cost.
Var_age_c (0)	Variance of age of Firms with the same cost
Min_age_c (0)	Minimum age of Firms with the same cost
Max_age_c (0)	Maximum age of Firms with the same cost
New_k_cost (0)	Sum of new capital of Firms with the same cost
Av_age_death (0)	Average age at death of Firms with the same cost
Var_age_death (0)	Variance of age at death of Firms with the same cost
Min_age_death (0)	Minimum age at death of Firms with the same cost
Max_age_death (0)	Maximum age at death of Firms with the same cost
Av_q_c (0)	Average quantity produced by Firms with the same cost
Var_q_c (0)	Variance of quantity produced by Firms with the same cost
Min_q_c (0)	Minimum quantity produced by Firms with the same cost

Max_q_c (0)	Maximum quantity produced by Firms with the same cost
Av_de_k (0)	Average modification in capital of Firms with the same cost
Var_de_k_c (0)	Variance of modification in capital of Firms with the same cost
Min_de_k_c (0)	Minimum modification in capital of Firms with the same cost
Max_de_k_c (0)	Maximum modification in capital of Firms with the same cost
De_k (0)	Total modification in capital of Firms with the same cost

The Object Ratio contains the sets of Firms with different capital/output ratios. It contains also some parameter used by probability functions. Note that since these parameters are placed here, Firms with the different capital/output ratios can use different probability functions parameters. Of course, it should be possible to move these parameters to the Object Market, allowing every Firm to access the same parameters.

Ratio

Id (p)	Identification parameter, used to distinguish different Ratios
A (p)	Value of ratio capital/production
b (p)	First coefficient used for probability functions (its use depends on which function has been selected)
c (p)	Second coefficient used for probability functions (its use depends on which function has been selected)
Q_ratio (0)	Sum of quantity produced by Firms with the same ratio
Num_firm_ratio (0)	Number of Firms with the same ratio
Av_age_ratio (0)	Average age of Firms with the same ratio

The Object Market has, besides the normal variables, a number of switches which control alternative possible investment functions, probability functions for the number of entering Firms, demand curve functions and others. They have been put here because since there is one single Object Market and so every Firm can access the same instance of the switches. Of course, this choice could be reversed by allowing Objects Cost and Ratio to have separate instances of these switches.

Market

depreciation (p)	Physical depreciation of capital, to be compensated with investments
v (p)	Cost of a unit of capital
lambda (p)	Fixed amount of investment per unit of capital.
eta (p)	Extra amount of investment per unit of capital.
delta (p)	Minimum amount of profits per unit of product to get extra amount of investment. See the equation for I.
ind_inv (p)	Switch to set different investment equations
epsilon (p)	Coefficient used to compute the treshold. See the equation for Treshold
ind_death (p)	Switch to set different equations to determine the exit of Firms.
Switch_demand (p)	Switch to set different demand functions.

Switch_draw (p)	Switch to set different equations to compute the initial capital for new Firms
ind_ent (p)	Switch to set different equations to compute the probability for new entrants. See the equation for gamma.
Dem_param1 (p)	First parameter for the demand function
Dem_param2 (p)	Second parameter for the demand function. Different use, depending on the type of demand function.
Q_tot (1)	Averall total quantity produced in the market. The lagged value is used to compute the Price.
Price (1)	Price. Its lagged value is used in one version of the gamma function. See the equation for gamma.
Num_firm_tot (0)	Overall total number of Firms
Av_age_tot (0)	Overall average age of Firms

The following Object is devoted uniquely to contain other Objects with different probability functions.

Prob_functions

(no elements)	

Some of the probability functions below need sets of elements to be determined. For example, the gamma function returns an integer between 0 and n, with probabilities given by a set of n+1 values. The user must be able to determine both n and the probabilities. Since Lsd does not use vectors, the only possibility is to use Objects. Hence, descending from the Object Gamma, there is a set of Object "pi" (whose number corresponds to n) each having one parameter used as probability. It is the user's responsability to ensure that the probabilities sum to 1.

Gamma

psi (p)	Parameter for the gamma function
gamma (0)	Draw from a gamma function whose elements are determined in Objects pi

pi

prob_pi (p)	Probability of drawing a determined integer. See the equation for the gamma function

Normal

standard_deviation (p)	Standard deviation for the draw from a normal function
normal (0)	Draw from a normal function.

Uniform

uniform (0)	Draw from a uniform. See the equation for uniform

Unifatoms

unifatoms (0)	Draw from a uniform atomic random function.

Di

prob_d (p)	Probability to draw “di” from the uniform atomic random function
di (p)	Value to be drawn from the uniform atom random function.

4.2 Equations

The following equations are contained in the file fun_dkw.cpp. See the comments on the equations for the Nelson and Winter model for an easier reading of the equations. Note that here is used the Lsd function `stat(...)`. It provides a set of descriptive statistics about a variable contained in a set of Objects. Hence, `stat(XXX, v)` places in the first five elements of vector v (v[0], v[1], ..., v[4]) the mean, min, max, variance and number of instances for the variable XXX contained in Objects descending from the calling Object.

M	Q_tot	Σ_i Q_ratio[0] The total quantity is computed as the sum over the ratio’s quantities.

M	Price	If Switch_demand=1:
		Price= max{Dem_param1+ Dem_param2*Q_tot[1], 0}
		
		If Switch_demand=2:
		Price= Dem_param1*exp(Dem_param2*Q_tot[1])
		
		If Switch_demand=3:
		Price= [Q_tot[0] +
		exp(-log(Dem_param1)/Dem_param2)]^(Dem_param2)
		
		The price is computed with a linear, an exponential or a power function depending on the value of the parameter Switch_demand. In all cases the parameter Dem_param1 is the intercept, and the Dem_param2 expresses the reactiveness.

R	Q_ratio	Σ_iQ_cost[0] The quantity at "ratio" level is computed as the sum over the quantities computed over its descendants costs.
C	Q_cost	Σ_iQ_firm[0] The quantity at "cost" level is computed as the sum over the quantities computed over its descendants firms.

C	New_firm	Number of entrants within a certain (A,m) combination, as given by the function Gamma below. Associated with this variable is the creation of New_firm new objects Firm (within cost, within ratio). For each of them: - m and A are given by the combination they are born in, - the tag New_born is set to 1. - K is set to 0 (the "size at entrance" will be drawn in the following period, when the new firm will be recognized by its tag New_born=1. See K below) - I, Q_firm and Age are initialized to 0 (they will then evolve regularly in the following period. See I and Q_firm below) NOTE: during the intra-period sequence, firm elimination (death) is performed jointly with new firm creation
F	I	Investment per unit of output: If ind_inv=0: I= (1/v)*lambda*max{Price[0]-m, 0} If ind_inv=1: I= (1/v)*(lambda+eta* [Price[0]-m-delta)>0])*max{price[0]-m, 0} (if ind_inv is set to 1, the propensity to invest changes from lambda to (lambda+eta) when the profit margin per unit of output is such that (Price[0]-m)>delta).

F	Threshold	Threshold for deleting (killing) firms: If ind_death=0: Treshold= epsilon*b If ind_death=1: Threshold= epsilon*min{A*Q_tot[0],b} (If inv_death is set to 0, the threshold is based on a fraction of the minimal size at entrance --in terms of capital--. If ind_death is set to 1, the death threshold is based on a fraction of the overall --system-- output, converted into capital units, still with a lower bound given by the minimal size at entrance). (Also, b can be A-specific)
F	K	The value of K is computed differently for incumbents and newborns (as designated by the tag). If New_born=0: if K[1] < Threshold: K[0]=0 otherwise: K[0]=K[1]*(1-depreciation+(I/A)) If New_born=1: (here K[0] is the size at entrance --in terms of capital--) if Switch_draw=1: K[0] drawn from a truncated normal if Switch_draw=2: K[0] drawn from a Uniform if Switch_draw=3: K[0] drawn from a Uniform with atoms
F	Age	The age is computed adding 1 to the previous value
F	Q_firm	The quantity produced by each firm is computed as K[0]/A

C	Av_age_cost	The average age of firms at "costs" level is the sum of the ages of firms divided by the number of firms. This computation also produces the other statistics on age at Cost level.
R	Av_age_ratio	The average age of firms at "ratios" level is computed as a weighted average of Av_age_cost using the Num_firm_cost as weights. This computation also produces the value of Num_firm_ratio.
M	Av_age_tot	The average age for all firms is computed as a weighted average of Av_age_ratios using the Num_firm_ratios as weights. This computation also produces the value of Num_firm_tot.
M	Num_firm_tot	Number of firms present in all the industry. It is computed during the computation of Av_age_tot.
R	Num_firm_ratio	Number of firms at "ratios" level. It is computed during the computation of Av_age_ratio.
C	Num_firm_cost	Number of firms at "costs" level. It is computed during the computation of Av_age_cost.
P	normal	This function returns a draw from a normal with - mean = (b+c)/2 - standard deviation = parameter standard_deviation truncated within the interval [b,b+c]. b is the inf of the support. (the objects P are "called" within (A,m) combinations. Hence one can indeed differentiate b, c according to A. See parameters explanation)
P	uniform	This function returns a draw from a uniform on the interval [b,b+c]. b is the inf of the support. (same comment)

P	unifatoms	This function returns a draw from a mixture of a uniform on [b,b+c] with a given number n of atoms within the interval. d_i in [b,b+c] indicate their locations and prob_d_i the masses attributed to them, i=1,...,n. Hence: case prob value 0 1- Σ_i prob_d_i draw from Un[b,b+c] i prob_d_i d_i

P	gamma	This function returns an (integer valued) draw from:
		(1 refers to 0 entrants)
		
		if ind_ent= 0:
		prob_p_1= prob_pi_1
		prob_p_i= prob_pi_i
		
		if ind_ent= 1:
		prob_p_1= prob_pi_1*exp(psi*max{Price[1]-m, 0})
		prob_p_i= prob_pi_i*[1/(1- prob_pi_1)]*
		[1- prob_pi_1*exp(psi*max{Price[1]-m, 0})]
		
		(if ind_ent is set to 1, the probabilities are corrected to take into account profitability. See parameters explanation)

Chapter 5 - Paillard Model

This model is due to S. Paillard, University of Paris 1. She has all the credit for the content of the model. Here the model is presented following the implemenation made by the author with Lsd and it is meant uniquely to show the potentiality of Lsd.

5.1 Model Contents and Initial Values

The Objects represented in the model are: Market, Firms, Capital, Techno and Cap_Prod. Each firm in the model is representative of a sector producing heterogeneous goods. Every firm uses the same capital type, produced by the Techno sector. Here the capital goods embody technological growth, and hence their potential producitivity is fixed for each vintage at the time is has been bought. The model focuses on the switch from an existing paradigm for capital to a new one: the two paradigms can be used by every firm in the same way (that is, to produce final goods) but they have different prices and offer to users (i.e. final good producers) different potential productivities. Up to a certain period, only the old capital type exist and the Techno sector offers only new improved versions of it (that is, the model is in a steady state with constant increases in capital productivity). Later, the new paradigm appears, and the Techno sector begins to offer both capital types, each one having its own technological path. The productivity increases are divided in vintages and generations. The new capital is not available at the same time to every firm, but they can access it at different times[19].

Each firm can invest resources to acquire new capital and hence expand its production capacity or replace old capital when considered obsolete. If the firm already have access to the new capital type, it must decide whether to buy it (more expensive and more productive) or the old one. The choice considers also the learning process necessary to adapt the production process, used to the old type of capital, to the new one. Spillover effects allow later adopters to exploit a publicly available knowledge built upon early adopters' experience. Replacement of old capital is allowed, affecting the average productivity not only through the modified composition of the production capacity, but also through a "dispersion" effect: the more different capital types are used at the same time, the less productive they will generally be because of coordination/compatibility problems in using many different types of capital.

The model is "closed", since the sum of wages paid by both final good producers and capital production sector consititutes the monetary demand. This demand is divided in fixed shares (assuming no substitubility nor complementarity between the final goods) and matched with the prices determined by each firm/sector with a mark-up rule. Since the amount of production is determined by firms using an estimation of the demand,

[19] For simplicity of implementation, the firms are identified by an integer (1, 2, etc.) and they can access the new capital type only if its generation is higher than the identificator. For example, the second firm cannot access the first generation of the new capital type, but only from the second onwards.

there is room for both unsold goods and unsatisfied demand. In any case, both unsold goods and unspent money are transferred to the next period.

The following tables contains the elements contained in every type of Object. As for the model previously discussed, the tables list parameters and variables. The first column contains the name of the element and a symbol: (p) stands for parameter and (i) (with i equal a positive integer) indicates a variable. The user must specify the initial values for every parameter and for every lagged value for the variables. The second column contains a brief description of the element and, when necessary, reference to the equation using the element.

The Object Capital contains the units of capital acquired at a certain time by a Firm. A new Object Capital is created at each time step for each firm. During the subsequent periods, the capital is used for production and the firm can reduce its amount in order to replace its production capacity with newer capital units.

Capital

Generation (p)	Generation of capital.
Vintage (p)	Vintage of capital
Paradigm (p)	Old (0) or new (1) type of capital
K (1)	Amount of capital
New_K (0)	Variation of capital
Productivity (0)	Maximum units of output per units of capital,
Act_produc (0)	Productivity corrected with the current capabilitites of the firm
Optimistic_produc (1)	Productivity corrected with expected capabilities
Q (0)	Output produced by the capital

The Object Firm estimates the demand in each period, makes investements modifying and expanding its production capacity (under financial constraints) and goes on the market to sell its product. In case it makes use, in some of its capitals, of the new paradigm, it has also a learning process which narrows the gap between potential and actual productivity of new type of capital.

Firm

Sector (p)	Identification for the firm
Market_Share (p)	Share of total demand
Expansion_inv (p)	Amount of investments dedicated to the expansion of production capacity (that is, not considering the replacement investments)
Mark_up (p)	Mark up over cost
Periods_mov_aver (p)	Number of periods for the moving average of demand,

	used to estimated the future demand
Pay_back (p)	Number of periods in which an investment must pay for itself. Used when comparing two types of capital
Replace_inv (p)	Amount of investments for replacements
Coeff_mov_aver (p)	Coefficient for the moving average.
Correction (1)	Correction used to compute the actual productivity
Production_capacity (1)	Sum of the total production capacity (i.e. over every capital types)
Profit (1)	Profits for the current period
Profit_left (1)	Cumulated profits from the past
Innovative (1)	Equals 1 for firms making use of new capital type and 0 for firms using the traditional capital type.
Produc_new (1)	Amount of output produced with new capital type
Cum_produc_new (1)	Cumulated output produced with new type of capital
Stock (1)	Cumulated unsold production
Demand_f (4)	Current actual demand. Note that the lag must be equal to the value of Coeff_mov_aver.
Exp_Demand (0)	Expected demand
Finance_poss (0)	Money available for production and investment
Money_production (0)	Money spent for production
Desired_Expansion (0)	Desired expantion, in units of outputs
Invest_Decision (0)	“Semaphore” variable. It does not have any actual function in the model, but its equation performs the choice of new capital to buy and decides how to distribute the production over the available capital types. Hence, any equation which needs to be computed after those operations, can “call” for the value of this variable in order to ensure the correct scheduling. See, e.g., the equation for K
Price (0)	Price
Learning (0)	Measure the capability in using the new type of capital
Inv_expenses (0)	Expenses for buying new capital (either to expand the production capacity or to replace obsolete capital).
Production (0)	Amount of output produced by the available capitals
Sold_f (0)	Amount of production sold
Labor (0)	Amount of labor used for production
Produc_old (0)	Amount of output produced with old capital type
Av_productivity_f (0)	Average productivity over the different capital types
Num_capital (0)	Number of different capital types
Max_gen (0)	Most recent capital generation for new capital type
Min_gen (0)	Oldest generation in use
Herfindal (0)	Measure of dispersion of capital types

The Object Market contains, besides the elements to ensure the trading mechanisms, also the information common to every firm, like the minimum skill in using new type of capital.

Market

Sigma_corr (p)	Coefficient used to compute the parameter Correction
Gamma (p)	Coefficient used in the equation for Wage
Sigma_skill (p)	Coefficient used in the equation for Public_knowledge
phi (p)	Coefficient used in the equation for learning
cst (p)	Coefficient used in the equation for learning
Optimism (p)	Coefficient used by Firms for estimating the future learning in using new capital type
Wage (1)	Wage
Public_knowledge (1)	Minimum skill in using the new type of paradigm
Aver_skill (1)	Average skill in using new type of paradigm
Unspent_demand_tot (1)	Money from demand not spent.
Av_productivity (2)	Average productivity over all firms
Demand_tot (0)	Monetary amount of demand (money available).
Sold_tot (0)	Monetary amount actual goods sold (money spent).
Investment_made (0)	Total investment made
Labor_tot (0)	Total units of labor employed

The Object Techno is the producer of capital. It has two distinct Objects for each type of capital (new and old paradigm), while its own elements basically translate the passage of time in vintage and generation numbers.

Techno

Gener_lenght (p)	Number of vintages before a generation jump
First_gener (p)	Initial period (i.e. vintage number) before first generation of new capital appears
Max_gener (p)	Last generation of new capital
Gener_k (p)	Current generation available
Vint_k (0)	Current vintage available

The Objects Cap_prod set the characteristics of the last vintage/generation capital, either for the new paradigm or for the old, namely the productivities and the prices. Moreover, it keeps track of the amount of labor used to provide the capital units bought by consumer goods producers in order to provide its share of the global monetary demand.

Cap_prod

Parad_k (p)	Type of the capital (1 for new and 0 for old)
Exp (p)	Coefficient used in the equation for Labor_k
Prod_Producing (p)	Productivity in producing this type of capital
Prod_increase (p)	Fixed increase in productivity for each vintage/generation
Cum_Prod_k (1)	Cumulated production of this capital type
Product_k (1)	Maximum productivity in using current capital
Price_k (0)	Price of current capital
Sold_k (0)	Amount of this capital sold at this period
Labor_k (0)	Amount of labor used in producing capital units

As usual, the Object Root contains the technical elements of the simulation, that is the number of steps desired and the seed for the random number generator.

Root

Last_iter (p)	Last simulation step
Seed (p)	Seed for the pseudo-random number generator
Time (0)	Indicates the time step of the simulation

5.2 Equations

The following table describes the equations of the model. It can be used in two ways: to understand the computations used in the model or to read and possibly modify the code for the equations. Each variable in the table has a formal description of the code implementing it (a sort of pseudo-code) and a verbal description of it. See the equations description in the Nelson and Winter model and the Programmers' Manual for a deeper explanation of the Lsd implementation of variables.

Note that there are many variables which cannot be computed individual, but they need to be computed at the same time. For example, the quantity of capital units a firm wants to scrap or to buy need to be decided centrally, even though the variables are located in different Objects, namely Objects Capital. For this purpose, it is heavily used the possibility (described also in the paragraph "Simultaneous Equations" above) to "write" new variables value all together during one single computation. As example, see the way variable K in Capital is computed. It asks for the most recent value of New_k and adds it to its own previous value. In turn, New_k does not have a proper equation. It simply calls the (meaningless) value of the variable Invest_decision and returns its own value. Its actual computation is made during the computation for Invest_decision, which, among its actions, stores the new value for New_k. This trick allows, in a rather simple and easy to understand way, to impose a precise scheduling over a number of virtually parallel computations.

Each line shows, before the variable label, the Object owing it. The symbols for the are the following:

- M = Market
- F = Firm
- Kf = Capital (for users)
- Kp = Cap_prod (capital for producer)
- T = Techno
- R = Root

As usual, the variables labels in the equations have attached an integer between square brackets, indicating the lagged value of the variable used.

Object	**Name**	**Function**
F	Av_productivity_f	[Production[0]/Labor[0]]*Correction[0] It is possible to compute this variable only if there has been some production, otherwise Labor[0] equals zero. In the other case, it is used the previous value.
F	Cum_produc_new	Cum_produc_new[1]+ Produc_new[0]
F	Demand_f	Demand_tot[0]*Market_share/Price[0]
F	Correction	$[\text{Herfindal}[0]/(\text{Max\_gen}[0]-\text{Min\_gen}[0]+1)]^{\text{Sigma\_cor}}$ It is computed from the Herfindal index and divided by the distance between two extreme generations.
F	Desired_Expansion	$\text{Exp\_Demand}[0] - \Sigma_i K_i[1] - \text{Stock}[1]$
F	Exp_demand	$\text{Demand\_f}[1]*\{1+ \text{Coeff\_mov\_aver} * \Sigma_{i=1}^{\text{Periods\_mov\_aver-1}}((\text{Demand\_f}[i]- \text{Demand\_f}[i-1])/ \text{Demand\_f}[i-1])\}$ $/(\text{Periods\_mov\_aver-1})$
F	Finance_poss	Profit[1]+Profit_left[1]

F	Herfindal	$\Sigma_{i=0}^{Max\_gen}[K_i[0]/\Sigma_j K[0]_j]^2$ It is the sum of the relative production capacity wich each generation of capital. Old capital type generations are considered all together as generation zero.

F	Innovative	if Cum_produc_new[0] >0 then 1
		else 0.
		
		It is one if the variable Cum_produc_new is greater than zero.

F	Inv_expenses	decided during the computation of Invest_decision
F	Invest_decision	- delete the capital with K[0]=0 - Choose which new capital to buy. It must use the old paradigm if the Gener_k[0] is strictly minor than the Sector. If it is not, than it depends on the comparison: $(Price\_k^{new}[0]- Price\_k^{old}[0]) < Pay\_back*Wage[0]*$ $\{1/Product\_k^{old}[0]- 1/[Learning[0]*Product\_k^{new}[0]*Optimism]\}$ - Add chosen Capital to the list - Organize existing Capital for decreasing levels of Productivity[0] (newest first) - Compute the money to spend for the production with the existing capital: For each Capital i consider the value min{K[1]*[Wage[0]/Productivity[0]], Finance_poss[0]} that is the minor between the cost of using each capital and the money available. The result is compared with: [Desired_expansion[0]+Production_capacity[1]]* Wage[0]/Productivity[0]

		that is the value of the desired production. Taking the minimum of the two insure that the firm does not produce more than the desired level. The result of the double comparison is the estimate of the money to be used for the production period with the existing Capital. The money are generally underestimated because the routine does not consider the correction due to the multiplicity capital types. Such effects will be instead used computing the actual production. - Expand production. Buy as many new capital units as possible up to the money left from production and the desired expansion (the Finance_poss[0] has been deducted the money to be spent for the production): max{0, min{Desired_expansion[0], Finance_poss[0]/[(Wage[0]/Act_produc^{chosen}[0])+Price_k^{chosen}[0]]}} Note that the cost of capital to buy is increased by the cost of production: buy to expand the production but only for the capital you can afford to produce. The superscript chosen stands for any of the two paradigm chosen in the beginning. The variable Expansion_inv[0] is determined. - Organize capitals for increasing values of Productivity[0], that is the oldest first. - Replace. Cyclically, for every capital of the firm, check the pay-back criterion: Price_K^{chosen}[0]<=Wage[0]*Pay_back*[1/Optimistic_produc[0]-1/Optimistic_produc^{chosen}[0]] If the criterion is true, then decide the value: min{K[0], Finance_poss[0]/ Price_k^{chosen}[0] } that is the minimum between the units of capital and the units of the chosen capital that the money left after producing and expanding can buy. At this stage are decided the values of

		New_k for each capital (but the chosen one). - The variable Replace_inv is determined. - The variable Sold_k for the chosen capital is updated. - The variable Inv_expanses is determined.
F	Labor	- After the variable Production $\Sigma_i Q_i[0]/Act\_produc_i[0]$
F	Learning	If Innovative[1]=0 then Public_knowledge[1] else max{Learning[1]+phi*{Produc_new[1]/[Cum_produc_new[1]+ cst]}*[Learning[1]*(1-Learning[1]], Public_knowledge[1]}. If the Firm is not innovative, than it just uses the previous period public knowledge. If it is innovative, than it compares the evolution of the its learning with the P.K. and choses the highest.
F	Max_gen	Most recent generation of capital available. Only new capital types are considered.
F	Min_gen	Oldest generation of capital still in use. Only new capital types are considered.
F	Money_production	After Invest_decision Finance_poss[0]-Inv_expenses[0];
F	Num_capital	Number of available capital types
F	Price	Mark_up*Wage[0]/Av_productivity_f[0]
F	Produc_new	After Production $\Sigma_i Q_i^{new}$ that is, the sum of the quantity produced with the new types of capital.

F	Produc_old	After Production $\Sigma_i Q_i^{old}$ that is, the sum of the quantity produced with the old type of capital
F	Production	After Invest_decision - Sort the Capital type for decreasing Act_produc: more productive first. - Decide the quantities to be produced with each capital as the minimum between the units of capital available and desired quantity to produce: max{0, min{K[0], Exp_demand[0]-Stock[1]}}
F	Production_Capacity	After the investments have been made, it is the sum of the capital units available. ΣK[0]
F	Profit	Sold_f[0]*Price[0] - Labor[0]*Wage[0]
F	Profit_left	Finance_poss[0]-Inv_expenses[0]
F	Sold_f	min(Demand_f[0], Production[0]+Stock[1])
F	Stock	Stock[1]+Production[0]-Sold_f[0]
Kf	Act_produc	If Paradigm==1 then Learning[0]*Productivity*Correction[1] else Productivity*Correction[1]
Kf	K	New_k[0]+K[1]
Kf	New_K	After Invest_decision New_k[0] is determined during the computation for Invest_decision
Kf	Optimistic_produc	If the capital is of the new type

		Learning[0]*Productivity*Optimism else Productivity
Kf	Q	Set in Production
Kp	Cum_Prod_k	Cum_Prod_k[1]+Sold_k[0]
Kp	Labor_k	Sold_k[0]/[Prod_Producing* Cum_Prod_k[1]^{Exp}];
Kp	Price_k	[Wage[0]/Prod_Producing[0]]/power(Cum_Produ_k[1], Exp);
Kp	Product_K	Product_k[1]*[1+Product_increase]
Kp	Sold_k	set during the Invest_decision of the firms.
M	Av_productivity	For all the firms average their productivity with the market shares: Σ_i Av_productivity_f$_i$[0]*Market_share
M	Aver_skill	Σ_iLearning[0]*Market_Share$_i$ The average skill is computed as the weighted average of the learning in each sector using the market shares as weights.
M	Demand_tot	Wage[0]*Labor_tot[0] + Unspent_demand_tot[1]
M	Investment_made	For all firm: Investment_decision
M	Labor_tot	Σ_iLabor$_i$[0] + Labor_k^{new}[0]+ Labor_k^{old}[0]
M	Public_knowledge	Public_knowledge[1]+Sigma_skill* [Aver_skill[0]- Public_knowledge[1]] The minimum knowledge in using new capital types is approaching the average skills at speed determined by the parameter Sigma_skill.
M	Sold_tot	Σ_iPrice$_i$[0]*Sold_f[0]
M	Unspent_demand_tot	Demand_tot[0]-Sold_tot[0]
M	Wage	Wage[1]*(1+Gamma*(Aver_Productivity[1]-Aver_Productivity[2])/ Aver_Productivity[2]);
S	Time	The general clock of the simulation
T	Vint_k	If(Vint_k[1]==Gener_lenght && t>First_gener &&

		Gener_k<=Last_gener) then 1; Gener_k=Gener_k+1 else Vint_k[1]+1 It means that the value of the new vintage is the next number after the old one unless a new generation appears. The latter case happens if: The time is after the value of the parameter First_gener and the number of generation reached is minor of Last_gener