

# Using LSD for Simulations in Social Sciences\*

Marco Valente  
Università dell'Aquila, Italy  
and  
DRUID, Aalborg, Denmark  
mv@business.auc.dk

## Abstract

This document provides a survey of the operations that are usually required to make the best use of the simulation technique in social sciences. It is sustained that these flavor of simulation modelling requires a much bigger control on the internal working of models as compared to simulations of natural systems, since it is not possible to rely on experimental verification of the models. The value of simulation results is given in these cases by the test of consistency of sparse intuitions; the capacity of the model to plausibly replicate qualitative aspects of the reality under study; the capacity of understand the causality chain of events in the simulated reality, in order to provide suggestions for the interpretations of complex real phenomena.

The consequences of this assumption are far-reaching, since they involve a much bigger efforts in the exposition of the model encoding and control of the data produced, as opposed to simulations of natural phenomena where, generally, few aggregate statistics are the only important element to be tested against observation. The technical problems that these requirements implies are worsened by the fact that most researchers in social sciences are poorly programmers, when they can program at all, given the poor mathematical and computer sciences training normally included in their curricula.

The solution to this problem is partly methodological and partly technical. On the methodological side, it is proposed to adopt a gradual approach to model building, starting from the implementation of very simple building blocks and increasing the complexity of models only after a complete testing of existing implementation. On the technical side it is presented a simulation language that permits to express a model in the most powerful language used on commercially existing computers (C++), but hiding all the technicalities behind user friendly interfaces. Laboratory for Simulation Development (Lsd) has been developed with the double intention: on the one hand, Lsd provides easy to use interfaces, allowing average computer users to fully exploit simulation modelling. On the other hand Lsd offers a flexible and powerful language to avoid limitations on the nature and dimensions of the models Lsd can implement.

---

\*This document is meant to be a guideline for a course on the use of simulation for students not trained in computer sciences. It draws on the experience gained by the author teaching such courses for the PhD training program ETIC, funded by the EU, and directed by Prof. P.Llerena and Prof. E.S.Andersen. Lsd has been developed by the author initially as part of a project in IIASA, Vienna, directed by Prof. G.Dosi, and perfected during the PhD in Aalborg, Denmark, supervised by Prof. E.S.Andersen. As any software project Lsd gained from the comments and suggestions of numerous users, namely the students of the cited courses and various PhD students who used Lsd for their studies . It also benefited from the comments of several simulation modellers, among which E.S.Andersen, N.Jonard, L.Marengo, T.Reichstein, G.Silverberg, M.Yildizoglu. Any error, imprecision, or bug is, of course, the author's fault. This work is part of the NORMEC project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Simulation Model elements</b>	<b>2</b>
2.1	Variables	3
2.2	Functions	3
2.3	Parameters	3
2.4	Objects	3
2.5	Data required for a simulation run	4
2.5.1	Initial values	4
2.5.2	Number of objects	4
2.5.3	Simulation settings	4
<b>3</b>	<b>Using simulation models</b>	<b>4</b>
3.1	Programming is difficult	4
3.2	Learning by coding	5
3.3	A discipline for using simulation models	5
3.3.1	Cycle I: Link results to initialization	5
3.3.2	Cycle II: Extend or correct the structure of the model	6
3.3.3	Cycle III: Model completed	6
<b>4</b>	<b>Using Laboratory for Simulation Development</b>	<b>7</b>
4.1	Why Lsd?	7
4.2	Lsd language components	8
4.2.1	Technical requirements	11
4.3	Lsd Model Manager - A first look	11
4.4	Implementing the first Lsd model program	13
4.4.1	LMM creates a new model project - <b>LMM: Browse models</b>	13
4.4.2	Introduction to Lsd equations - <b>LMM: Edit equation files</b>	16
4.4.3	Defining Lsd model elements - <b>Lsd: Model/Add a Descending Obj.</b>	17
4.4.4	Running Lsd simulations - <b>Lsd: Run/Run</b>	18
4.4.5	Results of Lsd simulation runs - <b>Lsd: Data/Analysis of Results</b>	19
4.4.6	Extending a Lsd model equations - <b>LMM: Edit equation files</b>	21
4.4.7	Initializing Lsd elements - <b>Lsd: Data/Init. Values</b>	22
4.4.8	Setting the number of objects - <b>Lsd: Data/Set Number of Obj.</b>	24
4.4.9	Simulation settings - <b>Lsd: Run/Sim. Settings</b>	26
4.4.10	Documenting the model - <b>Lsd: Model/Create Report</b>	27
4.4.11	Using the Model Report - <b>Lsd: Help/Model Report</b>	29
4.4.12	Using lagged variables - <b>LMM: using VL("Var",lag)</b>	31
4.4.13	Multi-layered object structure - <b>Lsd: Editing Model Structure</b>	33
4.4.14	Equations for multi-layered models - <b>LMM: using CYCLE(...){VS(...)}</b>	34
4.4.15	The environment of Lsd equations	36
4.4.16	Extending the model: quality and sales	37
4.4.17	Assessing the model's behaviour	39
4.4.18	Replacing a variable - implement <i>WAverageRW</i>	40
4.4.19	Dead-lock errors - Spotting and fixing temporal inconsistencies	41
4.4.20	Modelling Time: changing order of Lsd equations	44
4.4.21	Interpreting results	45
4.4.22	Lsd Debugger	46

<b>5</b>	<b>Lsd language for equations</b>	<b>48</b>
5.1	Macro language an raw C++/Lsd coding	49
5.2	C++ basic grammar for Lsd coding	49
5.2.1	Comments	49
5.2.2	Assignments, arithmetic operations and increments	50
5.2.3	if ... then ... else	50
5.2.4	Use of cycle for	52
5.3	Equation and Function	52
5.4	System variables available for equations' writing	53
5.5	Lsd object's	54
5.6	Lsd and C++ Commands for Lsd equations	56
5.6.1	V("X") or p->cal(X,0)	56
5.6.2	V_CHEAT("X", fake_caller) or p->cal(fake_caller,"X",0)	57
5.6.3	SUM("X") or p->sum("X",0)	58
5.6.4	STAT(X) or p->stat(X, v)	58
5.6.5	WHTAVE("X","W") or p->whg_av("W","X",0)	59
5.6.6	MAX("X") or p->overall_max("X",0)	60
5.6.7	Mathematical and probabilistic functions	60
5.6.8	WRITE("X",value) or p->write("X",value,0)	61
5.6.9	INCR("X",value) or p->increment("X",value)	61
5.6.10	MULT("X",value) or p->multiply("X",value)	62
5.6.11	SEARCH("X") or p->search("X")	62
5.6.12	SEARCH_CND("X",value) or p->search_var_cond("X",value,0)	62
5.6.13	RNDDRAW("X","Y") or p->draw_rnd("X", "Y", 0)	63
5.6.14	CYCLE(obj,"ObjLabel") or for(obj=p->search("ObjLabel");obj!=NULL;obj=go_brother(obj))	64
5.6.15	SORT("ObjLabel","VarOrParLabel",DIRECTION) or p->lsdqsrt("ObjLabel","VarOrParLabel",DIRECTION)	65
5.6.16	ADDOBJ("X") or p->add_an_object("X")	65
5.6.17	DELETE(obj) or obj->delete_obj()	66
5.6.18	PARAMETER or param=1;	67
5.6.19	INTERACT("TEXT",value)	67
5.6.20	close_sim() function	67
<b>A</b>	<b>Error Messages</b>	<b>68</b>
A.1	Configuration errors	68
A.1.1	/usr/bin/ld: cannot find -ltcl8.3	68
A.1.2	undefined reference to '_gxx_personality_v0'	68
A.1.3	Other undefined reference ... errors	69
A.2	Equations' programming errors	69
A.2.1	fun_XXX.cpp:99: parse error before ...	69
A.2.2	lsd_gnu.exe: Permission denied	69
A.2.3	fun_pippo.cpp:99: label 'end' used but not defined	70

”In social science, generally, the situation is ... [that] the behavior of the total system can be observed. The problem is to derive a set of component relations which will lead to a total system exhibiting the observed characteristics of behavior. The procedure is to construct a model which specifies the behavior of the components, and then to analyse the model to determine whether or not the behavior of the model corresponds with the observed behavior of the total system”.

Cohen, K. and Cyert, R., 1961, “Computer models in dynamic economics”, *Quarterly Journal of Economics*, 75-1, pg. 112-27.

## 1 Introduction

The use of computer simulations is increasing among scholars of social sciences, and, with it, is also increasing the scepticism with which the results so obtained are received in some parts of the purported audience. This document describes how to create, understand and use a simulation model. The goal is to provide a guideline to better implement models and convince others of the results produced.

For simulations in social sciences we assume that computers can be used for making numerical analysis in a different way than in physical sciences. In social sciences numerical adherence to experimental data is not always (if ever) possible, or even desirable. In fact, the few measures we can take depend so much on non-mechanical phenomena (i.e. human behaviour) that the mere replication of the series is utterly useless, unless you cannot provide a robust explanation. Put in other terms, in social sciences the “data” are so scarce, that they can be produced by too many underlying processes, so that they cannot be used to confute or confirm a theory.

If the experimental method cannot be used, what can we do? My suggestion is that we can proceed by means of “qualitative” analysis. It is not the adherence to some specific experimental evidence that we must chase, but the capacity to “understand” real events in the light of consistent and reliable causes. A “good” theory will suggest causal links that provide a good explanation of observed events. A “bad” theory will be inconsistent and unrealistic.

Probably, many people will question that the above proposal can be called “science” at all. However, it is a matter of fact that quite many people resort to use computer simulations in fields like economics, sociology and psychology. The goal of these exercises is “to see what happens if ...”. That is, to control if a set of hypothesis produces or not an expected result. And, either in case the expected result is produced or it is not produced, the most important contribution to the increment of knowledge consists in understanding why the model produced such data<sup>1</sup>.

This document discusses the technical problems of “understanding” what happens in simulated exercises. It is assumed that these technical problems cannot be solved by “hired guns”, professional programmers implementing the social scientist’s ideas. This is because the very process of implementing the model, observing its dynamics, and studying the result are crucial for the “understanding why” is the reason for the whole exercise.

Moreover, it is not true that the economist willing to implement a model needs to have a second degree in computer sciences. The amount of skills required to write 99% of the code involved in a simulation model is minimal, and concerns more the logic of the model

---

<sup>1</sup>It should not be dismissed the case of a model failing to produce an expected result. It is amazing how much we can learn from the analysis of a “wrong” model.

rather than the knowledge of programming languages. In the following it is presented a simulation language that has been developed by the author in order to eliminate completely the technical difficulty of using a programming language in a simulation model. But, at the same time, requires modellers to fully specify the “laws of motion” of the model, in order to avoid the magic box effect where possibly interesting results spring from inexplicable causes.

The next section describes the logical elements defining a simulation model, while the third describes the phases involved in its exploitation. The fourth section describes how Laboratory for Simulation Language (Lsd<sup>2</sup>) implements the steps described in the previous section. Finally, the fifth section illustrates in detail every element available to write Lsd equations.

## 2 Simulation Model elements

A simulation run consists in producing a sequence of values for some periods of time<sup>3</sup>. Technically, it therefore requires a program the contains variables updated several times according to specified algorithms. The set of variables in a model are usually grouped in “objects”. Although this is not strictly required for the actual working of the simulation program, the use of objects greatly facilitate the understanding of the model, and its use, by organizing the elements of the model in clear and easy to understand way.

- Objects
  - Variables
    - \* Equation code
    - \* Initial value
    - \* Documentation
  - Functions
    - \* Equation code
    - \* Initial value
    - \* Documentation
  - Parameters
    - \* Initial value
    - \* Documentation
  - Objects
    - \* Number of copies
    - \* Documentation
- Documentation
  - Description of model elements
  - Initialization used

---

<sup>2</sup>Lsd can be downloaded from [www.business.auc.dk/lsd](http://www.business.auc.dk/lsd). The distribution, both for Linux and Windows operative systems, includes several example models that can be used for exercises.

<sup>3</sup>Actually, there are two strands of simulation models: discrete time and continuous time. In the former case the model is represented by finite difference equations, while in the second it is defined by differential equations. Since computer works only in discrete time, also continuous time simulations must be implemented in discrete versions. For this reason, we will focus only on discrete time models.

– Results

- System code

## 2.1 Variables

The variables are the real core of a model. They can be thought simply as a label to which are associated a series of values for each of the time step of the simulation. The numbers are produced using an algorithm computed once for each time step, returning a value. Some variables may need to be initialized before the start of a simulation. In fact, in most cases models compute variables as elaboration of other values produced in previous time step of the simulation. At the very beginning of the simulation, during the first computation of the algorithms for the variables, there are no past values, and therefore the user needs to provide these values.

## 2.2 Functions

In some cases a simulation requires to produce numerical values that are independent from time, but are simply functions that produce values when requested. While a variable needs to have one, and only one, numerical value for each time step (and therefore its algorithm is computed only once at every time step), a function needs to return a fresh result from its algorithm any time it is requested.

## 2.3 Parameters

Parameters are numerical values that are not modified. Normally, parameters are initialized before the beginning of a simulation run, and do not change their values. However, parameters may be overwritten during the calculation of the algorithm for some variable. For example, a variable may be implemented to compute the average over some values. In the process, the algorithm can compute also the variance of the same values, and write this value in a parameter.

## 2.4 Objects

Objects are just containers of the numerical elements of the model. The great advantage of the objects is the possibility to use a nested structure where some objects contain, besides their own variables and parameters, other objects, forming a hierarchical structure on multiple levels.

This representation is permits to implement the reality modelled in a consistent way, where aggregate objects are formed by smaller, component elements, which in turn may be formed by other smaller elements. For example, a model may be composed by an object Market which contains many objects Firms, which, in turn, are formed by several departments.

The object structure of a model forms a much easier to understand way to represent most of the realities observed. Moreover, it is also a very handy way to manage the problem of determining the number of the variables required in a model. For example, the above mentioned structured allows users to easily increment the number of firms in the model, ensuring that each new firm includes the necessary departments.

## 2.5 Data required for a simulation run

### 2.5.1 Initial values

Before starting a simulation it is necessary to provide values for all the parameters of the model. Moreover, it is also necessary to provide the “past” values for those variables which are used with a lag in the equation of other variables.

### 2.5.2 Number of objects

For all object types present in the model it is necessary to specify how many copies must be included in the model. Note that if the object structure of the model contains many levels (e.g. markets containing firms containing departments etc.), the number of objects must be specified for each group, subgroup and so on.

### 2.5.3 Simulation settings

The most obvious setting is the number of time steps the simulation must execute. Another important setting consist in the “seed”. This value affects the simulation if random numbers are used. In fact, computers provide so called pseudo-random values. These are series of values that appear as if they were drawn from a random function<sup>4</sup>. The seed is a code such that the series obtained from the same seed are identical. This option permits to re-create identical (pseudo-)random events.

## 3 Using simulation models

### 3.1 Programming is difficult

Simulation models are meant to provide hindsight on some real world phenomenon by trying to replicate on the computer what is considered the most relevant aspects of some problematic reality. However, using a computer language is a hard task, and, as experience shows abundantly, chances are that the modeller gets so much involved in the technical challenges of programming that the theoretical question at the base of the whole exercise gets lost in the list of error messages. To avoid that technical problems dominate over scientific ones, a strict discipline is required. This discipline can be labelled as KISS: Keep It Simple, Stupid. That is, apply an extremely gradual approach. Implement only very few lines of code, whose scientific content may be small (or null), and add new code only when the existing one has been thoroughly tested. Always assume that any single line contains an error (at least). Fixing a bug in a known line is a matter of seconds. Finding out where a plethora of errors occur, not mentioning what kinds of errors, is all but impossible.

The KISS principle is even more necessary for simulation models because programming errors recognized by the computer as illegal code are only a small part of the problem. Much trickier is to find and fix theoretical errors. That is, perfectly legal code, executed by the computer as meant by the programmer, that simply does not express the modeller’s theoretical concepts. These are the most difficult errors to find and fix, since the simulation runs smoothly, but, for example, produces market shares summing up to more than one, or negative prices. To individuate these errors it is necessary to not only browse through the code, but also interpret the whole set of data produced in the simulation from the perspective of the abstract concepts implemented in the model.

---

<sup>4</sup>Computer languages provide several random functions, generally derived from the uniform [0,1] random function.

## 3.2 Learning by coding

The two paragraphs above may appear discouraging, and suggesting that working with simulation models is an activity that should be delegated to technical people with specific training and skills. In this way, one may be brought to think, the “theoretical scientist” provides the intuition and basic concepts, while the programmer returns data, if not directly proofs pro or against the initial intuitions. This is, after all, how mathematics is frequently used in social sciences, with the mathematician elaborating skillfully symbols interpreted by the economists with suitable metaphors (and barely understanding the passages delegated in appendix). However, this method cannot work with simulation models. In most cases simulation models in social sciences are not used to provide yes/no questions to be tested against experimental data. Rather, they aim to replicate some historical observation with the goal of understand which mechanism produced those observed results. There is no point in producing artificial phenomena “similar” to the real ones, if the model does not provide some further insight.

Indeed, most of modellers working with simulations admit that the very process of implementing the simulation program is an invaluable source of inspiration and understanding. Building a simulation model is, by definition, the construction of an abstract world where we define the general rules that must satisfy two conditions: i) implement an abstract concept (the modelled nature), and ii) be consistent with the programming logic. In very many cases, even before writing the first line for an algorithm, the simple “translation” of (what appear as) clear and simple scientific concepts in formal computing language shows other aspects that a simple verbal account had obscured. It is possible, for example, to “discover” that the elements of our imagined reality can be put to work together only under some limiting conditions. Or that some aspects, that initially seemed irrelevant, are instead of crucial importance.

## 3.3 A discipline for using simulation models

The KISS principle is, therefore, both a good practice to avoid being entangled in technical problem, and a scientific methodology aimed at exploiting the simulation beyond the simple number-crunching aspect. The Figure 1 reports the steps involved in the use of a simulation model.

The steps from (A) to (F) are rarely, if ever, followed linearly. Almost universally, the work of the modeller is concentrated in the three cycles indicated by the arrows. The numbering of the cycles refer to the time that modellers spend in it: cycle I takes most of the time, cycle II is performed only when cycle I is concluded and cycle III marks the beginning of a new project.

### 3.3.1 Cycle I: Link results to initialization

In this phase a model maintains constant its elements and the code implementing its dynamics, while the actual values used in the exercises are varied. The results produced can be understood only by means of the (usually complicated) interaction through time of all the elements of the model. The modeller has to study the graphical and statistical output of the model to fully understand not what happens in a simulation runs, but, most importantly, *why* the model has produced those results. This phase of using the model can necessitate many replications of simulation runs with different random values and tests with different initialization. In some cases, modellers can be content to assess the “robustness” of the results. That is, identical results are produced, at least with

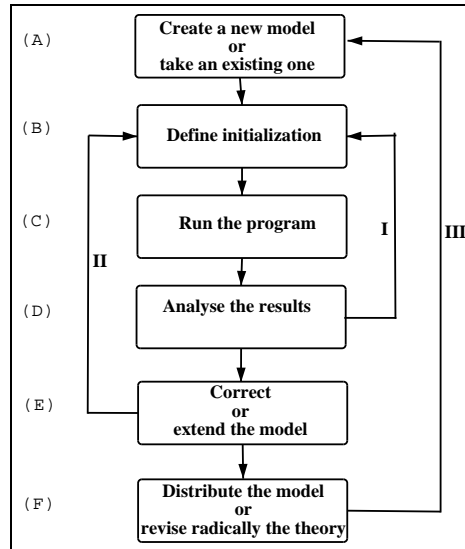


Figure 1: Development cycles of a simulation model (by E.S.Andersen)

statistical certainty, to support the theoretical claim of the model. However, in other cases, it is necessary to study the “outliers”. That is, to provide a full explanation of why something rare happened in the specific circumstances. In these latter cases it is necessary not only to rely on the study of aggregate series, but it is frequently necessary to control every single data at specific instants of the simulation.

### 3.3.2 Cycle II: Extend or correct the structure of the model

When the model produces fully understood results, there are two possibilities: the model produces interesting (either expected or unexpected) results, or the modeller found a technical or logical error in the implementation. Chances are that, most of times, one finds him/herself in the second situation. However, in both cases, the modeller is requested to work on the code of the model, altering its implementation. This phase produces a slightly revised version of the model, which needs to be re-tested via new numerical analysis of the results as a function of the initial data.

### 3.3.3 Cycle III: Model completed

When all bugs have been cleared and the model fully implements the theoretical intuitions at the base of its use, it is time to either spread the knowledge the model can provide (e.g. publish and distribute the code). It is also possible that the model proved wrong that initial intuition, or qualified it under unconsidered limiting assumptions. In any case, it is crucial that the model is packaged with usable interfaces and documentation so that other people (or the same modeller after some time), will be able to re-start a new project exploiting all the knowledge gained while developing the model. This will make easier to start a new project, either by the same modeller approaching a new theme or by other people willing to study the work already done.

## 4 Using Laboratory for Simulation Development

### 4.1 Why Lsd?

There are plenty of simulation languages, and much more programming languages, that can be used to implement practically every type of model. We could divide languages in respect of the type of audience they are mainly meant. Programmers' oriented languages are generally powerful, for example offering a wide set of libraries of frequently used functions, but are highly demanding on the skills required to use them. High level languages, on the contrary, are easy to use, but generally focus on specific types of models and are very rigid on the types of operations they allow. Lsd tries to break the trade-off being a low-level language (fast and flexible) surrounded by a layer of interfaces to perform easily the most common operations. A summary of the characteristics of Lsd is listed in the following:

- **Simplicity:** Lsd models are created effortlessly, with the modellers needing to focus exclusively on the model-related problems. LMM provides a development environment easy to use even for inexperienced programmers, for example suggesting the grammar of the most frequently used commands. Every function available to Lsd model program users is controlled with self-explanatory graphical interfaces and thoroughly documented by manual pages.
- **Transparency:** Lsd models contain only elements explicitly included by the modeller. Each element is automatically documented and is accessible via multiple interfaces.
- **Gradual modelling:** Lsd is designed to allow gradual development of models. Early versions of a model can be tested quickly and extended gradually avoiding "complexity traps".
- **Gentle power:** Lsd models run as compiled C++ code, possibly the fastest languages on commonly used systems, using dynamically allocated memory, exploiting at the best the computing power available. However, users need to write only the the computational content of the model and no other technical coding is required. Moreover, an extended set of functions and a macro language facilitates enormously the expression of the common operations.
- **Extensive running options:** any model can be run in many different modes, easily controlled by the user. It is possible to make a single run, as well as multiple runs for robustness tests. It is possible to produce run time plots to control a simulation on the fly, or have it run in memory only for maximum speed. It is possible to save any number of the series produced for post-simulation analysis. Simulations can be interrupted (on user requests or conditionally to certain events) to observe the status of the model and, if desired, to change current values.
- **Easy to distribute:** besides being multiplatform (any model is composed only by text files), the automatic documentation of a model ensures that on receiving a model any user can understand its working, using the model as suggested by the original author, and extend it as desired.
- **Modular:** Lsd model definitions keep completely separate model related code from system related one. This ensure, for example, that old models can be seamlessly transferred to newer versions of Lsd. Moreover, pieces of a model can very easily re-used in different models.

- **Easy to interface:** Lsd models' code can include any of the huge set of libraries available for GNU C++ compiler. Concerning data, Lsd models can be initialized with internal functions (e.g. create 1,000 random values) and provide a quite extended set of tools for statistical and graphical processing of the results. However, it is possible to include data from external files and produce output for high level analysis in external packages.

## 4.2 Lsd language components

Lsd models are stand-alone compiled C++ programs. That is, there are several source files, containing the code for Lsd shared by all models, and one file, different for each model, containing the code for the equations used to compute the values of variables. The whole code for a model is transformed in a program by a process called compilation, which also controls for possible grammar error in the processed code. The result of compilation (if everything goes well) is a running program containing the code for the equations of the model. This program, called Lsd model program, performs all the operations in using simulation models: create elements, initialize values, launch single or multiple simulation runs, inspect model status, analyse the results etc.

The process of creating a Lsd model therefore consists of two steps<sup>5</sup>:

1. write the code for the equations of the variables and produce a Lsd model program;
2. use the Lsd model program for every other operation (cycle I above)

The distribution of Lsd includes a program that facilitates enormously all the operations required to deal with the programming aspects. This program, called Lsd Model Manager (LMM), helps to select the model to work on (including creating a new one), to write the code for a model's equation, compile a program and interpret possible error messages, all of which by simple commands issued via standard scrolled menus. In the following we will describe the main steps in using the Lsd language (see fig. 2, pg. 10).

The main role of LMM is consists in assisting the coding of the **equations** of the model, although it performs also other functions related with the technical aspects of models (like selecting the model to work on, launching the compiler, setting compilation options, etc.). The equations are stored in a normal source file, containing only the code for the variables of the model.

The equation file is **compiled** by LMM, together with the rest of source files containing the Lsd system code. The result of this step may consist in errors. In this case, the errors are **grammar errors**, due to illegal code in the equation file, and the modeller is given information (like line number in the equation file) useful to determine the cause of the error and to fix it. When the compilation succeeds the resulting **Lsd model program** is automatically launched.

The Lsd model program contains the equations hard-implemented in C++. That is, the Lsd model program is endowed with the capacity to compute values of the set of variables contained in the equation file. But to run a simulation exercise it is necessary that the but the modeller provides the **configuration of the model**. This is the structure of the model (i.e. objects), variables and parameters, plus the numerical values required to kick off a simulation. This information is stored in text files, generated, loaded and modified by the Lsd model program. Topically, studying a model (or presenting its results)

---

<sup>5</sup>Which compose cycle II mentioned above

involves the creation of several configuration files, each sharing the same structure, but different initializations.

Once a model configuration is inserted in the Lsd model program, a **simulation run** can be executed. This consists in a number of time steps, during which all the variables of the models are updated with a freshly computed value. The system takes care of executing the code for the equations in the correct order.

It is possible that the simulation aborts for errors. These errors are **logical errors**, concerning, for example, the lack of a parameter used in an equation from definition in the model configuration. Or, more seriously, a temporal inconsistency requiring two values to be computed at the same time step using each other value. In these cases the program presents the data produced so far, allow the inspection of the model status at the time of the error, and issues messages with the information required to fix the error.

In case of success, the simulation produces **series of data** (for the variables that the modeller marked to be saved). The saved data can be treated with the Analysis of Result module, providing a fairly complete set of graphical representation of the series and descriptive statistics. It is possible to export the picture of the graphs, or to export directly the data series for analysis with specialised packages (e.g. SAS, SPSS, etc.).

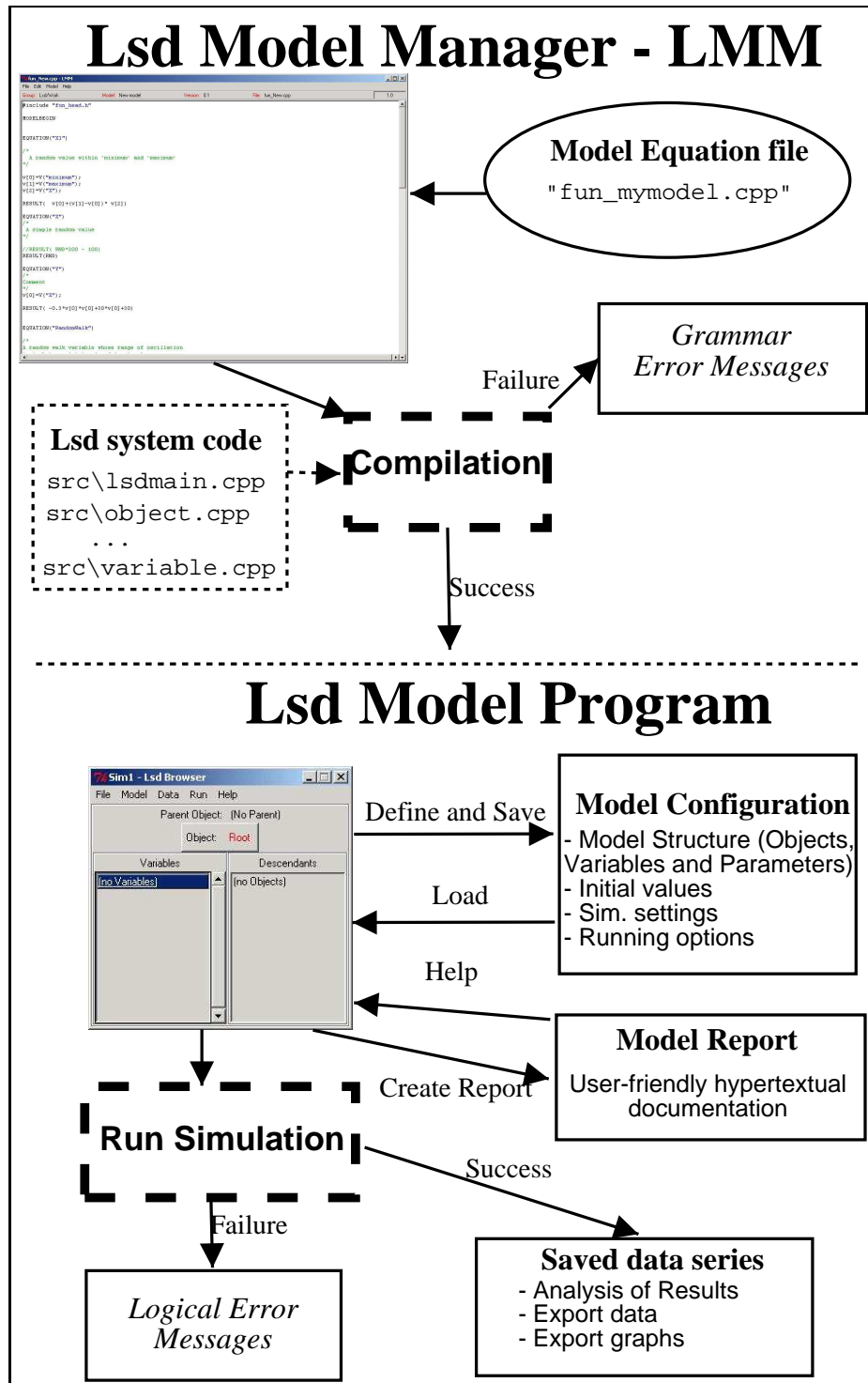


Figure 2: Steps of the Lsd language for simulations

### 4.2.1 Technical requirements

Lsd is designed to run smoothly on both Unix and Windows platforms without requiring other software than those distributed, or otherwise commonly found on the standard systems<sup>6</sup> In the following are listed the packages exploited by Lsd. This paragraph is not strictly required for using Lsd. Uninterested readers should be satisfied knowing that all these packages are, like Lsd, open source software available for free, although specific legal terms may differ.

Lsd model programs are compiled with GNU C++ compiler, requiring the standard libraries and the Tcl/Tk windowing language. Moreover, if available, LMM provides access to GDB for debugging models at source level<sup>7</sup>. The help pages are shown using the HTML browser available on the system, while the analysis of results needs GNUPLOT to create some types of scanner-plot graphs.

The Linux distribution includes only the code for Lsd and LMM, which must be compiled with a distributed batch file. Therefore, Linux users must ensure to have installed the compiler and Tcl/Tk. Although not strictly required, it is strongly suggested to have installed Netscape, GDB and GNUPLOT.

The Windows distribution includes all the software required. The CYGWIN (<http://cygwin.com>) distribution has provided the compiler and all the software necessary for compilation, including the standard libraries, and GDB. Tcl/Tk is partly taken from its own Windows distribution and partly (the static libraries) has been compiled on purpose for Lsd. WGNUPLOT is a port of GNUPLOT under Windows.

Lsd, LMM are copyright by Marco Valente, and are distributed under the GNU GPL (that is, you can use and distribute it for free), like most of the software required and/or distributed by Lsd. See the licenses for each specific software for further details.

## 4.3 Lsd Model Manager - A first look

The first step in using Lsd models is to run the Lsd Model Manager. LMM is basically a text editor, with added a set of commands used to manage Lsd model projects. When LMM starts you are offered three choices: operate on Lsd models, open a text file, or create a text file. Choosing to browse Lsd models a new window shows the set of models available (see figure 3)

This browser shows the models available and permits to create the structure of new ones. Models are contained in single directories, which can be located in "groups" containing related models. The Model Browser allows to navigate through the installed models: clicking on the label of a group the browser shows the content of that group. Clicking on the label for a model, that model is selected.

The browser' menu Edit permits to create models or groups, to copy models from one group to another (or to the same group with a different name), and to delete models.

The distribution includes two major groups: the "Example Models" group, containing several models of different types, and a "Work" directory. Exploring the distributed models you can read a brief description of the models. If you select one of the model, LMM quits the browser and is ready to work with that model. In case you want to use another model, in the menu **Model/Browse Models** in the LMM menu bar, you can always access again the models' browser. On LMM instance can work with only one model per time, although, if

---

<sup>6</sup>Unix users are expected to have a standard Linux box with installed the C++ compiler and its standard libraries. The MS Windows distribution includes all the software required.

<sup>7</sup>GDB permits to observe the running of a program line-by-line. Lsd model programs include a debugging function that gives access to a simulation equation-by-equation.

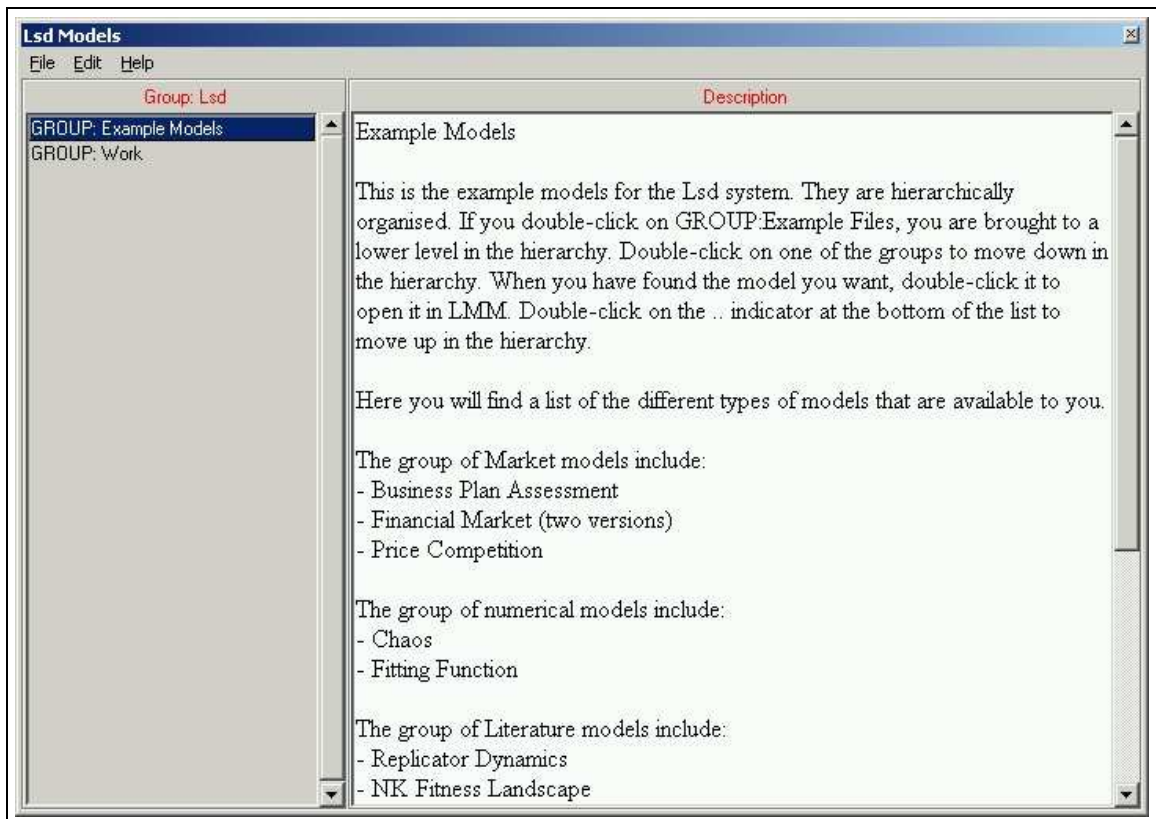


Figure 3: Lsd Model Manager - Model Browser

you really want to mess up your screen with dozens of windows, it is always possible to run multiple copies of LMM.

The LMM appearance is that of a standard editor, but for the menu “Model” and for a header below the menu bar (see figure 4).

The header shows the group containing the model, model name and its version number<sup>8</sup>, and the file currently loaded in LMM. The last element of the header shows the line and column position of the cursor in the editor.

The menu **File** deals with the text files loaded into the editor. Menu **Edit**, besides the usual commands, contains several functions particularly useful when writing C++ and Lsd code. Menu **Model** allows users to issue any command related to Lsd model programs. Finally, menu **Help** open few starting pages of the LMM manual, which is written as a set of standard HTML pages connected with hyperlinks. Notice that most of the entries in the menus are endowed with shortcuts, so that it is possible (and much faster) to activate the corresponding command using the keyboard. Moreover, the secondary button of the mouse (usually the right one) clicked on the LMM editing windows opens a short menu for frequently used commands. To get the documentation concerning a specific command open the LMM manual page and follow the link to the command.

<sup>8</sup>The version number of a model is used only to distinguish models. Two models with the same name and different ver. number are completely independent, although presumably the one with the higher version has been developed after the other one.

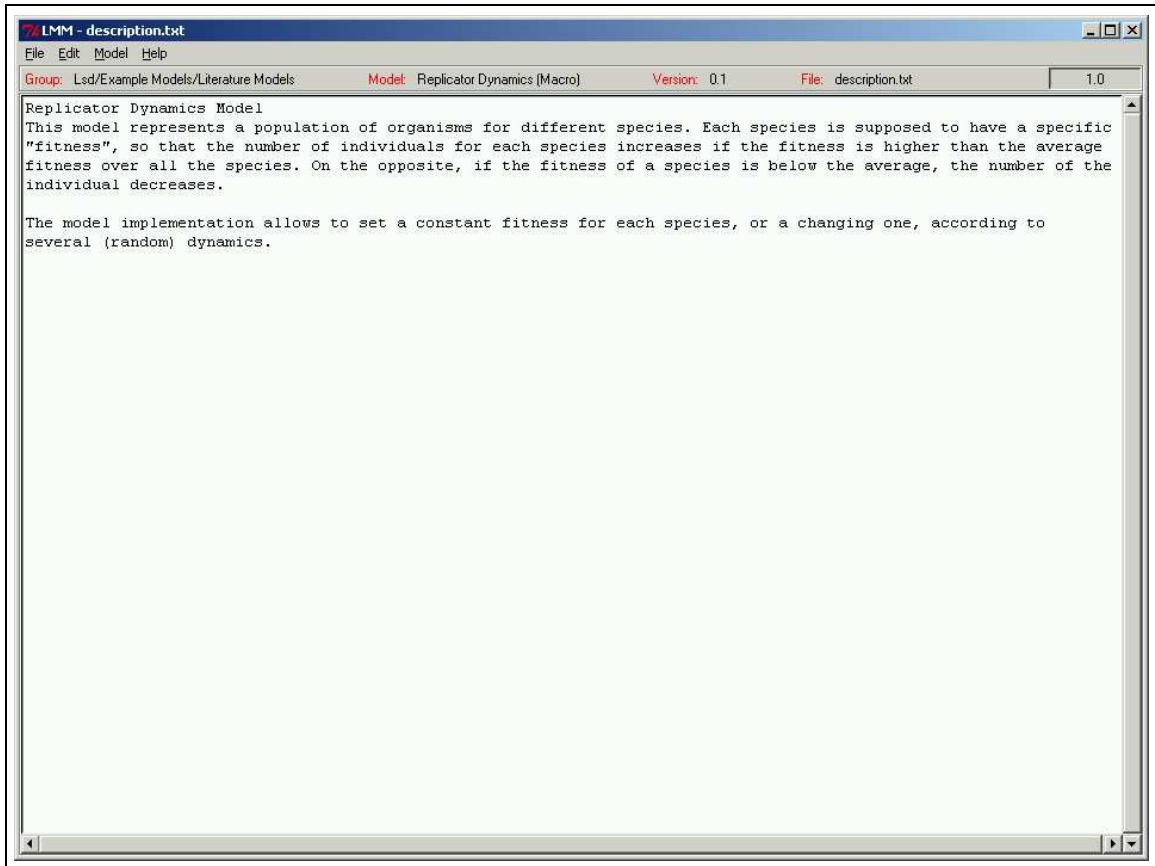


Figure 4: Lsd Model Manager

#### 4.4 Implementing the first Lsd model program

In this paragraph we implement a (quite stupid) model in order to highlight the elements composing a model and the procedures necessary to implement them.

The first thing in order to develop a model is to have a clear idea of what we want to implement<sup>9</sup>. For our present purposes we choose an extremely simple model, so to concentrate only on the LMM and Lsd operations necessary to implement it. It is also a generally good idea to start implementing a very simplified version for any model. Being able to fix errors while their code have just be written is a hugely successful technique, saving ages of time in respect of writing 1000 lines of code filled with inextricable garbage.

Let's say we are interested in implementing a model with one single variable, say  $X$ , computed as a random value.

##### 4.4.1 LMM creates a new model project - LMM: Browse models

The first operation we need is to create a model project. In Lsd a model is stored in a dedicated directory and must contain several files. In the paragraph we will use LMM to create all the files required and we will test that Lsd have been installed properly by running a copy of the Lsd model program, although, since no equations have been written yet, it will be not able to do much.

<sup>9</sup>Although, as mentioned in section 3.2 the original idea is very likely to be modified because of the very process of implementing it.



Figure 5: Create a new model project

1. Run LMM and choose option **Browse Models**. In case you have already opened LMM, choose menu item **Model/Browse Models**. The Lsd distributed package are included several example models, placed in the group *Example Models*. The directory structure of the distribution creates also an empty group where to locate models under development.
2. Enter in the group *Work* using the arrows keys or double-clicking on *Work* with the mouse.
3. Choose menu item **Edit/New model/group**. Choose the option to create a new model only. A new window will ask you the names to be used for the new model (see figure 5). You can accept all default options pressing the button **Ok**.

As general philosophy, all LMM and Lsd commands try to prevent possible errors performed by users. For example, if you try to create a model with the same directory name and/or the same name and version number of another model in the same group LMM will issue a warning.

The operation you have performed has created all the elements required for an “empty” Lsd model program, that is a model without equations, variables, objects etc. These elements, located in the dedicated directory you specified, are:

- An equation file. A C++ source file where to place the code for the equations of the model.
- A **makefile**. This is a script file containing the commands to be given to the compiler in order to produce an executable Lsd model program out of the Lsd system code and of the model specific equation file<sup>10</sup>. The actual **makefile** used by LMM is always re-created merging these two portions, so that when you send your model to another Lsd user, you will include only the model-related part of the **makefile**, and the receiver will continue to use his system dependent portion.

<sup>10</sup>TECHNICAL NOTE: The **makefile** is actually formed by a portion depending on the system you are working on (e.g. whether it is a Linux or Windows machine) and another part containing your compilation options (e.g. the name of the equation file, and whether to optimize the code for speed or include debugging information in the executable)

- Info on the model, that is the label and version number used to identify the model.

At the end of this procedure the the LMM editor window shows the equation file for your new model. Although this is pure C++, Lsd tries to hide all the technical parts of coding. So, the user is shown a file that, besides the initial line, is done by two subsequent commands: `MODELBEGIN` and `MODELEND`. These lines identify the beginning and ending positions where you can place the equations for your model. The `close_sim()` is used only for advanced operations, and can generally be ignored<sup>11</sup>.

The file containing the equations is all that it is technically required to create and run a Lsd model program, although, of course, without equations the Lsd model program will not be able to do much. However, just to test if everything goes ok, you can compile and launch the Lsd model program of your model. Do this with the command **Model/Run**. If the compilation process succeeds<sup>12</sup> you will be shown two windows. This two windows are the interfaces for a Lsd model program. One is a log window, used to communicate messages from the Lsd model program and the user. This window is mainly used during a simulation run, for example to interrupt a simulation. The other is the Lsd Browser, used to issue commands to the Lsd model program, is shown in figure 6. This window is used to load a configuration and modify it, besides launching the actual simulation run.

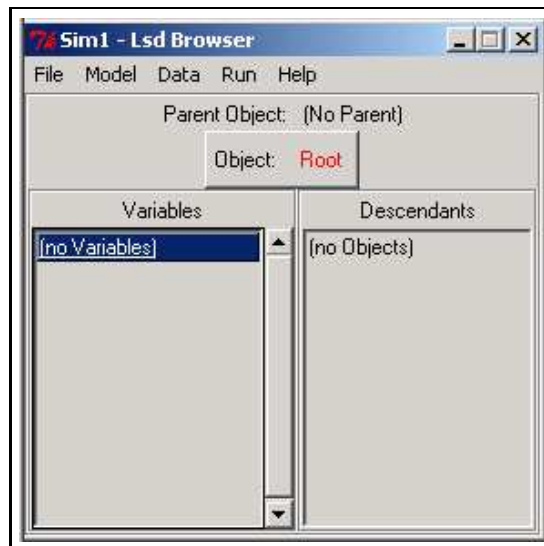


Figure 6: Lsd Browser

Before continuing you need to kill the Lsd model program (menu **File/Quit**). In fact, we are going to edit the model equations' file, and modifications to the equations' code

<sup>11</sup>TECHNICAL NOTE: Of course, these commands, as many others used in Lsd equations' coding, are not true C++, but are part of a Lsd macro language. Lsd macro language and C++ can coexist in the same equation file. For example, the `MODELBEGIN` macro declares a function (a method of a C++ class) and initializes all its local variables. If necessary, users may add new local and/or global variables to the file. `close_sim()` allows to perform post-simulation cleaning, like removing memory explicitly allocated during the simulation by the code written by the modeller. Of course, all memory used by the system is automatically dealt with.

<sup>12</sup>Compilation failures can be due to errors in the equations' code or to wrong information concerning your system. If the compilation fails, a new window reports the output of the aborted compilation process, containing messages indicating what errors have been found. Given that you have no equations' code at all, in case of error the only explanation is a wrong information about your system. Use the menu entry **Model/System Compilation Options** and set the default for your operative system in order to re-construct the basic information required.

cannot be embodied in a running Lsd model program: LMM must re-compile a new Lsd model program any time a fresh equations' file is produced.

#### 4.4.2 Introduction to Lsd equations - LMM: Edit equation files

Users define in the equations of the model all the operations that need to be executed by the model. Equations are associated to variables of the model. They are pieces of code that perform the specified C++ code and return a value which is assumed for each time step by the variable to which the equation is associated.

In practice, the modeller writes an equation by placing the lines of code implementing the equation in a block of code. The block of code for an equation begins with a line signalling the name of the equation and terminates with the result to be assigned to the variable.

In order to write the equation, let's return to LMM. Your editor should be still showing the empty equation file. If it does not, use the menu **Model/Show Equations** to have LMM re-open the file. It is very important that you do never try to open the file for the equations using the menu **File/Open File**. In fact, although this is not formally incorrect, there is the possibility that you edit the wrong file. In this case, the equations' editing is not included in the Lsd model program.

Place the cursor in any point after the line `MODELBEGIN` and before `MODELEND`. It is now time to discover some of the utilities that make LMM very useful to write Lsd model programs. Although you could write the text yourself, LMM gives you the possibility to include the most frequent lines of Lsd and C++ coding through forms that ask you the basic information, and then insert in the equation file the text required. These forms prevent the most frequent kind of coding errors, that is, the trivial typing mistakes. Now, we want to introduce a new equation. Open the menu item **Edit/Insert Lsd Script**. You are shown a window with a series of available scripts. Select the first item (*Equation/Function*), and click on **Ok**. There will appear a window asking the name of the variable for which you are writing the equation, and the option on whether it should be an equation or a function. Keep the default choice of equation<sup>13</sup>, insert "X" as label of the equation and press **Ok**. Beginning in the position where you located the cursor before activating the script, you will find the following text:

```
EQUATION("X")
/* Comment */

RESULT( )
```

As you see, the line `EQUATION("X")` marks the beginning of the equation. The line immediately after indicates where you could (and should) include a comment describing what the equation is supposed to do. The last line, `RESULT( )`, indicates the end of the equation and contains the numerical value returned by the equation to be assigned to the variable.

---

<sup>13</sup>Both functions and equations are associated to a variable and return a numerical value. Variables are computed once, and only once, at each time step. If other equations ask the value of a variable several times in the same time step, the equation of the variable is computed only once, and the subsequent times the same value is returned. Instead functions are computed only, and every time, they are requested by other equations, that is never or several times in each time step, possibly producing different values. If you are in doubt whether variable `MyVar` needs to use an equation or a function, as yourself: Does `MyVar` make sense tagged with time, as `MyVart`? If the answer is yes, then it must be an equation (99% of times), otherwise it is a function (very rarely needed).

Let's write the most simple equation possible. In between the parenthesis of the `RESULT( )` place the command `RND`. This is a Lsd commands that produces a random value drawn from a uniform random function between `[0,1]`. Therefore, the complete equation's code is:

```
EQUATION("X")
/*
  A simple random value
*/

RESULT(RND)
```

Save the equation file (menu **File/Save**) and re-run the Lsd model program with menu **Model/Run**.

#### 4.4.3 Defining Lsd model elements - Lsd: Model/Add a Descending Obj.

If the Lsd model program windows do not appear, and you have an error message instead, this means that you managed to put an error in your equation's code. Tell LMM to not run the old Lsd model program file (it is the one without the equation) and read the newly appeared *Compilation Results* window for indications on the probably line number where the error is located. See also the appendix [A.2](#) (pg. 69) for help on fixing the error.

Lsd model programs are all externally identical, in that the equations they embody do not have any visible effect on their interfaces. However, our present copy of Lsd model program is able to compute a value for a variable `X`. But we need to tell Lsd to define Lsd variables (and parameters and objects) that make use of those equations<sup>14</sup>. The Lsd Browser provides the interfaces for creating the elements of the model, but, before doing this, let's give a look at the browser (see fig. 6).

The Lsd Browser is composed by a menu bar, through which the user issues the commands, and a section describing the content of one object. Any model must contain at least one object, called *Root*, and this object is shown when a Lsd model program starts. The description of an object is composed the label of the "parent" object (that is, the object containing the object shown), the label of the object, and two lists, one for the variables and parameters (indicated with the general title *Variables*) and the other for the contained, or descending, objects (labelled *Descendants*).

Normally, the *Root* object should not contain any variable or parameter, but should serve only as container for the objects implementing the model. Therefore, let's start by creating an object descending from *Root*. Choose menu **Model/Add a Descending Obj.**. In the resulting window enter the label for the object, say *MyObj*. Now the Lsd Browser will show the *Root* object containing the *MyObj* object. Moreover, a new graphical window appears showing the object structure of the model below (i.e. contained into) the object *Root*. For the moment, it shows only one object.

Move the Browser to show the content of *MyObj*. To move the browser you have several possibilities. You can use the mouse by double-clicking the list of descendants on the label of the object you want to move to; you can double-click the graphical representation of the model on the object you want to see; you can just use the arrow keys to highlight the object you want to see and press enter when you have done. Notice that when the Browser shows *MyObj* the parent label shows that it descends from *Root*. You can double-click on this label to "move up" the browser showing the *Root* again (or press the letter 'u').

---

<sup>14</sup>Of course, it is possible to write code for variables that are not used in a simulation.

Eventually, we managed to have the Browser showing the content of a just created object, which is, obviously, empty<sup>15</sup>. Let's add a variable to this object. Choose menu item **Model/Add a Variable**. In the resulting window type the name of the variable for which we have an equation,  $X$ , and press **Ok** (ignore the field 'Maximum lags used' leaving the default value of 0). Now the Browser shows that *MyObj* contains a variable, called  $X$ . The list of variables shows **X (0)**; this means that  $X$  is a variable (as any label followed by integer numbers). Later will see that parameters are attached the letter **(P)**.

The definition of a model structure (that is variable, parameters and object) is stored in memory only. Before continuing, in order to be able to reload the model structure as we have defined it until now, save it with menu **File/Save**. By default the configuration is assigned the name of **Sim1**, although, of course, we could have used a different name.

#### 4.4.4 Running Lsd simulations - Lsd: Run/Run

Simulation models requires very many information to be provided in order to be executed. Lsd has requested to specify the very basic ones in order to run a model, and others, not influencing the results but only the way these are presented, have been automatically set by default. In this paragraph we start exploring all these factors that influence the capacity to exploit a simulation model.

If you have executed all the steps described above, you can now just run a simulation by choosing menu **Run/Run**. Before starting, Lsd reminds you what it is going to do, namely running a single simulation run keeping the results in memory, and writing the configuration currently in memory on a file, so that any file with that name will be overwritten. Pressing **Ok** will start the simulation.

The Log window shows one line for each time step successfully completed (you will see this only when the simulation finishes after few hundredths of seconds). At the end of the simulation the Log window reports the total time of the simulation and a finishing message, and the Browser reappears. Besides the lines in the Log window, there is no other difference with the Browser before the simulation run.

In fact, Lsd has done everything we have said it to do: compute the values of  $X$  as a random value. But we did not tell Lsd to save or show the results in any way, so we have lost (almost) all of them. Actually, one single datum is still available. When Lsd terminates a simulation run the Browser keeps the status of the simulation at the very last time step. The only way to obtain our results is therefore to repeat the simulation, this time using the options to save the results. We need to reset the Lsd model program telling to start again a simulation run. This is done loading the file containing the model structure we have previously defined: open menu item **File/Load** and choose the (probably only) file with extension `.lsd`.

Now we are ready to run again the simulation, but before doing this we will ask the Lsd model program to keep the values produced. Move the Browser to show the object *MyObj* and double-click on the variable  $X$  (or use the arrow keys to highlight and press enter). Now the Browser is transformed as shown in fig. 7.

This window allows several operations concerning the variable in question. For the time being we focus on the three checkboxes after the header with the name of the variable: **Debug**, **Save** and **Run Time Plot**. Check on all the three of them and click on **Continue** to return to the Browser.

---

<sup>15</sup>Beware of being in the right object. If, by mistake, you add elements to the wrong objects you will need to remove them and re-create the elements in the correct object.

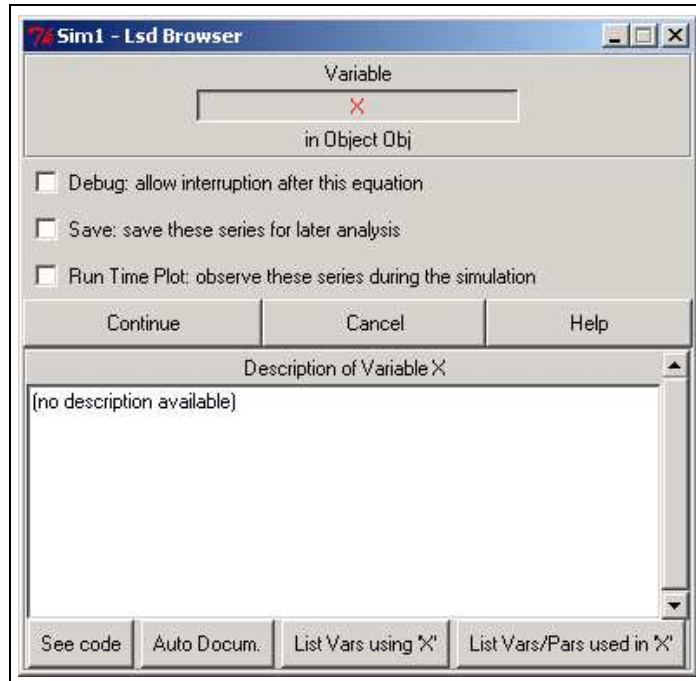


Figure 7: Lsd variable options

#### 4.4.5 Results of Lsd simulation runs - Lsd: Data/Analysis of Results

With the options we have set we tell Lsd that the values of Lsd must be saved for post-simulation analysis (**Save**) and that, during a simulation, we want to see the dynamics of  $X$  in a run time plotting window (**Run Time Plot**)<sup>16</sup>. Notice that the two options are independent, so that we may save the results of a variable even without plotting its values at run time, or, viceversa, observing its values without saving them for post-simulation analysis.

Now we can run again a simulation run (menu **Run/Run**). This time, the Log window does not show the steps completed, because a new window provides the graph reporting the values of  $X$  through the time steps. This way to observe the results provides an overall understanding of the dynamics of the variables concerned, although does not provide a detailed information on the data<sup>17</sup>.

A more precise presentation of the data produced during a simulation run is produced with the Analysis of Results module. From the Browser, choose menu **Data/Analysis Results**. A window will ask whether you want to analyse data stored in a file, or from the latest simulation run<sup>18</sup>. We want to use the data produced during a simulation (and, besides that, we don't have any file with saved data), therefore choose **Simulation**. The Lsd windows appear as in fig. 8.

This window allows several ways to present and elaborate results. The main body of the window is composed by three lists: **Series Available**, **Series Selected** and **Graphs**.

<sup>16</sup>The **Debug** option allows to control the internal computation of  $X$  during a simulation run. We will see that later.

<sup>17</sup>For reasons of computational efficiency the run time plot windows are not very precise in determining the extreme values produced, although it respects the proportions of the values plotted.

<sup>18</sup>If no simulation run has been executed Lsd asks directly for a data file. This would mean that you need first to run a simulation. Therefore, press **Escape**, run the simulation with menu **Run/Run** and open again menu **Data/Analysis Results**.

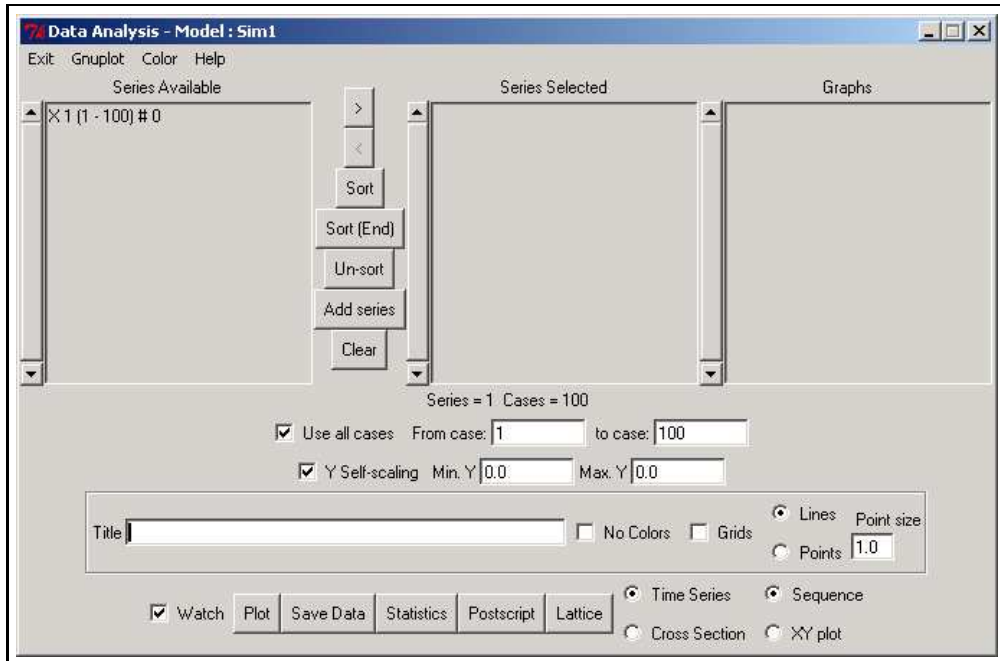


Figure 8: Analysis of Results window

Selecting the series in the first list you move them in the second list pressing  $>$  (or double-clicking on the series). The checkboxes in the lower right part determines how the data must be treated. The default setting are **Time Series** and **Sequence** asking for the temporal sequence of the data selected. Move the only series in the leftmost list ( $X$ ) to the the middle list. Leave the default options as indicated. Now you can press the button **Plot**; this will create an equivalent of the Run Time Plot window, in that the graph shows the time steps on the horizontal axis and the value of  $X$  on the vertical one. This time the graph provides several more information on the data. For example, hovering the mouse pointer over the window will give the coordinates of the point.

Besides plotting graphs the Analysis of Results module provides also some descriptive statistics. If the Analysis of Result window is hidden below the graph window, double-click anywhere on the graph. This will bring the Analysis of Result window in the foreground (this comes handy after you have produced some dozens of graphs). Press the button **Statistics** and observe the Log window. You can see the average, maximum, minimum values and other information concerning the series selected.

The Analysis of Result module in Lsd provides the most commonly used information concerning the results of the data produced in a simulation. Moreover, it can save the data in files ready to be imported in other packages for more sophisticated analysis. The possibilities offered by the Analysis of Results module are fully described in its help page (**Help/Help on Analysis of Results**). However, many options have no sense for the moment, since they involve the use of several series and we have only one, so we'd better exit from this module and explore other aspects of Lsd. Choose menu **Exit/Exit** to quit the Analysis of Result and return to the Browser. We are going now to update the equation file, inserting a slightly more interesting equation. Therefore, quit the Lsd model program, otherwise the editing to the equation file cannot be embodied in the Lsd model program.

#### 4.4.6 Extending a Lsd model equations - LMM: Edit equation files

The equation we have written is pretty basic. We have defined  $X$  as a variable returning random values in the range  $[0,1]$ <sup>19</sup>. Let's review the equation adding two parameters so that the user can decide the lower and upper limit of the random function. The mathematical formula for a variable  $X1$  to extend the random variation from  $[0,1]$  to arbitrary extremes:

$$X1 = \text{minimum} + (\text{maximum} - \text{minimum}) * X \quad (1)$$

Equation 1 shows that when  $X=0$ , then  $X1=\text{min}$ ; if  $X=1$ , then  $X1=\text{minimum}+\text{maximum}-\text{minimum}=\text{maximum}$ ; for intermediate values  $X1$  varies proportionally to  $X$ .

Let's implement an equation for  $X1$ . If you did not do that yet, close the Lsd model program (**File/Quit**) since we need to write a new equation in LMM. If LMM, for some reason, is not showing the equation file, choose menu **Model/Show Equation**.

In the equation file place the cursor below the line **MODELBEGIN**, above the line **MODELEND** and not inside the code of the equation for  $X$ . You are free to decide the order in which the equations' code appear in the equation file. Lsd will use the correct code for any variable and will use it when necessary.

As done before, choose **Edit/Insert Lsd Script** and choose **Equation**. Type the label **X1** for the new variable and press **Ok**.

In order to compute the values of variables an equation needs almost always to use the values of other elements in the model, either variables or parameters, and then make logical or mathematical elaborations on them. In the code for a Lsd equation we can use a Lsd function that provides the values of other elements in the model specifying their name. This function is called `V("VarOrParLabel")`, standing for "value of the element with label *VarOrParLabel*".

Using the `V("...")` function, the equation for  $X1$  could be written as (don't do that, yet) :

```
EQUATION("X1")

/*
  A random value within 'minimum' and 'maximum'
*/

RESULT( V("minimum") + (V("maximum") - V("minimum")) * V("X"))
```

Although the equation above is legal and working, it is a good practice to adopt a programming style that minimizes the computations required and improves the clarity of the code. This style suggests that any value from the system is requested only once (instead, in the expression above we request `V("minimum")` twice. Moreover, using the complete label of variables and parameters often makes the expression too long to be easily read.

Normally, equations should be written so that to assign to temporary C++ variables all the values required for the computations. Temporary variables are repositories that live only within the computation of one equation. In the equation file modellers have available the vector of temporary variables named `v[0]`, `v[1]`, `v[2]`, `v[3]`, ... Therefore, the equations for  $X1$  according to the normal Lsd equations' programming style is:

---

<sup>19</sup>If you don't know how random numbers are treated by computers, and have never heard the concept of "pseudo-random number" or "seed", you may want to read the section on this topic in the paragraph 5.6.7 (page 60) describing the random functions available in Lsd.

```

EQUATION("X1")

/*
  A random value within 'minimum' and 'maximum'
*/

v[0]=V("minimum");
v[1]=V("maximum");
v[2]=V("X");

RESULT( v[0]+(v[1]-v[0])* v[2])

```

In the first three lines of the equation we assign the values *minimum*, *maximum* and *X* to `v[0]`, `v[1]`, and `v[2]`, respectively, and eventually we write the operation in the `RESULT(...)` line, which is more easy to interpret.

To insert the `v[0]=V("minimum");` line and the others we have available a Lsd script. Trying placing the cursor in the desired location of the editor and choose menu item **Edit/Insert Lsd Script** and then choose the option **V("...")**. In the resulting window choose the desired index for the `v[...]` and the desired label for the value to request. Leave the other two options to the default values. When the window requesting the information for the script appears, you can use the keyboard to enter the values and the key Enter to move from one field to the next.

Notice that there is no difference in the equation between using parameters or variables. Actually, a good modelling style consists in implementing earlier versions of the model with all parameters and only one of few equations. Then, gradually, adding one equation for a former parameter transformed in an equation. The earlier equations will continue to work as before, with no change required.

After having concluded the coding save the file with **File/Save** (or using the shortcut pressing at the same time the key **Control** and **s**).

It is possible that you have typed an error in the code above, for example forgetting a semicolon at the end of a line, or forgetting one of the nested parentheses. If you try to run the Lsd model program with such an error the system complaints and will issue a new window asking whether you want to use the latest working Lsd model program available, or if you prefer to abort the process altogether (if you are in such situation, choose **Don't run** to abort the process). A new window will contain the message indicating approximately where the error has been found, so that you can fix it (see the appendix 5.6.7 on compiling errors.). Now we are ready to update the model configuration adding the new variable and the two parameters.

#### 4.4.7 Initializing Lsd elements - Lsd: Data/Init. Values

So far we have added the code for one equation. Therefore, after compiling (**Model/Run**), the Lsd model program is able to compute a new variable, *X1*. Therefore we need to define the new variable and parameters in the configuration.

First of all we need to retrieve the model configuration we have defined before: use **File/Load** and choose the Lsd file (likely called **Sim1**). What we need to do is to add three elements to the model: parameters *minimum* and *maximums*, and variable *X1*. To do this we need to move the Browser to show the *MyObj* object and then adding the two parameters using **Model/Add a Parameter** and the variable with **Menu/Add a Variable** (note that the parameters' labels in the Variable list have appended the symbol **(P)**).

Beware that the spelling of variables and parameters in the Lsd model program must perfectly match their spelling in the equation file, and that lower/upper capital letters matters. If a variable or parameter with the wrong spelling is inserted it is possible to edit its label. To do this double-click on the label of the element to edit so to open the option window (see fig. 7 at page 19) and double-click on the label of the element in the upper part of the window. Here you can change the nature of the element (e.g. from parameter to variable) or change the label spelling. Assigning an empty string to the label will remove the element altogether.

After having inserted the new elements we are still not able to run a simulation. In fact, the two newly inserted parameters need to be initialized, and an attempt to run a simulation would cause an error message to be issued and the simulation run aborted. To assign a value to the parameters in an object<sup>20</sup> you need to place the Browser to show the object concerned and then choose menu item **Data/Init. Values**. The Browser window is then transformed in a table showing one line for each element to initialize, as shown in fig. 9.

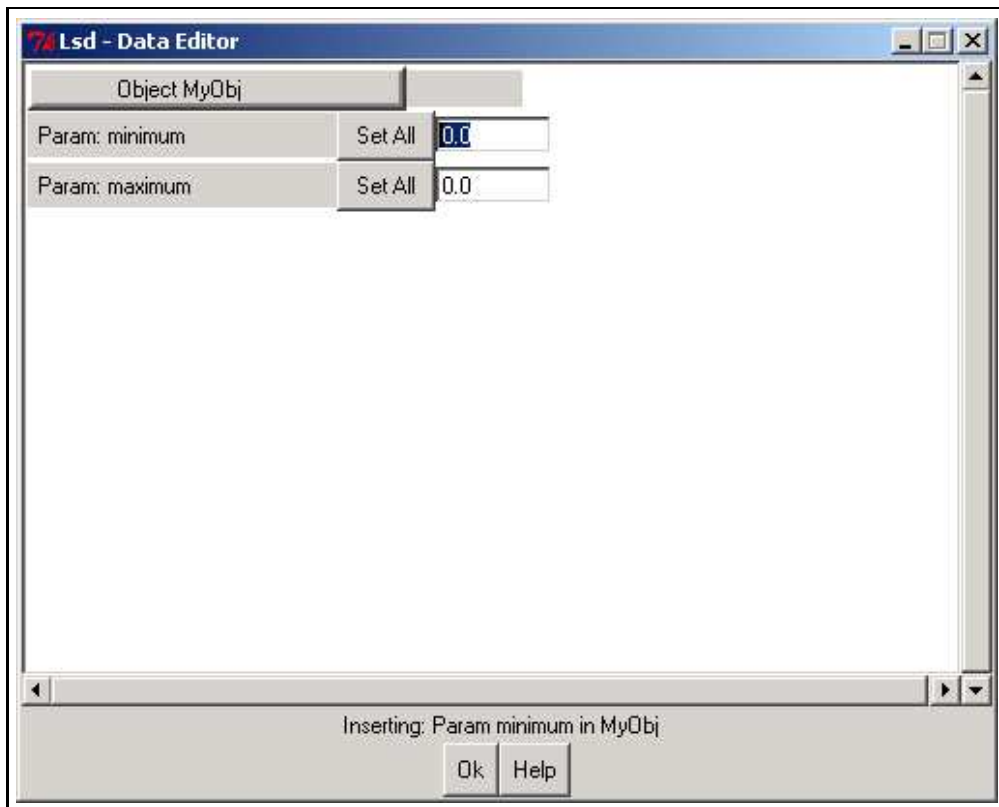


Figure 9: Initial values for *MyObj*

By default Lsd assigns a value of 0 to each element. You can type into the cells the elements desired, say -100 for minimum and 100 for maximum. Notice that the initial value window does not show the elements that do not need to be initialised, like *X* and *X1*. At the end of the initialization press **Ok** to return to the Browser.

Now we could run a simulation, although we may want to save the results of *X1*. Double-click on this variable and set on the option **Save** and the option **Debug**. Now we can run the simulation. The Run Time Plot window will report only the series of *X*, as

<sup>20</sup>Lsd permits to assign initial values only to the elements of one type of object per time.

before, because we did not set the option **Run Time Plot** for variable  $X1$ . However, the values of  $X1$  have been saved and we can observe them in the Analysis of Results window (menu **Data/Analysis of Results**). This time we have two lines in the list of the series available, one for  $X$  and one for  $X1$ . Observing their graph we see that they have the same dynamics, but for the range, as expected.

**Exercise 1** *What is the relation between the values of  $X$  and of  $X1$ ? In the Analysis of Result window check the box **Time Series** and the box for **XY plot**. Then insert in the **Series Selected** list firstly the variable  $X$  and then  $X1$ . This graph shows the values of  $X$  on the horizontal axis and the corresponding value (i.e. at the same time step) of  $X1$  on the vertical one. What is the result you expect and why? (if necessary, consult the help page).*

After having produced the graph, exit from the analysis of results and return the Browser. Since the Lsd model program contains the data from the latest time step, reload the configuration. You can use the menu **File/Re-load**, or the short-cut **Control-w**.

#### 4.4.8 Setting the number of objects - Lsd: Data/Set Number of Obj.

Until now we have worked with one single copy of the *MyObj* object we have created. But the power of object-based languages is that once you define a type you can easily work with as many copies as you want. In this paragraph we will multiply the copies of *MyObj* to see how Lsd manages not only single elements but whole sets of them.

If you still didn't do that, re-load a fresh model configuration (**Control+w**). Then, open menu item **Data/Set Number of Objects/All types of Objects**. The Browser window will become as in fig. 10.

This table shows only one line, since there is only one type of object. You can click on the number on the side of label for *MyObj* and insert the number of copies of this object you want in your model. For example, type 10. Pressing **Ok** you return to the Browser. As you see, nothing is changed, since the Browser shows only the structure of the model and not the number of copies. However, the graphical representation of the model now indicates that there are 10 copies of *MyObj*'s in the model.

The new copies have been created as identical copies of the only one previously existing, and therefore also the initial values for *minimum* and *maximum* are identical in all the copies, and the same settings for the elements (e.g. variables to save and/or to plot at run time) are applied. Open the menu **Data/Init. Values** to control for the parameters' values. Given the number of copies of *MyObj* now there are 10 copies for each type of parameters. Of course, users can set different values for each copy. The button **Set All** on the right of a label allows to set all the initial value for a parameter according to one of the available rules. For example, set the *minimum* values to increasing values starting from -100 and changing of 10 for each object. For doing this with **Set All**, click on the button on the right of the label for  $X$ . Check on the option **Increasing**. In the top entry marked as **Start** insert -100. In the second entry marked **Step** enter 10. Press **Ok** to confirm and exit. Now *minimum* is set to -100 for the first object, to -90 for the second, etc.

Set also *maximum* as starting from 100 and changing of -10 at each step. This will make Now the 10 objects will compute their  $X1$  varying over different intervals.

Running a simulation with the 10 copies will make no difference but that the series shown in the Run Time Plot have now become 10, identified with different colors and assigned a different number. Running the Analysis of Results after the simulation will reveal that 10 series for each variable have been saved, again each identified with a different

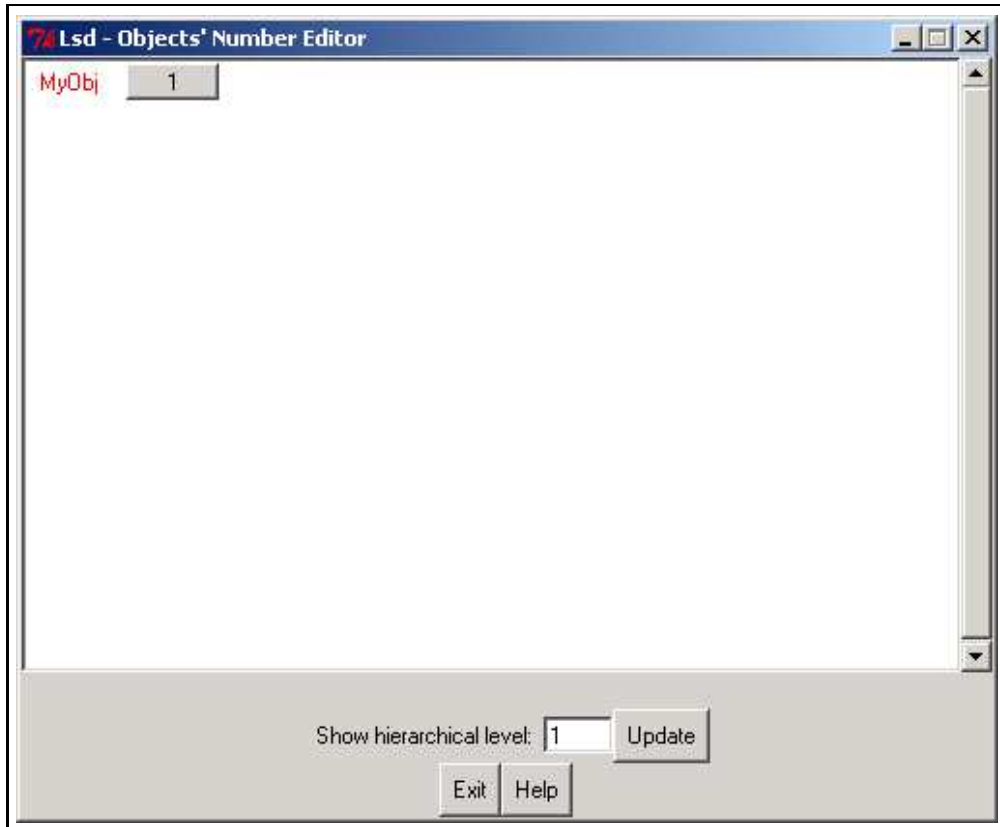


Figure 10: Setting the number of Objects

number. As you can see, the series of the later objects have a much smaller oscillation, since their range is smaller, than the earlier objects.

The results we have obtained are quite obvious: each of the 10 copies of the object computed the equations producing independent results. The code for the equation is only one, and it was used by all of the 10 copies of variables in the 10 objects. However, when, for example, the code for  $X1$  was computed the system has provided the values of parameters *minimum* and *maximum* in the same copy of the object whose copy of  $X1$  was computed. This is a property of the function  $V(\dots)$  and of all the similar functions: if there are many possible copies of the values requested, then  $V(\dots)$  chooses the copy “closer” to the copy of the variable whose equation is being executed. In our case, the  $X1$  variables are placed within an object together with the parameters *minimum* and *maximum* necessary to compute them, and therefore the  $V(\dots)$  returns those copies. For example, the copy of  $X1$  in the 3<sup>rd</sup> object will compute its equation making use of the copies of *minimum*, *maximum* and  $X$  contained in the same 3<sup>rd</sup> object. As we will see, programmers have a wide variety of options to write code that makes use of values in different objects. However, in the vast majority of cases you will not need to use these options, and can rely on the Lsd system to retrieve the correct values for you.

We will return on this topic concerning how Lsd solves potential conflicts like the presence of multiple copies of requested elements. However, for the moment, we continue to explore how users can affects other aspects of simulation runs.

After the simulation re-load the configuration with **Control+w**, so that we can see another aspect of configuring a simulation run.

#### 4.4.9 Simulation settings - Lsd: Run/Sim. Settings

The simulation runs we have launched up to now have all done 100 steps, since this is a default value assigned to a new model and we did not modify. Of course, users can edit this, and other options for controlling the simulations that do not concern directly the computation of the model. These options are set using the menu item **Run/Sim. Settings** which shows a window as in fig. 11.

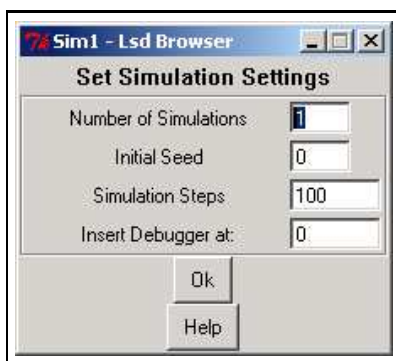


Figure 11: Setting simulation options

The options control the following behaviour of the simulation runs:

- **Number of Simulations:** by default users run one single simulation per time, control the results and then change either initial values or the equations before testing another (single) simulation run. The results are stored in memory only, unless the user saves them explicitly after the simulation run. However, in certain cases one is interested in knowing how the model behaves repeating several times the simulation using different random values, to test the robustness of some phenomenon. In these cases users can set the number of simulation runs to be higher than 1. The results of each simulation runs (i.e. the data produced during the simulation by the variables marked to be saved) are stored into files at the end of each simulation run with extension `.res`. Moreover, it is created a summary file (extension `.tot`) containing the very last value for each variable so to compare the values of saved variables from different simulations. Analysis of results can load any of these files and analyse the results there contained. Notice that if some variable is supposed to create Run Time Plots during a simulation, every simulation will produce a new Run Time Plot. If you set ask for 100 simulations, this will create 100 windows... The menu item **Run/Remove Run Time Plots** will remove all the windows at once.
- **Initial Seed:** computers are not able to produce random events. However, there are numerous mathematical tricks that provide sequences of values that have the same properties as random values. Setting the same “seed” for repetitions of the same simulation implies the use of exactly identical “random” events (which, in fact, are actually called “pseudo-random” events). Instead, setting different seeds produces different random events. If there are more than one simulation runs to be executed, the system automatically provides increasing seed values at the starting of each simulation run, and the same seed value is used to name the file where the results are stored, so to be able, if necessary, to reproduce one specific run with the same (pseudo-)random events.
- **Simulation steps:** the number of steps to be executed for each simulation run.

- **Insert debugger at:** : most of the time spent by programmers on whatever software project does not concern writing code, but the investigation of anomalous behaviour of the program, like unexpected crashes or absurd values. Simulation modelling is not an exception to this rule, so Lsd provides a “debugger”, that is a function that supervises the running of a simulation and, if necessary, interrupts it giving the modeller access to each and every value of the model in order to control what is actually going on in the model at that time. With this option users determine at which time step the debugger must be activated. If this value is 1, then the debugger is active from the very time step. If it is 0 or a negative value, the debugger is never activated. If it is, say, 56, then the debugger is activated at the 56<sup>th</sup> time step. When the debugger is active the simulation is interrupted as soon as one of the variables marked to be debugged (see variables’ options, fig. 7 at pag. 19) completes its equation<sup>21</sup>.

All the settings above, but the last one concerning when to activate the debugger, are stored in the file together with the model configuration.

**Exercise 2** *When running more than one simulation the summary file shows the latest value of saved variables. The series in this file, therefore, actually consider as “time step” the sequence of simulations. For example, the 5<sup>th</sup> “time step” in the summary file concerns the latest result of the 5<sup>th</sup> simulation.*

*Launch a battery of 20 simulation runs setting the initial seed to 1. Control that the last values reported in the simulation result files (extensions `.res`) match the ones contained in the summary file (extension `.tot`).*

#### 4.4.10 Documenting the model - Lsd: Model/Create Report

One of the biggest problems in the success of simulation modelling is that people reading a paper based on the results of a simulation model are sceptical since it is very difficult to understand the code written by someone else (and frequently people cannot even understand their own code after months they have written it). The solution is to fully document the simulation program, including comments explaining what the code does, why those initial values have been chosen, and how the elements of a model interact together. Of course, this is a tedious work, that hardly any modeller does. Lsd tries to obviate to this problem providing a fully automatic documentation of models providing a sort of “manual” for each model that immediately permits to understand the code and the working of the model.

Each element of a model (object, variables and parameters) is endowed with a textual documentation, called **description**, that modellers can use to describe what the element does. For the modeller it is therefore easy to sketch out in few words for each element the description of each element. Moreover, Lsd does part of the job automatically, obviating to the lack of documentation from lazy modellers. Let’s see how Lsd generates the model documentation automatically.

Use menu item **Model/Generate Automatic Descriptions** (and subsequently choosing **All elements**). The system scans all the model elements and the equations’ code and allocates the resulting information for each and every element. The information differs for different elements, indicated below:

---

<sup>21</sup>Notice that Lsd users can also use standard C++ debuggers, like GDB (included in the Windows distribution) which instead give access to a simulation run not every equation completed, but every single line of code completed. However, the use of these debugger is quite complicated, while the use of the Lsd debugger is simple and intuitive.

- **Object:** Include the list of the equations where the name of the object appears (for example because the variable creates or destroys the object).
- **Variable:** If exists, copy in the description the comment written by the modeller in the very first lines of the code for the equation of the variable. Then include the list of variables whose equation contains the name of the variable, typically because they use its value.
- **Parameter:** Include the list of variables whose equation contains the name of the parameter, because they use its value.

Besides the description, parameters and any other element that need to be initialised<sup>22</sup> are also endowed with a text document describing how the values for the elements have been decided.

The elements' descriptions (and the comments on the initialization) appear in the options' window of the model elements activated by clicking on the element label in the Browser (see fig. 7 at pag. 19). In this window it is also possible to gather other potentially useful information concerning an element, like the code of the equation for a variable, or the list of the elements used that can be used to open these elements' option window.

These Lsd interfaces and texts allow a very easy way to comprehend how even a large model, with many elements works and is simulated. However, it is also possible to export all this information in a standard HTML document, called *Model Report*, that allows an exploration of the model using a standard HTML browser (i.e. Netscape). To create the *model report* for your model use menu item **Browser/Create Report**. The Browser will open window like the one depicted in fig. 12.

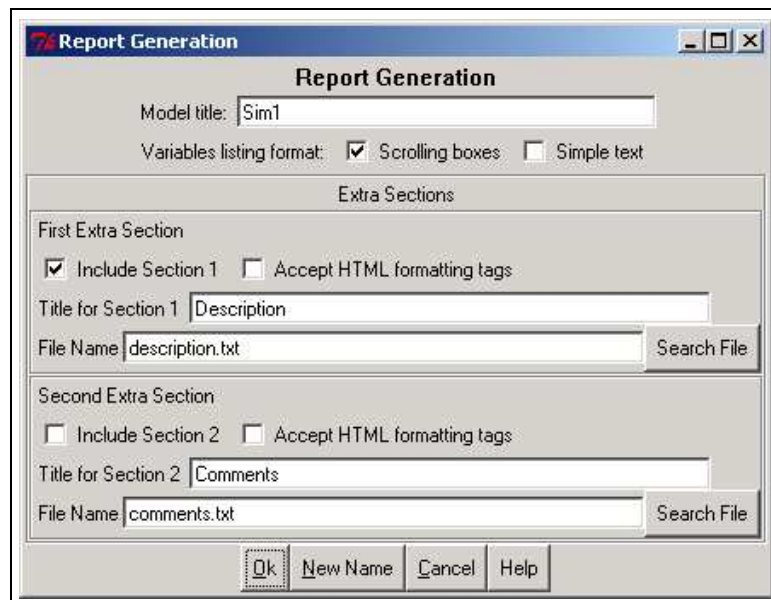


Figure 12: Options for the Model Report

This window permits to set several options, for example including an external file with comments on the aim and the results of the model. However, most of times the default options should suffice.

<sup>22</sup>We will see in a moment that some variables need to be assigned one or more values in order to start a simulation. These are the variables that are used with past values in some equation, and that, therefore, in the very initial steps of the simulation require user-defined values for “negative” time steps.

Pressing **Ok** will start the creation of the report. At the end of the process the system will start your default web browser to open the report (see the next paragraph for the use of the *Model Report*).

Notice that the operation of updating the description of the variables of the model costs just two clicks of the mouse, and the same applies for the creation of a new report. Therefore, this document is also a very useful tool for modellers when they are revising a model editing the equations in order to control for possible conflicts with other elements of the model.

#### 4.4.11 Using the Model Report - Lsd: Help/Model Report

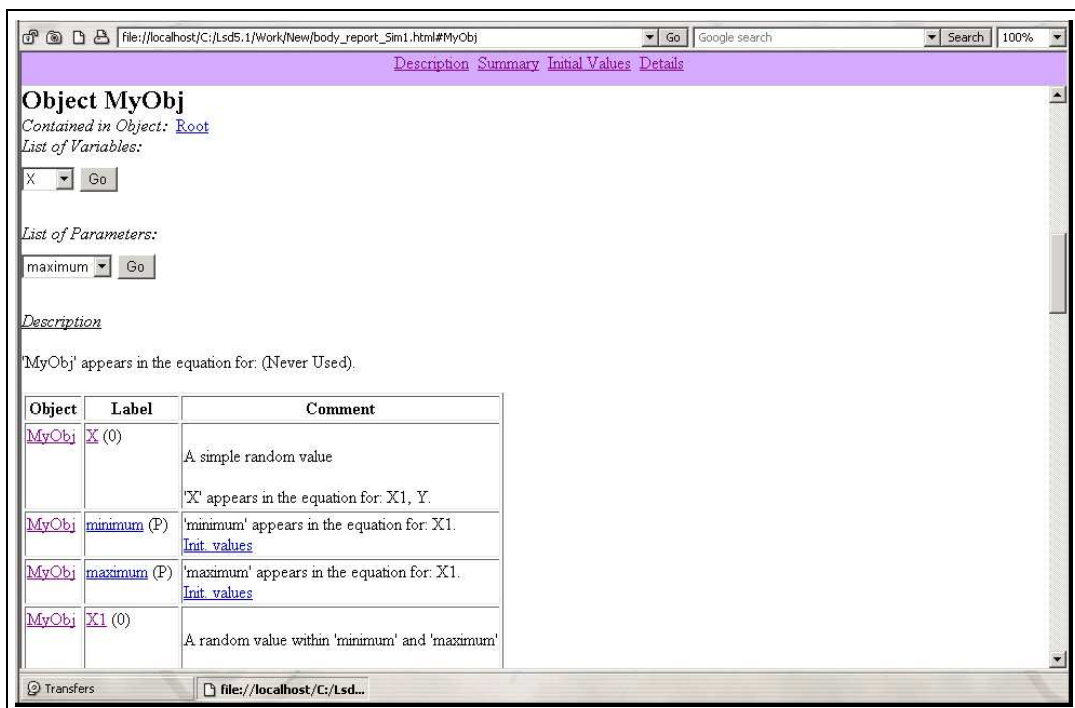


Figure 13: Example of Model Report

Fig. 13 shows a portion of the Model Report for the model developed until now. The Model Report is opened using the menu item **Help/Model Report**.

The report is composed by four sections reporting different types of information concerning the elements of the model, each of which is linked to the others with hyperlinks in order to easily move around the report.

The sections, whose beginning can be reached clicking on the header bar of the document, contain the following information:

- **Description:** If exists, this section includes a text describing the model or any information the modellers considers relevant for the users. Our Model Report lacks, for the moment, a description section.
- **Summary:** This section begins with a textual representation of the objects forming the model. The labels of the objects can be clicked to reach that object's sub-section. Then there are the sub-sections for objects, each formed by a table whose columns report:

1. A link to the beginning of the subsection to the same object
  2. The label of the parameters and variables in the object, which is also a link to the row of the same element in the section **Detail**.
  3. The text forming the description of the element and, if exists, the text commenting the initial values for the element.
- **Initial Values:** This section begins with a textual representation of the objects forming the model. The labels of the objects can be clicked to reach their subsection. Below are listed the sub-sections for the objects, each formed a label of the objects and the number of copies for that object in the model, and by a table whose columns report (only for parameters and variables requiring initial values):
    1. The label of the parameters and variables in the object, which is also a link to the row of the same element in the section **Description**.
    2. The values used to initialise the element for a simulation run.
  - **Details:** This section begins also with a textual representation of the objects forming the model, which can be clicked to reach their subsection in this section. Then there is the list of elements reporting the most detailed information, which differ depending on the nature of the elements (any element cited is a link to its cells in the summary section):
    - **Objects:** the detailed description of objects report the object that contains it; the list of objects contained; the list of variables; and the list of parameters.
    - **Variables:** the detailed description of variables contain: the label of the object that contains it; the list of the variables and parameters used in the equation for the variable; the list of the variables whose equation makes use of its values; two links to the sub-section for the variable in sections **Summary** and **Initial Values**<sup>23</sup>; a link to the beginning of the section **Summary**. Eventually, this subsection contains also the code for the equation of the variable (whose quoted model elements are themselves links).
    - **Parameters:** the detailed description of parameters includes the link to the object containing the parameter; the list to the variables that use the the values of the parameter; three links to the sub-section for the parameter in section **Summary**, to its sub-section in section **Initial Values**, and to the beginning of the section **Summary**.

The Model Report provides not only all the information concerning the elements of the model, but also how they are connected, thus providing a useful guide on how, for example, changing an equation affects the model. Moreover, the report provides readers of a model with the complete knowledge concerning the model components and how it has been implemented and initialized, all in a very easy to access manner. Ideally, modellers presenting the results obtained via simulation could distribute the model report in order explain readers the inner mechanism of their model, without requiring a full immersion in the whole code of the model program itself.

The Model Report is extremely useful to be kept updated during the development of the model. All the descriptions of the model elements are stored in the configuration files, and therefore loaded and saved together with the configuration itself. When a new

---

<sup>23</sup>This latest link exists only if the variable requires initial values.

elements is added to the model (i.e. a new variable with its equation) it is enough to add the description of the new element in the configuration and re-create the Model Report.

#### 4.4.12 Using lagged variables - LMM: using VL("Var",lag)

One of the main reasons for using simulation models is that understanding the result of even simple temporal dynamics is very complicated. The equation we have implemented until now are not really dynamical, since they compute values as elaboration of present-time values, that is, at the same time step. We have a real dynamics when we make elaborations over values from the past.

The most simple dynamical function is the so called *random walk*. The equation for random walk is (note that we now use the temporal index t):

$$R_t = R_{t-1} + U(-k, +k) \quad (2)$$

That is, the value at any time t of a variable following a random walk is equal to the value of same variable at the previous time plus a random value, drawn from a range which normally includes both negative and positive values. Let's implement a Lsd equation to see how to express the variable's values with temporal lags. The code for the equation of the random walk is<sup>24</sup>:

```
EQUATION("RandomWalk")
/*
A random walk variable whose range of oscillation is in between
'minimum' and 'maximum'
*/

v[0]=VL("RandomWalk",1);
v[1]=V("X1");

RESULT( v[0]+v[1])
```

As you see, the equation uses the function VL("Var",n) (where n is a positive integer). This function works as the original function V("Var"), but for the fact that it returns not the value at the same time step, but with n lags, that is from n previous time steps. Note that V("Var") is an expression identical to VL("Var",0).

Save the equation file and run the Lsd model program (**Model/Run**). Load the configuration from the file (**File/Load**), reach for the *MyObj* (double-click on it) to add the new variable *RandomWalk* as we have done before for *X* and *X1*. Now we are ready... to face our first abrupt simulation crash.

Run the simulation and you will see the alarming message shown in fig. 14.

This window tells us that Lsd has encountered a logical error preventing it from continuing the simulation. This means that somewhere we have written some code for an equation that is perfectly legal from the computational viewpoint, but is wrong given the specific context of model status. Let's see now how we must proceed in order to find out where the problem is.

We may already suspect that the problem lies in the equation for *RandomWalk*, since it is after the insertion of this equation that the model failed. However, to be confirmed on this let's follow the indication of the error window and read the messages in the log window:

---

<sup>24</sup>At this point we assume the reader is able to remember how to use the menu **Insert Lsd Scripts**.



Figure 14: Simulation aborted for a serious logical error.

...

Lag error: Variable RandomWalk requested lag 1 but available only 0

Fatal error detected at time 1. Offending code contained in the equation for Variable:  
RandomWalk

...

The text above says that a “Lag error” has occurred concerning the variable *RandomWalk*. The second part of the message tells us the time step and the equation during which the error occurred, confirming our suspect.

The problem is that Lsd tries to save as much memory as possible in order to allow for simulating very large models for many time steps. When a variable computes its value, the memory used to store this value is freed when the time step finishes and a new one begins to be computed. Therefore, if we want to know the value of a variable at time  $t-1$  during time step  $t$ , we need to explicitly tell the system to maintain the value for one time step more than the default.

We should normally do this when creating a variable. If you remember, when adding a variable to the model we have two fields available: one for the variable’s label, and one for the **Maximum lags used**, which is set to 0 by default. We should have told Lsd when creating *RandomWalk* that this variable needs to store 1 lagged value.

We didn’t, so we need to fix this problem. However, at this moment the Lsd model program is in the status of aborting a simulation. The only commands available in this status are shown in the window shown in fig. 14. We can:

- **Abort:** shutdown the Lsd model program altogether (which we will do in a moment);
- **Analysis:** access the **Analysis of Results** module having available the data saved produced up to the time step before the one of the crash. This is useful when the crash occurs after several time steps have been successfully completed, to understand what when wrong. In our case, the crash occurred at the very first time step, so that we don’t have any data to analyse. understand why a model
- **Debug:** inspect the state of each and every element of the model. This functionality (which we still have not explored) allows to understand what is the value of each element (even the ones not saved) when the crash occurred.

Let’s kill the Lsd model program clicking on **Abort**. We return to LMM and run again the Lsd model program (menu **Run/Run Model**). In the Lsd model program load

the configuration and let's move to fix the error: move the Browser to show the object *MyObj* and double-click on the variable *RandomWalk*. We are shown the options' window; double-click on the label of the variable (in red on the top of the window) and we have the possibility to modify the nature of this element (see fig. 15).



Figure 15: Change nature of an elements.

We have the possibility to edit the label of the element (handy when you have misspelled it), or to transform a parameter in a variable and viceversa. And to modify the number of lagged values saved for the variable. Choose the middle option and set **Lags** to 1. Press **Ok** and you return to the Browser. As you see, now *RandomWalk* appears in the list of **Variables** of *MyObj* with attached the number 1, while *X* and *X1* have 0. This means that *RandomWalk* is a 1 time step lagged variable.

Can we finally run a simulation now? Not yet. At the time 1, the very first time step, the equation for *RandomWalk* says that  $RandomWalk_1 = RandomWalk_0 + X1_1$ . But where the system take *RandomWalk*<sub>0</sub>? There is no time step 0, and therefore this value must be provided by the modeller. It is the value from which the random walk has to start: it can be any value and Lsd does not have any clue on which one to use.

*RandomWalk*<sub>0</sub> is an initial value, exactly as any parameter's value. To set this we use the same interface used to set *minimum* and *maximum*: choose **Data/Init. Values** and you are shown the window to set all the parameters of *MyObj* plus the newly inserted variable's lagged value<sup>25</sup>.

Let's set **RandomWalk -1** to 1000 (use **Set all** for setting all the values in the 10 copies of the object) and return to the Browser. Double-click on the **RandomWalk** label in the **Variables** list and set on the options for saving and debugging this variable.

Return to the Browser and run the simulation (**Run/Run**. After that enter in **Analysis of Results** and plot some time series for the *RandomWalk* variables to observe their dynamics. When finished, exit the **Analysis of Results** and, in the main Browser, re-load the configuration with Control+w.

#### 4.4.13 Multi-layered object structure - Lsd: Editing Model Structure

Until now we have worked with a model with only one type of object, *MyObj*. It is a "flat" model, in that every copy of the object works on its own, independently from the others. However, generally models includes objects at different hierarchical levels. These "deep" models have agents (e.g. firms) interacting via a higher order entity (e.g. market). For example, a model for a market may contain an object *Market* containing, in turn, two types of objects *Demand* and *Supply*. Therefore, *Market* is "higher" in the hierarchy, representing a more aggregate element than *Supply* or *Demand*. In turn, *Supply* may

<sup>25</sup>Note that if we had defined *RandomWalk* having 2 lags (because an equation contained the expression  $RandomWalk_{t-2}$ ), then we would have two lines in the initial values: one for *RandomWalk*<sub>0</sub> and another for *RandomWalk*<sub>-1</sub>.

contain several objects *Firm*'s, as objects still lower in the hierarchy (i.e. smaller). In this paragraph we will see how Lsd manages multi-layered models.

Let's modify our model where the *MyObj* we have used up to now is contained within another object in which to compute aggregate statistics, like the average value for all the random walk variables<sup>26</sup>. Place the Browser to show the *MyObj* and then choose menu item **Model/Insert New Parent**. In the resulting window type the name of the new object, say **AggregateObject**. After confirming, verify that your model now includes *Root* containing *AggregateObject* containing *MyObj*.

If you open now the interface to set the number of copies for the objects (**Data/Set Number of objects/All types of objects**), you will see that the window shows one copy of the newly created *AggregateObject*. Along the button with **1** a text signals that there are other objects contained in the object shown. Click on the text<sup>27</sup> to make appear the number of objects descending from *AggregateObject*.

If you want to create a new high level object, we have seen that Lsd automatically creates all the parameters, variables and values necessary. Now you can verify that it creates also the objects contained. Try to increase the number of copies for *AggregateObject* from 1 to 3. As result, the window will show that the model is now composed by three identical copies of the highest level objects, with the same level of descending lower level objects. You can change independently the number of copies for each of the groups of *MyObj*'s in the three *AggregateObject*'s, or modify at once all of them, depending in whether you mark on or off the option **All equal** when requested to enter the number of copies for a single group of *MyObj*.

Save the configuration with 3 copies of *AggregateObject* containing 10, 20, and 30 copies respectively. When done press on **Exit** to return to the Browser. Notice how the graphical representation of the model on the right of the Browser now expresses the new configuration.

Now that we have created an aggregate object, let's place in it a variable. Put the Browser on the *AggregateObject* and add a variable labelled *AverageRW* (**Model/Add a Variable**). Save the configuration (**File/Save**) and close the Lsd model program: we need to update the equation file to add the new variable's equation, and this can be done only with no Lsd model program running.

#### 4.4.14 Equations for multi-layered models - LMM: using `CYCLE(...){VS(...)}`

Back in LMM let's write the equation for *AverageRW*. The code should simply implement the expression for the average value of all the *RandomWalk*, supposing having n copies of objects *MyObj*:

$$AverageRW_t = \sum_{i=1}^n \frac{RandomWalk_t^i}{n}$$

From the programming viewpoint we have the problem of ensuring that each copy of *AverageRW* is computed using with all, and only, the values of *RandomWalk* contained in the group of objects *MyObj* descending from one specific copy of *AggregateObject*. That is, we want to be sure, for example, that the second copy of *AverageRW* does not include in the computation values of *RandomWalk* from objects contained in the first or third copy of *AggregateObject*. Moreover, when writing the equation, we cannot know how many copies

---

<sup>26</sup>Actually, we already have a higher level object, *Root*. But, as already said, it is better to never use *Root* to contain other than objects.

<sup>27</sup>Alternatively, you could type '2' into the entry for the hierarchical level to show and click on **Updated**.

of *RandomWalk* are used, since this is defined by the users in the model configuration, and we want our equation to work in general for any number of copies.

The language used for the Lsd equations permits to overcome these problems in a very easy and intuitive way. Let's see the code for the equation and will learn to use two new Lsd functions. The code for the equation is the following:

```
EQUATION("AverageRW")
/* Average value of all the RandomWalk
values from the descending objects
*/

v[0]=0; v[1]=0;
CYCLE(cur, "MyObj")
{
  v[0]=v[0]+VS(cur,"RandomWalk");
  v[1]=v[1]+1;
}

RESULT( v[0]/v[1])
```

Let's see in detail the commands in the code above, keeping in mind that the equation is computed for a variable contained in *AggregateObject* and must use the values of *RandomWalk* contained in the lower level *MyObj*.

The first two lines of the code simply set to 0 two temporary variables, which will be used to cumulate the values of *RandomWalk*'s and the number of their copies of *MyObj*. The actual calculation is done in the command `CYCLE(cur, "MyObj"){repeated code}`. This command causes the code contained in the subsequent group of lines to be repeated again and again once for each copy of *MyObj*. Therefore the number of repetitions depends on the number of copies of the objects whose label is *MyObj*<sup>28</sup>. During each repetition the cycle assigns to `cur` a different copy of *MyObj*. `cur` is the equivalent for objects of what temporary variables are for numbers: a temporary memory location for an object. In programming language `cur` is a "pointer", that is a variable that can be assigned an area of memory, in our case Lsd objects<sup>29</sup>.

In the body of the cycle, that is the code included in between the two curly brackets, there are two commands. Both commands cumulate numerical values. The second cumulate simply 1's, so that, after the cycle, `v[1]` contains the number of repetitions of the cycle, that is, the number of *MyObj*. The first line cumulates in `v[0]` a more complicated stuff: `VS(cur, "RandomWalk")`. This command, as you may have guessed, is another form of the `V("...")` family, returning the values of elements (i.e. variables or parameters) of the model. The specificity of `VS("...")` is that the programmer must specify the object where the element is contained. Instead, as we have seen, the general form `V("...")` delegated the system to retrieve the object containing the element to be evaluated. The general form `V("...")` cannot be used in this equation. In fact, the code for the equation of *AverageRW* is executed at level of object *AverageObject*. In these objects there is no variable *RandomWalk* so the system starts a search for an object containing the desired variable. But all the descending objects *MyObj* are equally "distant" from the object

<sup>28</sup>If there are no copies of *MyObj* descending from the *AggregateObject* containing the *AverageRW* whose equation is executed the internal calculations in the cycle are never executed.

<sup>29</sup>The pointer `cur`, as well as the numerical variables `v[i]`, are local C++ elements that a modeller has available to store temporary some element of the model. They are created for each equation and destroyed at the end of the computation, so that they cannot be used to transfer values from one equation to another.

*AggregateObject*, where the equation for *AverageRW* is executed, and therefore the system cannot correctly find any specific copy of *MyObj*<sup>30</sup>.

Using `V("...")` instead, we tell the system in which object is contained the element to compute. We tell the system to use the copy of *RandomWalk* contained in `cur`, which, through the iterations of the cycle, “points” to all the copies of *MyObj* descending from the *AggregateObject* whose *RandomWalk* has to be computed.

Eventually, when the cycle is finished, the equation has scanned every *MyObj*; for each of them the equation has added 1 to `v[1]` and the value of *RandomWalk* to `v[0]`, so that we have the denominator and numerator of the division for the average, which is then assigned as result.

#### 4.4.15 The environment of Lsd equations

Before continuing let’s briefly consider how Lsd treats the equations code. A model in Lsd is a data structure: objects containing variable, parameters and other objects. The equations must be thought of by modellers as abstract computations to be executed at the general time step  $t$  by the general copy of that type of variable. The equations we have written are the computational translation of a difference equation model that may be expressed with the following equations:

$$\begin{aligned}
 \text{AverageRW}_t^i &= \sum_{h=1}^{n_i} \frac{{}_t\text{RandomWalk}_t^{i,h}}{n_i} \\
 \text{RandomWalk}_t^{i,j} &= \text{RandomWalk}_{t-1}^{i,j} + {}_t\text{X1}_{t-1}^{i,j} \\
 \text{X1}_t^{i,j} &= \text{minimum}^i + (\text{maximum}^i - \text{minimum}^i) \text{X}_t^{i,j} \\
 \text{X}_t^{i,j} &= r.v.U(0, 1)
 \end{aligned} \tag{3}$$

The Lsd version neglects the time suffix  $t$ , which is redundant (but includes the lag indication when necessary), and relies on the model’s structure of objects instead of using the impractical and error-prone indexing system. In the rest of this paragraph there are some further details on the functioning of the Lsd equations.

In the beginning of each time step each variable in the model is scanned following and arbitrary order and the equation for the variables is executed to provide the new value for that time step<sup>31</sup>. If an equation requires other variables to be completed, as it is typically the case, then this equation is temporarily interrupted, the required variables are updated (i.e. their equations are computed), and then the original equation is completed. If a variable is requested after having being already updated for that time step, then it returns its previously computed value, without re-executing its equation. This mechanism permits modellers to ensure that variables are updated in the correct order as implicitly specified by assigning lagged values, disregarding the order in which equations are listed in the equation file or variables are listed in the model definition.

In our case, as in practically any model, there are many copies for each variable, one in each copy of a type of object. Every copy of a variable executes the same computations, though on different values. For example, every copy of *RandomWalk* needs its own lagged value and its own copy of *X1*.

Lsd manages to obtain this by using a hidden information to the equations’ code. When the equation is activated all the Lsd functions, like `V("...")`, “know” the copy of

---

<sup>30</sup>Using `V("RandomWalk")` in an equation for a variable in *AggregateObject* will return always the value of the very first copy of *RandomWalk*.

<sup>31</sup>Actually, the order of execution is, firstly, the higher order objects’ variables, and then their descending objects.

the object containing the variable whose equation is executed. This bit of information is available to the modeller, too, as the object pointer (i.e. the C++ variable storing objects) called "p". For example, the expression `V("X1")` is equivalent to the expression `VS(p,"X1")`. In the former case we rely on `Lsd` to find out the correct copy of `X1`. The system is executing the equation for `RandomWalk` in some copy of `MyObj`, so that `V("X1")` will return the value of `X1` contained in the same object. Instead, in the second case, using `VS(p,"X1")`, we tell the system to return the value of the copy of `X1` in `p`, which is the same copy of `MyObj` containing the copy of `RandomWalk` under computation.

There are many functions and technical variables available to the modellers for expressing their equations. The general approach of `Lsd` coding is to offer modellers and easy way to perform the most common operations, so that it is very simple to write simple modelling expressions. But `Lsd` allow modellers also to intervene on every aspect of the simulation, if necessary, so that it is possible to express whatever computation one may need.

#### 4.4.16 Extending the model: quality and sales

We are proceeding gradually adding new equations one by one. For the moment we have implemented just a set of random variables that independently create a set of random walk processes, and we have computed an average of these processes. Let's interpret this model as a metaphor for (very simplified) processes of technological improvements.

That is, we interpret `MyObj` as a firm performing R&D whose result (totally random) is a quality level for its product (the `RandomWalk` variable). We want to link the relative quality of each firm (relative to the average quality) to the level of sales. That is, we need two new variables, and their equations. In mathematical terms the equations are:

$$RelativeQuality_t = \frac{RandomWalk_t - AverageRW_t}{AverageRW_t}$$

$$Sales_t = Sales_{t-1}(1 + alpha * RelativeQuality_t)$$

The variable `RelativeQuality` should not pose any problem of interpretation. If a firm (i.e. `MyObj`) has a quality level (i.e. `RandomWalk`) identical to the average of all firms, then its relative quality is null. If it is better, its relative quality is positive, otherwise it is negative.

The second equation deserves some more attention. It states that the level of sales for a firm is inertial, changing slowly in the same direction of relative quality. The equation says that the value of sales at any time step equals its previous value plus a share (`alpha`) which is added or removed depending on the relative quality. It is implied that two firms having identical relative quality may differ pretty much in their level of sales. The difference resides in the previous values of sales, since the equation of sales says that the relative quality determines the relative growth of sales: both firms may grow of, say 20% in respect of their previous size, which may be quite different in absolute levels.

Let's see the equation's code for `RelativeQuality`. It is:

```
EQUATION("RelativeQuality")
/*
Compute the relative quality
of a firm as the relative ratio
of the difference between RandomWalk
and AverageRW
*/
```

```

v[0]=V("RandomWalk");
v[1]=V("AverageRW");

RESULT( (v[0]-v[1])/v[1] )

```

There is nothing exceptional in this equation, but it is worth noting something we still did not meet in our model. *RelativeQuality* is a variable stored in *MyObj*. Therefore, the `V("RandomWalk")` expression returns the value in the same copy of *MyObj* containing the copy of *RelativeQuality* executed. But what about `V("AverageRW")`? *AverageRW* is a variable contained in *AggregateObject* that is in an object “higher” than the one where the equation is executed, *MyObj*. How does Lsd manage this situation? Assuming the obvious: the copy of *AverageRW* returned in each equation is the one contained in the *AggregateObject* copy containing the copy of *MyObj* whose *RelativeQuality* is executed. This automatic retrieval of data in “upper” object is possible because Lsd models are strictly hierarchical, so that a lower level object is directly linked to only one higher level object.

The second equation’s code does not pose any particular problem:

```

EQUATION("Sales")
/* Compute the level of sales as the relative
growth proportional to alpha of the
relative quality */

v[0]=VL("Sales",1);
v[1]=V("RelativeQuality");
v[2]=V("alpha");

RESULT(v[0]*(1+v[2]*v[1]) )

```

Now we can compile the model and, if there are no grammar error in the equations’ code<sup>32</sup>, we can use Lsd model program for a quite long set of operations, but that you should find almost automatic, by now:

1. load the configuration file (**File/Load**);
2. move the Browser to show the object *AggregateObject*;
3. add the parameter *alpha* (**Model/Add a parameter**);
4. initialize the parameter to 0.2 (**Data/Init. Values**);
5. move the Browser to show the object *MyObj*;
6. add the variable *Sales* (**Model/Add a variable**, indicating that it uses 1 lagged value);
7. add the variable *RelativeQuality* (**Model/Add a variable**);
8. initialize the values in *MyObj* (**Data/Init. values**) setting all the *Sales*’ values to 1000.

---

<sup>32</sup>If there are errors, choose not to run the existing Lsd model program and see the second line in the Compilation Results. There should be a line number, indicating approximately where the error has been found.

Before running a simulation double-click on the labels for *Sales* and *RelativeQuality* setting on the option to save the variables' values, so that we will be able to analyse their values after the simulation run.

If there are errors appearing after the simulation has been launched, chances are that you misspelled some of the newly inserted variables or parameter. Remember that elements' spelling must be identical in the Lsd model configuration and in the equation file. If you have an error, check in the Log window which element could not be found by the system. Then re-start the Lsd model program from LMM and load the configuration. You will be relieved seeing that all the modifications you made to the configuration have been saved just before running the simulation, so that you will just need to edit the label of the elements to be corrected.

#### 4.4.17 Assessing the model's behaviour

Let's see how the model behaves. First of all we need to configure the initial values in a sensible way. Set the number of *AggregateObject* to 1 containing ten copies of *MyObj*, and the *alpha* parameter set to 0.2. Set the values for parameters *minimum* and *maximum* are all set to -10 and 10 respectively. This ensures that the random walks are centered on 0 without systematic biases. Finally, set the all the lagged values for *RandomWalk* and *Sales* to 1000, just to remember where we did start from.

Any user of Lsd running this model with the above initialization will produce exactly the same results. For example, in figure 16 it is reported the series for the relative quality levels over the first 100 time steps.

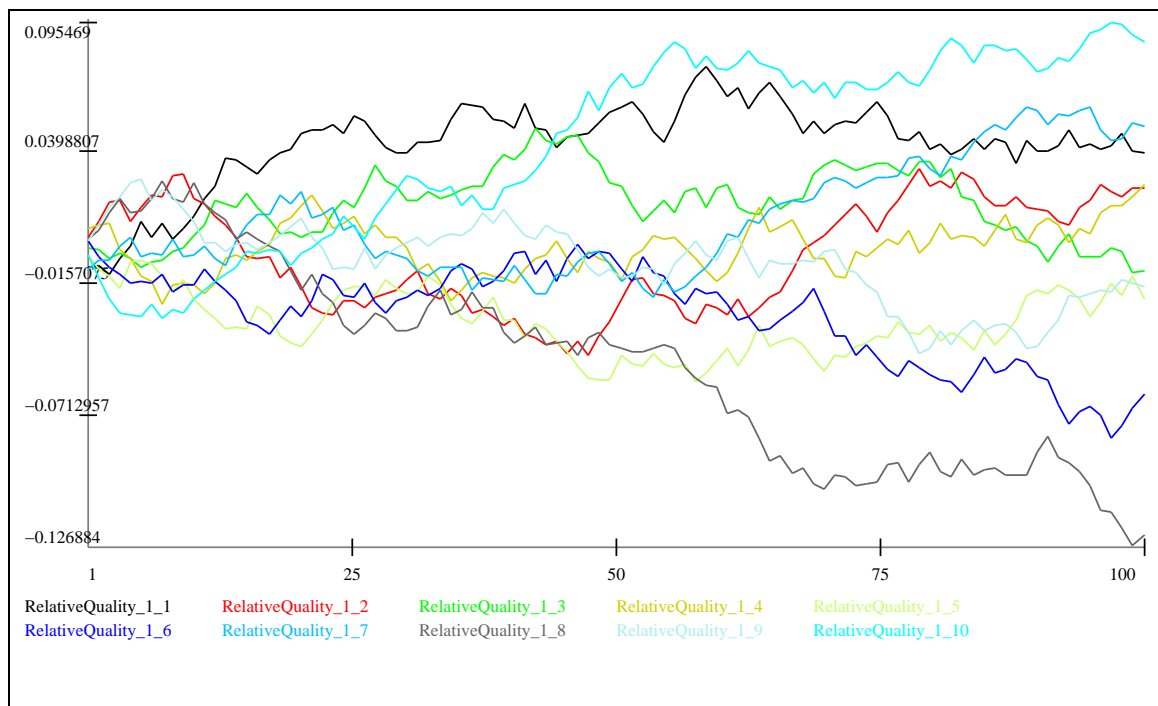


Figure 16: Relative quality over time steps from 1 to 100.

One may wonder how it is possible that supposedly random events happen identically on any computer run. This is one of the advantage of using simulations. You do have random events, but you can re-run a simulation using exactly the same set of (pseudo-)random events so to control what exactly happened. If you want, as usually is the case,

to run the same simulation (i.e. same initialization) with different random events, then you need to change the **seed** value, which is set in menu **Model/Sim.Setting**.

If we control the graph of the *Sales*' variables we see that there is a correspondence between the latter and their *RelativeQuality* values<sup>33</sup>.

However, if we extend the simulation to span over a longer period, say, 1000 time steps (**Run/Sim.Setting**) we see something unexpected. The level of sales for one of the firms increases exponentially. The reason is that for the firms having above average quality (i.e. *RandomWalk*) tend to remain so for some time, and consequently the increment of sales gets bigger and bigger in absolute terms.

Given our interpretation of sales improving according to the relative quality in respect of the average, what we need to change is the way the average is computed. The simple average we have used so far considers all firms as equally important. Instead, if we want to interpret this average as an indication of the average quality a general consumer can interpret, we need to ensure that the individual firms' qualities are compared with the *weighted* average of qualities, where the weights are given by the level of sales.

In the next paragraph we will implement the weighted average, and we will discover an unexpected error, and the solution offered by Lsd.

#### 4.4.18 Replacing a variable - implement *WAverageRW*

The equation for a weighted average is simply:

$$WAverageRW_t = \frac{\sum_{i=1}^n RandomWalk_t^i * Sales_t^i}{\sum_{h=1}^n Sales_t^h}$$

This expression guarantees that firms' qualities with higher sales levels "count more" than firms with lower level of sales. Consequently, we will not observe an absolute growth of the total level of sales, but only a different distribution of a constant amount. More on this when we will be able to run the model. Before this, however, we will have to fix some problems.

Let' begin to implement the equation. The code for *WAverageRW* is:

```
EQUATION("WAverageRW")
/*
Weighted average value of all the RandomWalk
values using Sales as weights
*/
//set to 0 two temporary variables
v[0]=0;
v[1]=0;

CYCLE(cur, "MyObj")
{//cycle through all MyObj copies
//compute the value of RandomWalk for MyObj in 'cur'
v[2]=VS(cur,"RandomWalk");
//compute the value of Sales for MyObj in 'cur'
```

---

<sup>33</sup>In Analysis of Results select all the *RelativeQuality* and all the *Sales* series. Then check the options for **Cross Section**, **XY plot**, and **Points**, and press **Plot**.

```

v[3]=VS(cur,"Sales");

v[0]=v[0]+v[3]; //sum of Sales
v[1]=v[1]+v[2]*v[3]; //sum of Sales*RandomWalk
} //end of the cycle

RESULT( v[1]/v[0])

```

In order to use this new variable, we need to upgrade the equation for *RelativeQuality* (see the equation's code in paragraph 4.4.15 at pag. 36). In the equation replace the line:

```

...
v[1]=V("AverageRW");
...

```

with

```

...
v[1]=V("WAverageRW");
...

```

Compile and run the Lsd model program (**Model/Run**). Load the configuration and move the browser in *AggregateObject*. Here add the new variable *WAverageRW*. We are now ready to run the simulation, but it will last long...

#### 4.4.19 Dead-lock errors - Spotting and fixing temporal inconsistencies

The simulation aborts immediately. What is the problem? Writing simulation models, like any computer program, is prone to two types of errors: firstly, we may write *grammar* errors. These errors, that you are likely to have already experienced, are mistakes in the code such that the compiler cannot understand the commands in the code. Typically, we can forget the semi-colon at the end of a line. The second type of errors concerns grammatically correct code, that the compiler can successfully interpret, but that implements illogical or inconsistent commands. This is what happened to our upgraded model: the Lsd model program has been created, because the commands were correct, but during the actual simulation run Lsd realised that there is an inconsistency. Let's see firstly what a dead lock error is, then we analyse the information provided by Lsd to find the cause of the error, and, finally, we will fix our model.

The dead lock errors are cycles of computations that the computer is not able to resolve. They are the equivalent of asking a computer to solve the chicken-egg problem, with the difference that computers actually try to solve it. For example, consider a set of three variables with their equations, each using one of the other variables:

$$\begin{aligned}
X_t &= f_X(Y_t) \\
Y_t &= f_Y(Z_t) \\
Z_t &= f_Z(X_t)
\end{aligned}$$

One of the basic concepts in computing is that of subroutine. They are parts of code that execute specific operations, for example computing a number as elaboration of other values. If a subroutine requires a value which is still not available, the program

interrupt its current operation (remembering at which point it was interrupted and any intermediate result obtained so far) and executes the subroutine for the requested value. When the subroutine has finished, the initial operations can continue using the result of the subroutine

Let's see how this work with our example. Suppose to start from  $X_t$  (though the same applies starting from  $Y$  or  $Z$ ). The equation for  $X$  starts to be executed, but its complete computation requires the value of  $Y$ . Therefore, the computer interrupts the execution of  $f_X$  and begins computing the equation for  $Y_t$ . Also  $f_Y$  cannot be completed because it requires a value not yet available,  $Z_t$ , and therefore also the equation  $f_Y$  is interrupted and the computer begins to compute  $Z_t$  using its equation  $f_Z$ . But one of the values to be used in this latest equation is  $X_t$ , whose equation, though initiated, still did not provide the value for  $X_t$ . Therefore, following blindly the rules of computing, the processor would start to compute the equation  $f_X$ , which is interrupted in order to compute  $Y_t$ , etc. ...

In ancient operative systems a circular set of subroutines like this used to lock computers in that the processor initiated to compute each subroutine without being able to finish any computation, and refusing to accept commands from the keyboard (therefore the name). Modern operative systems avoid to lock computers, but the program entering in this sort of errors crashes without any notice, making impossible for the programmer to spot the faulty lines in the code. Instead, Lsd recognizes that a dead lock has occurred and interrupts the simulation providing all the information required to fix up the error.

Before continuing to analyse the Lsd tool kit for fixing dead lock errors, let's make a brief comment on dead lock errors. Simulation programs are not mathematical but computational logical structures, and this is nowhere clearer than in the case of dead locks. A mathematical model can well contain a set of equations as in the example above. It is interpreted as: *The set of value(s) of  $X$ ,  $Y$  and  $Z$  such that the three equations  $f_X$ ,  $f_Y$  and  $f_Z$  are satisfied.* The difference between computing and mathematics consists in the symbol “=”. In mathematics it is the condition such that the values on both sides of the equation are identical. In computing, instead, “=” means that the variable on the left of “=” must assume the value on the right. Therefore, for example, in mathematics you can never write  $X = X + 1$ , since there is no number equal to its subsequent. Instead this is perfectly normal in computing: the command assigns to  $X$  its own value plus one. Conversely, in mathematics you may write  $X + Y = 2$ , but it makes no sense in computing, since there is no variable on the left to which assign the value. Note that, in programming languages, you always have two different symbols for assigning values to variables (e.g. “=” in C++) and for testing the condition on whether two values are identical (“==” in C++).

So, the main problem with dead lock errors is to identify the chain of equations that, calling each other, caused the never ending cycle to occur. Let's see the information Lsd gives us to find out what type of error is and how to fix it. They are contained in the **Log** window of the Lsd model program. The messages you find consist in several lines providing information on the status of the model when the error occurred. The crucial lines are the last ones. In our case these lines are:

```
Level   Variable Label
3   RelativeQuality
2   Sales
1   WAverageRW
0   Lsd Simulation Manager
```

They indicate that the model has started the simulation (level 0); the system began the model trying to compute the variable *WAverageRW* (level 1). This computation was interrupted in order to obtain the required value for *Sales*, whose execution started at level 2. But also the equation for *Sales* needed to be interrupted in order to compute first the value of *RelativeQuality*. Up to here the system worked normally. In fact, at any time step Lsd tries to compute the new values for each and every variable in the model, starting from the ones in the top level objects. If their equations require updated values from variables not yet updated, then it interrupts the current computation in order to execute first the variables required<sup>34</sup>. In programming jargon, when a subroutine, like the equation for a variable, is interrupted in order to compute another subroutine, you say that it is “place on the stack”. The level indexes in the **Log** message concern the “stack levels” at which an equation is executed.

In our case, the error is caused by the fact that the equation for variable *RelativeQuality* requires the present value of *WAverageRW*, which cannot complete its computation. Here the system realized that a dead lock risked to be initiated and issued the error message, blocking the simulation.

Now we know what is the error in our model: the average quality indicator uses the value of sales which is a function of the relative quality, which, finally, requires the average quality. Since all these variables must be computed by their respective equations before being used, the system does not know how to solve the circularity of the commands contained in the equations’ code. It is up to us, as programmers, to find a solution, which consists in changing one of the equations involved using a lagged value for one of the variables. The variable to choose is not important from the computational viewpoint, but depends on the interpretation of the variables. The most logical option, in our example, may be to change the equation for *RelativeQuality* in order to make use of the past values of *RandomWalk* and *WAverageRW*. In this case, we tell the model that the relative quality of today (time t) is a function of the relative qualities of yesterday (t-1), inserting a lag in the response from quality (*RandomWalk*) to the relative quality.

Having understood the error, and found how to fix it, we need to change the equations’ code. Firstly, however, we need to kill the Lsd model program that, though the simulation is blocked, is still running. The program offers three options:

- **Abort:** pressing this button we kill the Lsd model program. We will use this button after we will understand where the problem is.
- **Analysis:** this button activates the **Analysis of Result** module, providing the results produced up to the time step before the one when the error was produced. This is useful to investigate problems emerging later during a simulation run. In this case, the error was produced at the first time step, so there are no data actually available.
- **Debug:** with this button we can investigate the status of the model when the error appeared. We can see all the copies of the variables and parameters, even the one not selected to be saved. We will see this functionality later.

In our case we don’t have data to analyse, because the simulation crashed at the very first time step. Also the **Debug** option is not useful, since the error does not depend on the values produced in the simulation. Therefore, we kill the program clicking on **Abort** and return to LMM in order to fix the equation for *RelativeQuality*.

---

<sup>34</sup>Each variables is tagged with the time when it was lastly computed. Therefore, it is frequent the case that one variable is firstly computed because requested by another equation. If, in the same time step, its value is requested again, its equation is not re-executed, but the formerly obtained value is re-used.

#### 4.4.20 Modelling Time: changing order of Lsd equations

What we need to do in order to fix our error is simply to switch the order in which the equations are executed within a time step: first relative quality, and then average quality. It requires few changes to the equations' file, that we will see in a moment. Firstly, however, it is worth reasoning on what we have discovered by means of the dead lock error, and what we are going to do to fix it.

There are two ways to see this: a mathematician, or a modeller used to standard analytical models, may see this as an annoying quirk required by the stupidity of computers, unable to solve even the simplest set of linear equations. Under this view, a dead lock error, and the way to solve it, is only a technical problem. The opposite way is to interpret the discovery of a dead lock error as an improvement of our knowledge of the modelled phenomenon. In fact, normally people represent to themselves a model by individual variables, and equations, neglecting the overall temporal or logical network linking the variables to form the overall model. Actually, this is the very reason for using simulations: I tell the computer how to compute X, Y, Z etc., and then observe what happens through time. The fact that a dead lock error occurred is a signal that your equations are wrong. How can your model be a reasonable approximation of real world events, if they cannot even logically take place?

Most of the times, dead lock errors point to missing conceptual elements of the model, which is well worth to analyse. For example, consider a model where a set of firms decide the quantity to produce as a function of the market price, and the market price is a function of the quantity produced. Besides the functional forms, the model is still not complete: I need to specify whether firms decide firstly their production as a function of past price, or if the price is computed first as a function of past quantities. Generally, I will obtain different results in the two cases, since they assumes different types of behaviour by consumers and producers. The error shows that a missing part of the model needs to be filled. This is an example of why simulations are a useful analytical tool: being forced to think how to implement a given phenomenon, forces modellers to devise rigorous algorithms, and therefore to have a precise idea on how the world really functions.

Let's go back to our problem: we need to update the equations for *RelativeQuality* in the following way:

```
EQUATION("RelativeQuality")
/*
Compute the relative quality
of a firm as the relative ratio
of the difference between RandomWalk
and AverageRW
*/

v[0]=VL("RandomWalk",1);
v[1]=VL("WAverageRW",1);

RESULT( (v[0]-v[1])/v[1] )
```

You should be able by now what the changes do: *RelativeQuality* is now computed as the relative difference of *RandomWalk* and *WAverageRW*, both with their past values:

$$RelativeQuality_t = \frac{RandomWalk_{t-1} - WAverageRW_{t-1}}{WAverageRW_{t-1}}$$

Now we can compile our model, load the configuration, and execute successfully a simulation run.

#### 4.4.21 Interpreting results

The simulation run, though safely executed, has produced a wild series of values, in which we can be easily get lost. Let's try to understand what has happened, by running a simulation without random variability.

In essence, our model contains random elements (*RandomWalk*) and a distribution element assigning *sales*. Let's see how the distribution element work. To do this, we should transform the *RandomWalk* variables in parameters, so to remain constant throughout a simulation run. However, we can obtain the same results assigning 0 to *minimum* and *maximum*. This would produce a constant value of 0 for all the *X1* variables at any time step:

$$X1_t = \text{minimum} + (\text{maximum} - \text{minimum}) * X_t = 0 + (0 - 0) * X_t = 0$$

and therefore leaving unchanged *RandomWalk* at the initial value for any time step:

$$\text{RandomWalk}_t = \text{RandomWalk}_{t-1} + 0 = \text{RandomWalk}_1$$

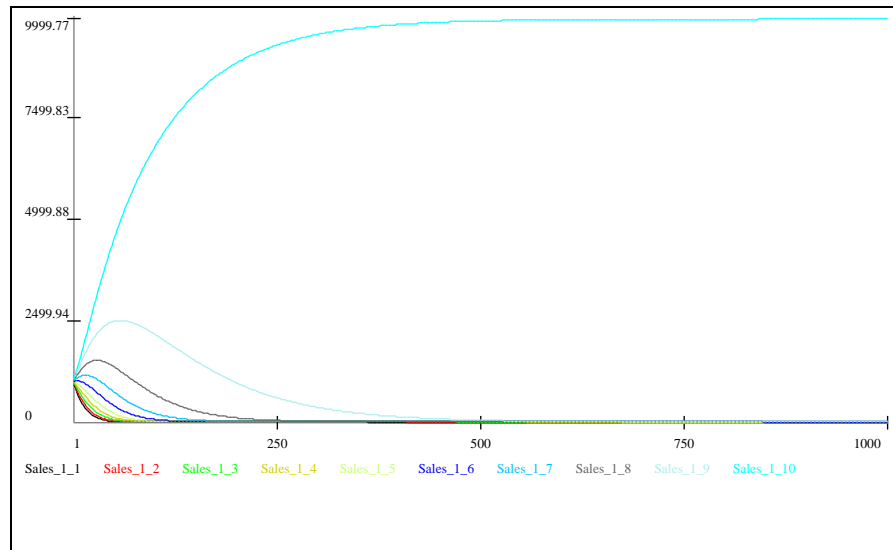


Figure 17: Sales time sequence with constant qualities.

Move the Lsd browser to show the objects *MyObj*. Open the initial values window (**Data/Init.values**) and use the **Set All** for:

- *minimum*, setting all of them equal to 0
- *maximum*, setting all of them equal to 0
- *RandomWalk* (-1), setting all of them to increasing values starting from 1000 with step of 100

All the *Sales* (-1) should remain to 1000. The above settings ensures that each *RandomWalk* is assigned a different value of 1000, 1100, etc. which will remain constant through a simulation run. The initial average quality must be set to 1450, because of the different *RandomWalk* values. Move the Lsd browser to show object *AggregateObject* and initialize *WAverageRW* (-1). Leave *alpha* to 0.2.

Save this configuration with a different name (say **Sim2**) in order to differentiate it from the other.

Now our model represents a group of firms of varying quality, and we are ready to test how their sales levels are affected. Run the simulation, and then open the **Analysis of Results** module. Select the sales variables and plot their time sequence values. The result should be like the graph shown in fig. 17. We can see that the 10<sup>th</sup> firm gains in the end all the sales, while the others decrease to 0. This is obvious because the 10<sup>th</sup> firm has the highest quality. The interesting aspect of the simulation consists in the patterns of the different sales' variables. The worst ones (e.g. 1, 2 etc.) always decrease their level, while the firms from 5 to 9 initially increase their sales levels that later falls down to zero. Why is this happening? In order to solve this question we will make use of a new functionality offered by the Lsd model programs, the Lsd Debugger.

#### 4.4.22 Lsd Debugger

In many cases we have a simulation model producing (more or less sensible) results that are difficult to trace back to some equation in the model. Moreover, the results can expose errors that we don't know where they originated. The Analysis of Results can show what has happened, but we may find ourselves wondering *why* those series have been produced.

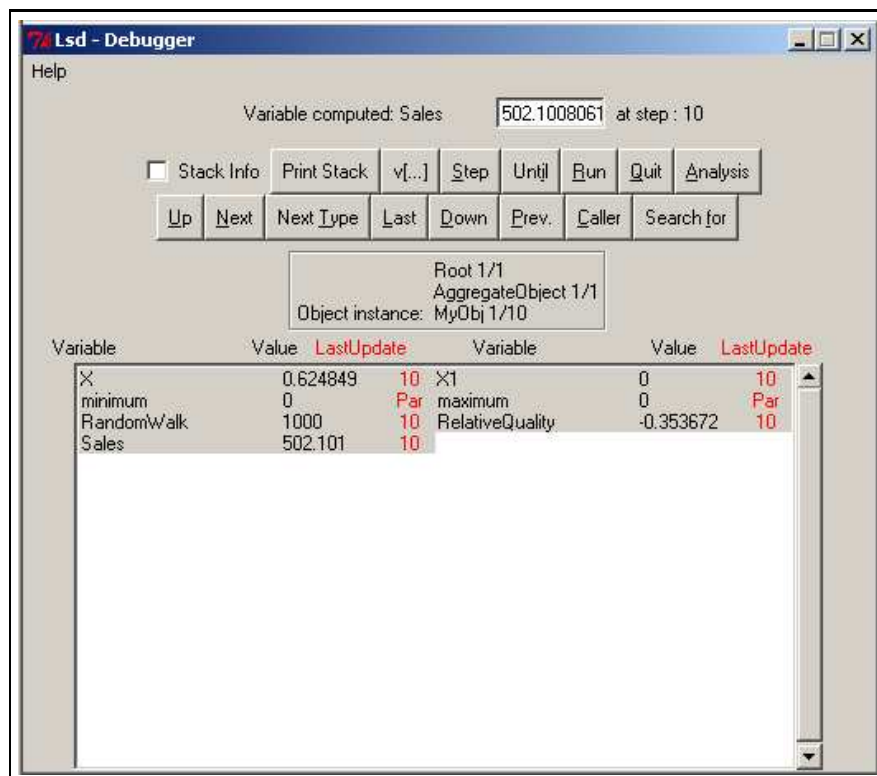


Figure 18: Debugger set on the variable *Sales* at the 34<sup>th</sup> time step.

The Lsd model programs are endowed with a module that permits to interrupt a simulation run at any moment and investigate the status of each and every element of the model, the Lsd Debugger<sup>35</sup>. Let's begin to quit the Analysis of Results (**Exit/Exit** and re-load a fresh configuration (press the keys Control+w).

<sup>35</sup>The name is due to the use of this function for spotting errors, that is bugs, produced in a simulation, that you want to fix, de-bugging the model.

In order to interrupt a simulation run we need to tell the model two pieces of information: at what time step we want the interruption to occur, and, within a time step, which equation's computation we want to observe. Open menu **Run/Sim.Setting** and write in the field **Insert Debugger at:** a time step, say 10. After having pressed **Ok** move the Lsd Browser to show the content of *MyObj*; double-click on *Sales* and check on the option **Debug: ...** in the resulting option window for this variable, and, finally, press **Continue**. Now we can run the simulation as usual with **Run/Run**. The Lsd model program will compute the first 10 steps, and then it will stop at the 10<sup>th</sup> showing the window reported in figure 18.

This window provides a large number of possibilities to observe the model and, if necessary, also to modify it in the middle of the simulation run. We will see only a few of these now, but check the menu **Help/Lsd Debugger Help** for a complete presentation. Let's see firstly the items contained in the page.

The top part of the figure (shown in figure 19) contains the header of the debugger, which contains the name of the variable just computed, the value resulting from its computation, and the time step of the simulation.

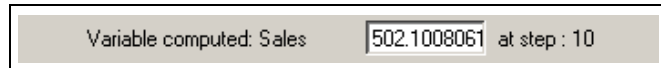


Figure 19: Debugger header: name of the variable, value and time step.

Below the header there is a row of buttons used to control the simulation run. These buttons allow operate and provide information on the dynamics of the model. For example, button **Run** would tell the model to continue the simulation until the end, **Quit** aborts the simulation at current time step. For the moment don't use any of these buttons.

The second row of buttons allows users to browse through the different copies of the objects forming the model. For example, try to press the key "u" (for "Up") and "d" (for "down")<sup>36</sup>. The lower half of the window will change each time showing the current content of the only copy of object *AggregateObject* and the first copy of *MyObj*. The number of the copy and the label of the object shown by the debugger is indicated in figure 20 (to reach this press 3 times the button **Next**. This label shows the text **Object instance:** on the left the currently shown copy (or instance). Moreover, it shows also all the labels (and instance indexes) of the objects "above" the shown one, beginning with *Root*.

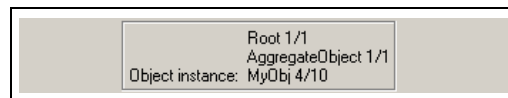


Figure 20: Debugger's label of the object's copy currently shown (the 4<sup>th</sup> copy of *MyObj* in a group of 10).

When a copy of an object is shown the debugger lists all the variables and parameters contained in the object, indicating their current value. Note that the values of variables are tagged with a number, printed in red. This number is the time step when the variable has been updated last time. Typically, when the debugger interrupts a simulation run in the middle of a time step, some of the variables have already been computed, while others still need to receive their newly computed value. Comparing the value **LastUpdate**

<sup>36</sup>The underlined letters in the buttons' text indicate the keys available as alternative to click on the buttons.

for a variable with the current time step of the simulation shown in the debugger’s header you can see whether the variable has been already computed or not. In our case, all the variables copies with **LastUpdate** set to 9 still needs to be computed in the current time step (10).

So, now we can answer our question: why some copy of *Sales* decrease while others, non-optimal, still increase their level? Let’s explore the set of objects *MyObj* (pressing repeatedly “n”). We see that all the values for *RelativeQuality* are negative, for the copies from 1 to 6 included. We know, from the equations, that a negative value is caused by *RandomWalk* smaller than *WAverageQuality*. Given our set up (*minimum* and *maximum* set to 0), variables *RandomWalk* never change. Actually, it is *WAverageQuality* that does change. At time 9 this variable has the value of 1547.2, as we can observe moving the debugger in *AggregateObject* pressing **Up**. The change is due to the changes in the weights of the average: while higher quality firm’s sales increases, so does its weight in the average, pushing up the value of *WAverageQuality*. Therefore, the values of quality for some firms was above average in the beginning, and therefore their sales level increased. But later *WAverageQuality* increased so to overcome the levels of quality for each firm but the 10<sup>th</sup>, so that eventually all these firms have decreasing levels of sales.

## 5 Lsd language for equations

Modellers express their equations as chunks of C++ and Lsd code producing a numerical value<sup>37</sup> which is attached, during a simulation run, to one copy of the variable which the equation refers to. Equations must be thought of as equations in a difference equation model, where each variable’s computation is expressed as the elaboration of present and past values of variables, and the parameters of the model. If the model includes more than one copy of a variable, the equation is re-computed, using the appropriate parameters, for each copy.

Two facilities permit an extremely simple code-writing:

1. Each variable is contained in an object, and any object is potentially replicated in many copies. The same code is applied to every copy of the variable, but all the Lsd functions concerning operations on the model (e.g. request the value of a parameter) are activated by an object. During the writing of an equation the modeller has available the specific copy of the object whose equation is executed, so that the Lsd operations provide different results as a consequence of being applied from different objects.
2. The equations are computed once and only once at each time step of the simulation for each copy of the variables, so that they provide a result tagged with the time it was computed. Modellers don’t need to think about the order of execution of equations, since this is automatically determined by the system<sup>38</sup>.

The overall philosophy of the Lsd/C++ language for equations is that modellers express in a very simple way the most obvious and frequently used operations. For example, it is possible to request a value of a parameter or variable without specifying anything else

---

<sup>37</sup>Lsd limits to consider only real valued numbers (double precision floating point).

<sup>38</sup>In case of inconsistencies the system issues a warning providing information to fix the error. For example, if the equation for  $X_t$  requires the value of  $Y_t$  and the equation for  $Y_t$  requires the value of  $X_t$ , then the system doesn’t which equation must be computed first, and therefore interrupts the simulation inviting the modeller to set one of the variable in one of the equation with a lagged value.

than the label of the element required. Lsd will interpret this as requesting the value at time  $t$  for the computation of the variable at the same time step.

However, it is possible to overrule the default automatic system using one of the many Lsd functions that permit to access, read and modify any aspect of the model, like, for example, overwriting values of parameters, creating or destroying objects. Before looking at these functions, in the next paragraph we present the list of system (i.e. C++ variables available when writing the code for the equations).

## 5.1 Macro language an raw C++/Lsd coding

Lsd is a C++ based language, so that every C++ expression is also a legal Lsd expression. Moreover, Lsd provides a set of C++ implemented functions in order to easily access the model content. The Lsd functions can be used by the modellers in two forms: their raw C++ expression, or a simplified macro version. There is absolutely no computational difference between the two versions, since, in fact, the macro expressions are translated in their raw C++ meaning before being processed by the compiler of the Lsd model program. Therefore, an equation can easily contain at the same time some macro expressions and some raw C++ ones. The difference is that the macro version are much simpler to use, since they are shorter than their C++ equivalent.

The macro language has been recently introduced in Lsd (since ver. 5.0, in April 2001), and therefore some of the older example models are still expressed with raw C++ expressions only. In the text of this document we refer only to the macro version of the Lsd functions, while the titles of the paragraphs devoted to them report also the raw C++ expressions.

## 5.2 C++ basic grammar for Lsd coding

The code for the equations must be legal C++ code, extended with the Lsd functions described below. The code is composed by lines of commands that the computer generally executes sequentially, moving to the next line when the previous one has been completed.

Any line of code must respect the C++ rule of terminating with a semi-colon “;”, unless the line is a multi-column Lsd command like EQUATION, or C++ command, like `if(condition)`.

### 5.2.1 Comments

The code can include comments, that is text that is ignored by the compiler and serves to describe the code to readers. Comments in C++ comes in two forms:

```
/*  
  
This is a multi line comment, continuing until  
a sequence "star slash" is encountered  
  
*/  
  
//this is a single line comment,  
//terminating at the end of the line
```

### 5.2.2 Assignments, arithmetic operations and increments

If `a` is a variable, then the programmer can assign a value with the command “=”:

```
a=4.3;
```

Any assignment must be terminated with a semi-colon “;”.

It is also possible to assign values from other variables, and use the standard mathematical operations, using the parentheses to group relevant priorities:

```
a=b+3-d/(e+g)*(h+i);
```

The above command expresses in C++ the formula:

$$a = b + 3 - \frac{d}{e + g} * (h + i)$$

Less obviously, it is also possible to use the same variable on both sides of the assignment:

```
a=a+32;
```

The above line assigns `a` with its previous value increased of 32. Therefore, if `a` had the value of, say, 5, after the above line it is assigned the value of 37.

These commands incrementing the value of a variable are so common that they have also a short way to express them. For example, the command `a=a+32;` can be expressed also with the command `a+=32`, saving the expression of `a` in the left part of the assignment. This short expression can even be used for the other arithmetical operations:

```
a=a+32; ⇔ a+=32;
```

```
a=a/32; ⇔ a/=32;
```

```
a=a*32; ⇔ a*=32;
```

```
a=a-32; ⇔ a-=32;
```

Note the difference that exists between the equal sign “=” used as assignment (i.e. load the value on the left to the symbol on the right), from the logical condition of identity (are the right and left sides identical?). Computer languages must use two different symbols for the two operations. In the following paragraph on the `if` conditional statement we will see the symbol used for the logical identity.

A C peculiar command (`++`) allows both to increase of 1 a variable and to use it as assignment. The command `++` (and its sister command `--`) works differently depending on whether it is used after or before a variable symbol. If it is used after, the command firstly assign the current value, and then increases it of 1. Instead, if `++` is used before a variable, it firstly increases its value, and then assigns this. For example :

```
a=3;
```

```
b=a++;
```

```
c=++a;
```

At the end of the above commands, we will see that `a` equals 5 (3 and two increments), `b` equals 3 (because `b=a++` firstly assigns the value of `a` to `b`, and then increments `a`), and `c` equals 5, because `a` increased from 4 to 5 before being assigned to `c`.

### 5.2.3 if ... then ... else

As said before, lines of code are executed sequentially, but there are exceptions. A frequently used exception is the conditional execution of different blocks of line depending on whether a condition is verified as true or not. The grammar of the conditional command is the following:

```

if(condition)
{
    /*
    insert here any line you want to be executed
    if "condition" is true
    */
}
else
{
    /*
    insert here any line you want to be executed
    if "condition" is false
    */
}

```

The curly brackets “{}” can be skipped if there is only one line to be executed conditionally.

The `condition` is normally based on one of the following binary relations, assuming `a` and `b` to be two numerical values or variables containing numerical values:

- **Equal** `a==b`: the condition is true if `a` equals `b`.
- **Larger** `a>b`: the condition is true if `a` is larger than `b`.
- **Larger or equal** `a>=b`: the condition is true if `a` is larger than or equal to `b`.
- **Smaller** `a<b`: the condition is true if `a` is smaller than `b`.
- **Smaller or equal** `a<=b`: the condition is true if `a` is smaller than or equal to `b`.

The `condition` can be composed with the logical operators of negation, logical “and”, logical “or”:

- **Negation** “!”: the `!condition` is true if `condition` is not true and viceversa
- **Logical “and”** `&&`: given two conditions `cond1` and `cond2`, the condition `(cond1 && cond2)` is true only if both `cond1` and `cond2` are true, in the three other cases (one or both false) is false.
- **Logical “or”** `||`: given two conditions `cond1` and `cond2`, the condition `(cond1 || cond2)` is true if at least one `cond1` or `cond2` is true, otherwise is false.

For example, the following code distinguishes three possible situations:

1. `a` equals `b` and `c` does not equal `d`
2. `a` equals `b` and `c` equals `d`
3. `a` does not equal `b` (irrespective of `c` and `d`)

```

if(a==b && !(c==d))
{
    /*
    "a" equals "b" and "c" is different from "d"
    */
}

```

```

else
  { // if the process is here, than one of the two conditions above is not true
    if(a==b)
      { /*
        "c" equals "d"
        */
      }
    else
      { /*
        "a" is not equal to "b", but "c" may or may not equal "d"
        */
      }
    }
}

```

Conditions in C++ are just integer numbers, where “0” (zero) means false, and any non-zero integer means true.

#### 5.2.4 Use of cycle for

A very frequent command allows to repeat a block of lines again and again until a specific condition is satisfied. The grammar for the cycle for is:

```

for(INIT ; CONDITION ; ENDCYCLE)

```

followed by the block of code to be repeated contained between curly brackets “{” and “}”.

The for command executes the following steps:

1. Execute INIT;
2. control CONDITION. If it is true...
  - (a) execute the block of code
  - (b) execute the commands contain in the field ENDCYCLE
  - (c) return to 2
3. ... else exit because CONDITION is false

Example Repeat a block of code 5 times setting a variable from -2 to 2:

```

for(i=-2; i<3 ; i++)
{
  /*
  in the execution of these lines i
  assumes values -2, -1, 0, 1 and 2
  */
}
//here i equals 3

```

### 5.3 Equation and Function

The code for a variable can be expressed either as “equation” or “function”. By far the most frequent case is that of equations, although in some cases using functions can be more elegant and fast. The difference between the two can be referred to the case where the variable needs a time tag (equation) or does not need it (function). Let’s see the difference

in more detail. Non expert programmers should stick to use only equations, and skip this paragraph until they develop some experience with Lsd programming.

The difference between the two forms of coding a variable is that equations are computed once and only once at each time step. If the value of the variable computed with an equation is requested twice or more times during the same time step, the equation is executed only once, and the other times the same value is returned. Of course, a different value is computed for each copy of the variable.

Instead, functions are re-computed each and every time the value of the variable is requested, even many times within the same time step, always producing a newly computed value.

As example of an equation, consider a model where variable *Price* is determined on the market. Each of many firms uses *Price* (say to determine the profits), so that each time step the value of *Price* is requested many times. Of course, the modeller wants that, at the same time step, an identical value of *Price* is used by all firms.

As example of a function, consider a model where firms can innovate the technology randomly, with a probability determined by several factors, like the total investment in R&D of the industry and its own investment. The modeller may desired to create one single variable computing the probability to innovate for any firm and returning the failure or success of the innovation. Of course, the code must be re-computed every time a firm tries the innovation, even within the same time step, since it makes no sense using an identical result for all the firms.

Practically, the difference between an equation and a function resides only in the way the code for a variable is expressed, while there is no difference in the Lsd model structure. The header for an equation is expressed as:

```
EQUATION("Var")
```

while the header for a function is:

```
FUNCTION("Var")
```

Modellers have available exactly the same Lsd functions for equations and functions, but there is a potential error that needs to be treated carefully. Functions' code is executed only if another variable requires the value of the function's variable. Instead, equation's code is computed at each time step, even if the value is never used by other variables in the model. As we will see in the next section, modeller have access to the "caller" object, that is, the object containing the variable that requested a variable to be computed. In case of equations, it is possible that the caller object does not exist, because it is the system that requested the variable to be updated. Therefore, the use of the "caller" object should be protected against this possibility.

#### 5.4 System variables available for equations' writing

The system provides modellers with C++ variables to be used in the code to work on the model structure. There are three types of these variables:

1. **Variable specific:** during the computation of an equation the modeller knows the relevant information concerning the specific copy of the variable being computed. By far, the most relevant element of these variables is the object containing the variable which is computed.

2. **Temporary local variables:** these C++ variables can be used to store intermediate results before completing the equation. The values stored in a temporary variable cannot be transferred from one equation to another.
3. **Global variables:** these C++ variables store globally relevant data, for example `t` is a variable storing the current time step of the simulation.

Table 1, pg. 55, illustrates in detail all the C++ variables available.

## 5.5 Lsd object's

This paragraph describes some of the technical implementation of Lsd, in order to allow expert users to optimize their models. Non-interested readers can skip the rest of the paragraph.

The definition of model in Lsd is based on Lsd object's. The `object` is the basic component of the model, and the structure of `object's` makes up the model. The `object` should though of as a container that is linked to other `object's` in the model and contains a set of variables and parameters. The links of an `object` (call it `obj`) are the following:

- `obj->up`: is the link to the `object` containing `obj`. Any model structure begins with the `object` called *Root*, which is the only `object` for which the link `obj->up` is empty (technically equals NULL).
- `obj->next`: it is the link to the next element of the group of `object` to which `obj` is part of. `obj->next` can assume three different forms:
  1. `obj->next` equals NULL: the `obj->next` is empty, implying that `obj` is the last element of its group;
  2. `obj->next` links to an `object` of the same type of `obj` (i.e. same name, same set of variables, parameters and descending `object's`);
  3. `obj->next` links to an `object` of a different type of `obj`. That is, `obj` is the last element of its group, but then there is another group of `object's` descending from the same one.
- `obj->son`: is the link to the first `object` descending from `obj`. If `obj->son` is empty, then `obj` contains no `object's`, or descendants.

Any `object` used in the equation, being one of the readily available like the `object` under computation `p` or another stored in the temporary variable for `object` `cur`, could provide access to their linked objects: `up`, `next` and `son`.

However, this is never necessary, and is useful only to speed up the computation of particularly large models. Only expert users should make use of them.

<b>Computed variable's specific elements</b>	
<code>p</code>	Object containing of the variable under computation (we refer to this also as “currently computed object”).
<code>c</code>	Object containing the variable whose equation triggered the computation of the current one. If the computation is simply requested by the system, <code>c</code> equals the conventional value <code>NULL</code> , and any operation with it will cause the simulation to abort.
<code>CURRENT</code>	Latest value computed for the variable under computation, irrespective of the time step it was computed
<code>lastupdate</code>	Latest time step in which the variable under computation has been updated.
<b>Temporary elements available to store intermediate results</b>	
<code>v[0], v[1], ...</code>	temporary variables for values to be assigned and used within in an equation
<code>cur, cur1, ...</code>	temporary variables for objects to be assigned and used within in an equation <sup>39</sup> .
<i>userdefined</i>	Modellers can define their own temporary variables, if necessary, according to the usual grammar of C++, declaring a variable after the line <code>MODELBEGIN</code> . For example, the line <code>double myvar;</code> creates a temporary variable for real value. The variable exists only until the end of the equation.
<b>Global variables</b>	
<code>t</code>	Current time step
<code>maxstep</code>	Number of steps for the simulation
<code>quit</code>	Normally set to 0 while the simulation is running. It is set to 1 when the simulation must finish normally and is set to 2 when the simulation aborts for an error
<i>userdefined</i>	Modellers can define their own global variables, if necessary, according to the usual grammar of C++, declaring a variable before the line <code>MODELBEGIN</code> . For example, the line <code>object *myobj;</code> creates a global variable for object. The variable exists throughout the entire simulation run, so that it can be initialized and used in every equation.

Table 1: List of system variables available to implement an equation

## 5.6 Lsd and C++ Commands for Lsd equations

This section describes all the Lsd functions available for coding the equations, together with a short description of the most commonly used C++ commands.

The typing and naming conventions used to describe the functions are the following:

- "Label": refer to a string of characters, normally used to express the label of a parameter, variable or object, to be inserted in a Lsd function;
- *Label*: refer to the Lsd name of a variable, parameter or object;
- *t*: it is the time step during which the equation is computed;
- *lag*: a positive integer value, referring to the lag in respect of *t*;
- object under computation: this expression refer to the object *p* containing the variable whose equation is computed;
- *obj*: a general variable containing an object (may be *p*, or a temporary variable for object *cur*);
- *value*: a real valued number;

### 5.6.1 V("X") or p->cal(X,0)

Return the value of *X* at time *t*, choosing the copy "closest" (see the example below) to the currently computed copy of the variable. There are three other members of the family:

- VL("X",*lag*) or p->cal("X",*lag*): return the value of *X* closest to the currently computed objects at time *t-lag*;
- VS(*obj*, "X") or obj->cal("X",0): return the value of *X* closest to the object *obj* at time *t*;
- VLS(*obj*, "X", *lag*) or obj->cal("X",*lag*): return the value of *X* closest to the object *obj* at time *t-lag*;

**Example** This is obviously the most used Lsd function. This function is implemented in such a way to always return the intuitively obvious value, that is, to select the correct copy of the value requested choosing among possibly many copies. For example, imagine that we have an object *Industry* containing variable *Price*, and many descending objects *Firm*'s, which, in turn, contain variables *Quantity* and *Revenues*. The equation for *Revenues* can be written as:

```
EQUATION("Revenues")
/*
Compute the Revenues as the product of price (in Industry) and
Quantity (in Firm)
*/

RESULT(V("Quantity")*V("Price") )
```

As we have already mentioned, the code for the equation is replicated identically for all the copies of the object *Firm* in the model every time it must compute the value of *Revenues*. Each time, the code will use a different value of *Quantity*, using the copy of the

variable contained in the same copy of the object that contains also the copy of *Revenues* which is being computed.

Moreover, the code expressing the value of *Price* does not differ from that for *Quantity*, although in the first case Lsd must access the object *Industry* while in the second it needs the object *Firm*. Similarly to what said above, the equation behaves “correctly” even in case the model includes more than one copy of *Industry*, and therefore many groups of objects *Firm*. In this latter case, the value of *Price* used in each execution of the equation will refer to the copy of *Industry* containing the copy of the object being computed<sup>40</sup>.

The function  $V(\dots)$  is applied within an equation; which is computed in behalf of a specific copy of a variable; which is contained in an object. Let’s call this object the *currently computed object*, although it is a bit imprecise since it is one of its variables to be computed. When  $V(\dots)$  is executed Lsd starts a search in the model looking for an object containing the desired variable or parameter. The search starts from the currently computed object and can continue, if necessary, up to the exploration of all the objects in the model. Given an object where to search an element, say `obj`, the following rules are used (marked with mnemonic labels):

1. **Search here:** search among the variables and parameter of `obj`.
2. **Search down:** search among the variables and parameter of the objects descending (i.e. contained in) `obj`.
3. **Search up:** search among the variables and parameter of the object parents of (i.e. containing) `obj`.

The rules start to be applied from the currently computed object and are used recursively to search, for example, also the objects contained in the objects of the currently computed object. The search terminates either with one copy of the object found (and in this case the required value is returned), or, if no such an element is found, an error message is produced and the simulation aborts. Typically, this is the case when the modeller spells by mistake in different way an element in the equation file and in the model structure. Or simply has forgot to add an element used in a new equation to the model structure.

The  $VS(obj, "X")$  and  $VLS(obj, "X", lag)$  members of the same family of  $V("X")$  are different in that, instead of starting the search of the object from the currently computed object, the modeller can specify in the `obj` field another one.

Almost all Lsd functions requiring to read values of variables and parameters make use of the same searching strategy described here.

### 5.6.2 $V\_CHEAT("X", fake\_caller)$ or $p \rightarrow cal(fake\_caller, "X", 0)$

(Very rarely used function, and only for reasons of efficiency. Only expert Lsd users should use it.) Return the value of  $X$  at time  $t$  closest to the object under computation. The equation for  $X$  will use `fake_caller` as if it were the object that requested its computation.

Normally, when the equation for variable  $Y$  is requested because another equation needs its value, then the equation for  $Y$  can access the object containing the copy of  $Y$  which triggered its computation (see object `c`). Instead, this function allows the equation for the triggered equation to be “cheated”, in thinking another object actually requested its value.

The other members of the family are:

---

<sup>40</sup>That is, the object whose copy of *Revenues* is computed.

- `VL_CHEAT("X", lag, fake_caller)` or `p->cal(fake_caller, "X", lag)`: Return the value of  $X$  at time  $t$ -lag closest to the object under computation. The equation for  $X$  will use `fake_caller` as if it were the object that requested its computation.
- `VS_CHEAT(obj1, "X", fake_caller)` or `obj1->cal(fake_caller, "X", 0)`: Return the value of  $X$  at time  $t$  closest to the object `obj1`. The equation for  $X$  will use `fake_caller` as if it were the object that requested its computation.
- `VLS_CHEAT(obj1, "X", lag, fake_caller)` or `obj1->cal(fake_caller, "X", lag)`: Return the value of  $X$  at time  $t$ -lag closest to the object `obj1`. The equation for  $X$  will use `fake_caller` as if it were the object that requested its computation.

This function is very rarely used, and almost always there are better ways to implement what this function does. Only expert users should use it, generally for reasons of computational efficiency.

### 5.6.3 `SUM("X")` or `p->sum("X", 0)`

Return the sum of  $X$ 's computed at time  $t$  contained in a group of objects descending from the object under computation.

The other members of the family are:

- `SUML("X", lag)` or `p->sum("X", lag)`: Return the sum of  $X$ 's computed at time  $t$ -lag contained in a group of objects descending from the object under computation.
- `SUMS(obj, "X")` or `obj->sum("X", 0)`: Return the sum of  $X$ 's computed at time  $t$  contained in a group of objects descending from `obj`.
- `SUMLS(obj, "X", lag)` or `obj->sum("X", lag)`: Return the sum of  $X$ 's computed at time  $t$ -lag contained in a group of objects descending from `obj`.

**Example:** Suppose that the variable *TotalProduction* is contained in an object *Industry* that contains several objects *Firm* with a variable called *Production*. The equation for *TotalProduction* is:

```
EQUATION("TotalProduction")
/*
Sum of the variables Production in all the descending objects
*/

RESULT(SUM(Production))
```

### 5.6.4 `STAT(X)` or `p->stat(X, v)`

Compute descriptive statistics for  $X$  at time  $t$ , supposedly a variable or parameter contained in a set of objects descending from the one under computation.

The function stores the result of the statistics in the vector of temporary variables `v` as follows:

- `v[0]`=number of elements;
- `v[1]`=average
- `v[2]`=variance

- `v[3]`=maximum value
- `v[4]`=minimum value

The function can also be used as:

- `STATS(obj, "X")` or `obj->stat("X", v)`: Compute descriptive statistics for  $X$  at time  $t$ , supposedly a variable or parameter contained in a set of objects descending from `obj`.

**Example:** Suppose that the variable *TotalProduction* and parameters *AverageProduction* and *MaximumProduction* are contained in an object *Industry* that contains several objects *Firm* with a variable called *Production*. The equation for *TotalProduction* is:

```
EQUATION("TotalProduction")
/*
Sum of the variables Production in all the descending
objects. Moreover, it also stores the average and maximum production
*/
STAT("Production");
WRITE("AverageProduction",v[1]);
WRITE("MaximumProduction",v[3]);

RESULT(v[1]*v[0])
```

### 5.6.5 WHTAVE("X","W") or `p->whg_av("W","X",0)`

Return the average of  $X$  weighted with the values of  $W$ , both computed at time  $t$  and defined as two elements located in a group of objects descending from the currently computed one.

Members of the same family are:

- `WHTAVEL("X","W",lag)` or `p->whg_av("W","X",lag)`: Return the average of  $X$  weighted with the values of  $W$ , both computed at time  $t$ -lag and defined as two elements located in a group of objects descending from the currently computed one.
- `WHTAVES(obj, "X","W")` or `obj->whg_av("W","X",0)`: Return the average of  $X$  weighted with the values of  $W$ , both computed at time  $t$  and defined as two elements located in a group of objects descending from `obj`
- `WHTAVELS(obj, "X","W", lag)` or `obj->whg_av("W","X",lag)`: Return the average of  $X$  weighted with the values of  $W$ , both computed at time  $t$ -lag and defined as two elements located in a group of objects descending from `obj`

**Example:** suppose to have a group of objects *Firms* containing two elements, *Productivity* and *MarketShare*. A variable contained above this group can contain the following equation:

```
EQUATION("AverageProductivity")
/*
Average Productivity, computed as the average of Productivity
weighted with MarketShare.
*/

RESULT( WHTAVE("Productivity","MarketShare") )
```

### 5.6.6 MAX("X") or p->overall\_max("X",0)

Return the maximum value of  $X$  at time  $t$  over a group of objects descending from the currently computed one.

Members of the same family are:

- MAXL("X", lag) or p->overall\_max("X",lag): Return the maximum value of  $X$  at time  $t$ -lag over a group of objects descending from the currently computed one.
- MAXS(obj, "X", lag) or obj->overall\_max("X",lag): Return the maximum value of  $X$  at time  $t$  over a group of objects descending from obj.
- MAXLS(obj, "X", lag) or obj->overall\_max("X",lag): Return the maximum value of  $X$  at time  $t$ -lag over a group of objects descending from obj.

**Example:** suppose to have a group of objects *Firms* containing variable *Productivity*. The variable *MaximumProductivity*, contained in an object above this group can be computed with following equation:

```
EQUATION("MaximumProductivity")
/*
Maximum value of Productivity among all the set of descending objects.
*/

RESULT( MAX("Productivity") )
```

### 5.6.7 Mathematical and probabilistic functions

When writing simulation models it is frequent the case of needing random events. In the next few line it is explained the overall concept “pseudo-random” number generator. Uninterested readers can skip this part, and see the list of available random functions. Remember that, being standard C++, it is always possible to link a Lsd model to a library of external functions.

Computers cannot flip coins, or choose otherwise events in a really random way. But there are many programs that produce series of values that appear as if they were drawn from random functions. That is, if you make any test on these series, they pass, for example, a test of normality. However, being in reality a deterministic series, it can be reproduced exactly any time as necessary. Programmers control the generation of random numbers using the “seed”. Before using a random series, it is necessary to initialise the program using a seed value. Two series of random numbers produced with the same seed value are exactly identical, while series produced with different seeds have no relation. Therefore, before running a simulation, you need to assign a seed value. Replicating a simulation with the same seed generates exactly the same (pseudo-)random events, while replicating the simulation with different seeds generates totally different random events.

The list of mathematical and probabilistic functions available to insert in an equation is the following.

- abs(a): return the absolute value of a;
- min(a,b): return the minimum between a and b;
- max(a,b): return the maximum between a and b;
- round(a): return the integer closest to the real value a;
- exp(a): return the exponential of a;
- log(a): return the natural log of a;

- `sqrt(a)`: return the square root of `a`;
- `pow(a,b)`: return the power `b` of `a`;
- `RND`: return a value drawn from uniform random function between 0 and 1;
- `UNIFORM(min, max)`: This function (actually is a macro) produces a random uniform value in the interval `[min,max]`.
- `rnd_integer(min, max)`: return a random integer value in the interval `[min, max]` with uniform probability
- `norm(mean,dev)`: return a random value drawn from a normal random function with mean `mean` and deviation `dev`;
- `poisson(mean)`: return a draw from a poisson random function with mean `mean`;
- `gamma(mean)`: return a draw from a gamma random function with mean `mean`;

Many other functions are available, like, for example, all the trigonometric functions, as part of the C++ mathematical library.

### 5.6.8 `WRITE("X",value)` or `p->write("X",value,0)`

Overwrite the element  $X$  contained in the same object containing the equation's variable, setting the time of update of  $X$  to 0 (conventional time indicator for parameters). The function does not return any value.

Another form of this function is:

- `WRITEL("X",value, time)` or `p->write("X",value,time)` : Overwrite the element  $X$  contained in the same object containing the equation's variable, setting the time of update of  $X$  to `time`.

**Example:** (see the example for function `STAT` in paragraph 5.6.4 pg. 58)

### 5.6.9 `INCR("X",value)` or `p->increment("X",value)`

Overwrite the value of  $X$  with the value `X+value`, using the  $X$  closer to the currently computed object, the most recently updated value of  $X$  and not modifying the indicator of time of updating (`lastupdate`) of  $X$ . The function returns the new value of  $X$  after the increment.

Another form of this function is:

- `INCRS(obj, "X", value)` or `obj->increment("X",value)`: Overwrite the value of  $X$  with the value `X+value`, using the  $X$  closer to the object `obj`, the most recently updated value of  $X$  and not modifying the indicator of time of updating (`lastupdate`) of  $X$ . The function returns the new value of  $X$  after the increment.

**Example:** Suppose to have the equation in object *Firm* containing variable  $A$  for productivity, variable  $Q$  for production, parameter  $K$  for capital and variable  $I$  for investment. Then, the equation for  $Q$  could be written as:

```
EQUATION("Q")
/*
Compute the quantity produced as the product of K (increased of investment)
and A.
*/
v[0]=V("I");
v[1]=INCR("K",v[0]);
v[2]=V("A");
RESULT( v[2]*v[1] )
```

### 5.6.10 MULT("X",value) or p->multiply("X",value)

Overwrite the value of  $X$  with the value  $X*value$ , using the  $X$  closer to the currently computed object, the most recently updated value of  $X$  and not modifying the indicator of time of updating (`lastupdate`) of  $X$ . The function returns the new value of  $X$  after the increment.

Another form of this function is:

- `MULTS(obj, "X", value)` or `obj->multiply("X", value)`: Overwrite the value of  $X$  with the value  $X*value$ , using the  $X$  closer to the object `obj`, the most recently updated value of  $X$  and not modifying the indicator of time of updating (`lastupdate`) of  $X$ . The function returns the new value of  $X$  after the increment.

**Example:** Suppose to have the equation in object *Firm* containing variable  $A$  for productivity, variable  $Q$  for production, parameter  $K$  for capital, parameter *alpha* expressing the percentage of  $K$  lost because of its use, and variable  $I$  for investment. Then, the equation for  $Q$  could be written as:

```
EQUATION("Q")
/*
Compute the quantity produced as the product of K (increased of investment and
reduced for consumption) and A.
*/
v[0]=V("I");
v[1]=V("alpha");
MULT("K", (1-v[1]) );
v[2]=INCR("K",v[0]);
v[3]=V("A");
RESULT( v[2]*v[3] )
```

### 5.6.11 SEARCH("X") or p->search("X")

Return the first object  $X$  found descending from the currently computed one.

Another form of the function is:

- `SEARCHS(obj, "X")` or `obj->search("X")`: Return the first object  $X$  found descending from `obj`.

### 5.6.12 SEARCH\_CND("X",value) or p->search\_var\_cond("X",value,0)

Return the first instance of an object found starting a search from the object under computation (with the strategy described in Lsd function `V("...")`, paragraph 5.6.1, pg. 56) conditional to contain element  $X$  with value `value` as computed at time `t`.

Other members of the same family are:

- `SEARCH_CNDL("X",value, lag)` or `p->search_var_cond("X",value,lag)`: return the first instance of an object found starting a search from the object under computation and conditional to contain element  $X$  with value `value` as computed at time `t-lag`.
- `SEARCH_CNDS(obj, "X",value)` or `obj->search_var_cond("X",value,0)`: return the first instance of an object found starting a search from `obj` and conditional to contain element  $X$  with value `value` as computed at time `t`.

- SEARCH\_CNDLS(obj, "X", value, lag) or obj->search\_var\_cond("X", value, lag): return the first instance of an object found starting a search from obj and conditional to contain element *X* with value value as computed at time *t-lag*.

**Example:** Suppose to have a model where several firms in different industries offer products at different prices, and you want to select randomly a price.

The model has an object *Economy* containing several objects called *Industry* (and a parameter *NumIndustries* containing the number of *Industry*'s), each of which contains a group of objects called *Firm*. *Industry* and *Firm* contain parameters called *IdIndustry* and *IdFirm*, set to increasing values from 1 to the last element in the group. Moreover, suppose that each *Industry* contains parameter *NumFirms* containing the number of descending objects *Firm*, each with a parameter *Price*.

An equation could have the following code to select randomly one *Industry* and, within this, a random *Firm* reporting the *Price*:

```
EQUATION("ChooseRandomPrice")
/*
Choose randomly a firm within a randomly chosen industry, and return
the price of the product in the chosen firm
*/
v[0]=V("NumIndustries");
v[1]=rnd_integer(1,v[0]);
cur=SEARCH_CND("Industry",v[1]);

v[2]=VS(cur,"NumFirms");
v[3]=rnd_integer(1,v[2]);
cur1=SEARCH_CNDS(cur1,"Firm",v[3]);

v[4]=VS(cur1,"Price");
RESULT( v[4] )
```

### 5.6.13 RNDDRAW("X", "Y") or p->draw\_rnd("X", "Y", 0)

Return a randomly chosen object with label *X* among a group of them contained in the object under computation, where each object has probability of being chosen proportional to the value of variable or parameter *Y*, as computed at time *t*.

Other members of the same family are:

- RNDDRAWL("X", "Y", lag) or p->draw\_rnd("X", "Y", lag) : Return a randomly chosen object with label *X* among a group of them contained in the object under computation, where each object has probability of being chosen proportional to the value of variable or parameter *Y*, as computed at time *t-lag*.
- RNDDRAWS(obj, "X", "Y") or obj->draw\_rnd("X", "Y", 0) : Return a randomly chosen object with label *X* among a group of them contained in the object obj, where each object has probability of being chosen proportional to the value of variable or parameter *Y*, as computed at time *t*.
- RNDDRAWLS(obj, "X", "Y", lag) or obj->draw\_rnd("X", "Y", lag) : Return a randomly chosen object with label *X* among a group of them contained in the object obj, where each object has probability of being chosen proportional to the value of variable or parameter *Y*, as computed at time *t-lag*.

The above functions need to compute the total sum of *Y* in order to assign the probabilities. A related group of functions skip this computation allowing the modeller to provide the sum directly, making the execution of the function faster. The family of functions is:

- `RNDDRAWTOT("X","Y", tot)` or `p->draw_rnd("X", "Y", 0, tot)` : Return a randomly chosen object with label  $X$  among a group of them contained in the object under computation, where each object has probability of being chosen equal to  $Y/tot$ , with the value of variable or parameter  $Y$  computed at time  $t$ .
- `RNDDRAWTOTL("X","Y", lag, tot)` or `p->draw_rnd("X", "Y", lag, tot)` : Return a randomly chosen object with label  $X$  among a group of them contained in the object under computation, where each object has probability of being chosen equal to  $Y/tot$ , with the value of variable or parameter  $Y$  computed at time  $t-lag$ .
- `RNDDRAWTOTS(obj, "X","Y", tot)` or `obj->draw_rnd("X", "Y", 0, tot)` : Return a randomly chosen object with label  $X$  among a group of them contained in the object `obj`, where each object has probability of being chosen equal to  $Y/tot$ , with the value of variable or parameter  $Y$  computed at time  $t$ .
- `RNDDRAWTOTLS(obj, "X","Y", lag, tot)` or `obj->draw_rnd("X", "Y", lag, tot)` : Return a randomly chosen object with label  $X$  among a group of them contained in the object `obj`, where each object has probability of being chosen equal to  $Y/tot$ , with the value of variable or parameter  $Y$  computed at time  $t-lag$ .

**Example:** Suppose to write a model where consumers choose a product randomly as a function of the market shares of the sales of each firm. The model should include an object *Industry* containing a function *Choose*, and a group of object *Firm* (containing variable *MarketShare*). The code for *Choose* in *Industry* can be expressed as:

```
FUNCTION("Choose")
/*
Choose randomly a firm with probability proportional
to the market share and return the IdFirm
*/
cur=RNDDRAWTOT("Firm", "MarketShare",1);

RESULT( VS(cur, "IdFirm") )
```

**5.6.14** `CYCLE(obj,"ObjLabel")` or  
`for(obj=p->search("ObjLabel");obj!=NULL;obj=go_brother(obj))`

This Lsd expression is a very frequently used form of the `for` cycle (see 5.2.4, pg. 52). It is used to create a cycle where at each iteration a different element of a group of *ObjLabel* is stored in the temporary variable for objects `obj`. In the normal form, the group of objects must be contained in the currently computed object.

Another form of the cycle is:

- `CYCLES(objfrom, obj,"ObjLabel")` or  
`for(obj=objfrom->search("ObjLabel");obj!=NULL;obj=go_brother(obj))`: in this form the group of objects *ObjLabel* is contained in the object `objfrom`.

**Example** A commonly used index of concentration is the inverse Herfindal index<sup>41</sup>. The index is computed as 1 divided by the sum of the square of market shares:

$$InvHerd = \frac{1}{\sum_{i=1}^n (ms_i * ms_i)}$$

Suppose to have the variable *InvHerd* contained in an object containing also a group of objects *Firm*, containing variable *ms* for market shares. The equation for *InvHerd* is:

<sup>41</sup>The index is a value from 1 to n, the number of firms. The lower the value the higher the concentration.

```

EQUATION("InvHerf")
/*
Compute the sum of ms for firms and return its inverse.
*/
v[0]=0; //set the counter to 0
CYCLE(cur, "Firm")
{
v[1]=VS(cur, "ms");
v[0]=v[0]+v[1]*v[1]; //sum the square of ms
}
RESULT( 1/v[0] )

```

**5.6.15** SORT("ObjLabel","VarOrParLabel",DIRECTION) or  
p->lsdqsort("ObjLabel","VarOrParLabel",DIRECTION)

Sorts the group of objects descending from the object under computation with label *ObjLabel* according to increasing (if *DIRECTION* is "UP") or decreasing (if *DIRECTION* is "DOWN") values of *VarOrParLabel*.

The other members of the same family are:

- SORTS(obj, "ObjLabel", "VarOrParLabel", DIRECTION) or  
obj->lsdqsort("ObjLabel", "VarOrParLabel", DIRECTION) : Sorts the group of objects descending from obj with label *ObjLabel* according to increasing (if *DIRECTION* is "UP") or decreasing (if *DIRECTION* is "DOWN") values of *VarOrParLabel*.
- SORT2("ObjLabel", "VarOrParLabel1", "VarOrParLabel2", DIRECTION) or  
obj->lsdqsort("ObjLabel", "VarOrParLabel1", "VarOrParLabel2", DIRECTION) : Sorts the group of objects descending from the object under computation with label *ObjLabel* according to increasing (if *DIRECTION* is "UP") or decreasing (if *DIRECTION* is "DOWN") values of *VarOrParLabel1*; if two objects have identical values of *VarOrParLabel1*, then their ranking is determined by the values of *VarOrParLabel2*.
- SORTS2(obj, "ObjLabel", "VarOrParLabel1", "VarOrParLabel2", DIRECTION) or  
obj->lsdqsort("ObjLabel", "VarOrParLabel1", "VarOrParLabel2", DIRECTION) : Sorts the group of objects descending from obj with label *ObjLabel* according to increasing (if *DIRECTION* is "UP") or decreasing (if *DIRECTION* is "DOWN") values of *VarOrParLabel1*; if two objects have identical values of *VarOrParLabel1*, then their ranking is determined by the values of *VarOrParLabel2*.

The sorting method is the "qsort" implemented in the standard GNU C library, adapted to Lsd objects.

**Example** Consider a model where an object *Industry* contains a group of objects *Firm*. The following lines of code in an equation for a variable contained *Industry* obtain the *Price* value of the firm with the highest market shares:

```

...
SORT("Firm", "MarketShare", DOWN); //sort firms
cur=SEARCH("Firm"); //take the first in the group
v[0]=VS(cur, "Price");
...

```

**5.6.16** ADDOBJ("X") or p->add\_an\_object("X")

Add a copy of object *X* to the currently computed object, including its descendants if these exist. All the numerical values concerning the object (parameters, the number of

copies of descendants, and the lagged values of the variables), are identical to the values set for the very first copy of object *X* defined in the configuration of the model when starting the simulation run. The same applies for the elements' options concerning whether to save values for analysis or results and debugging options. The series for save data will fill with missing values the values up to the time of creation of the object.

The function returns the copy of the object, so that the modeller could initialize some its values.

Other functions member of the same family are (all return the newly created copy):

- ADDOBS(obj, "X") or obj->add\_an\_object("X"): add new copy of the object *X* to obj. Initialization of the new object is done as described above.
- ADDOBJ\_EX("X",obj) or p->add\_an\_object("X",obj): add new copy of the object *X* to the currently computed object. The new copy is an identical copy of obj, including the initialization for parameters, lagged variables and number of descendants.
- ADDOBS\_EX(obj1, "X",obj) or obj1->add\_an\_object("X",obj): add new copy of the object *X* to obj1. The new copy is an identical copy of obj, including the initialization for parameters, lagged variables and number of descendants.

**Example** Suppose to have a model with an object *Industry* containing a group of object *Firm* having a variable *Production MarketShare* and *Capital*. In *Industry* we could have a variable *Entry* creating a new firm, whose market share is set to 0, the initial capital is set to a conventional value, and allowing the newly created firm to compute its own production.

```
EQUATION("Entry")
/*
Insert a new firm and initialize the variables
*/
cur=SEARCH("Firm"); //take the first object Firm in the descending group
cur1=ADDOBJ_EX("Firm",cur);
WRITES(cur1, "MarketShare",0); //assign 0 to the market share of the new firm

//parameter in Industry defining the initial capital for new firms
v[0]=V("InitCapital");
WRITES(cur1, "Capital",v[0]); //assign 0 to the production of the new firm

//assign 0 to the production of the new firm setting
//the variable as computed at previous time
WRITELS(cur1, "Production",0, t-1);

//allow the equation for Production to compute the
//initial level of production
V("Production");
RESULT( 1 )
```

### 5.6.17 DELETE(obj) or obj->delete\_obj()

Remove object *obj* from the model. Variables and parameters saved will show their value up the time step they have been computed, filling with missing values the other times.

The object removed is immediately removed from the model, and therefore this function must always be used for objects different from *p* or *c*. In fact, using the command DELETE(*p*) will ask Lsd to remove the object whose equation is being computed, which is not possible.

### 5.6.18 PARAMETER or param=1;

This line placed anywhere in the code for an equation transform the variable computed in a parameter. This avoids the code for the variable to be computed again during the simulation run.

**Example** Suppose that your model contains a group of firms, and you want to be sure that the parameter *IdFirm* is set to increasing values (some user may mishandle the initialization of a model). Then you could write an equation for a variable *Init* which will be computed only once:

```
EQUATION("Init")
/*
Assign correctly initial values
for IdFirm, and then transform Init in a parameter.
*/
v[0]=1;

//for all firms
CYCLE(cur, "Firm")
WRITES(cur,"IdFirm", v[0]++); //assign IdFirm and increase v[0]

PARAMETER
RESULT( 1 )
```

### 5.6.19 INTERACT("TEXT",value)

This function interrupts the simulation (much like the debugger, see paragraph 4.4.22 at pg. 46) showing the text specified in the equation and allowing the entry of a value by the user of the model.

For example, the following equation asks the user of the model to ask for a value:

```
EQUATION("Price")
/*
Price value, computed with a linear demand curve.

If the price is negative, ask the user the price value
*/
v[0]=V("Quantity");
v[1]=V("a");
v[2]=V("b");
v[3]=v[1]-v[2]*v[0];

if(v[3]<0)
v[3]=INTERACT("Negative price. New price? ",v[3]);

RESULT( v[3] )
```

Of course, this function requires the user to pay attention during a simulation run. Actually, this function is generally used only for debugging purposes, in order to observe the model status in particular cases.

### 5.6.20 close\_sim() function

Every model has available a C++ function which is always executed at the end of a simulation exercise, after the last variable has been computed. This is to allow models

that allocate memory or keep open files to clean up the environment before returning to the user.

In most cases modellers do not need using this function, which is defined in the end of the equation file as a do-nothing function. In case of necessity modellers can fill this function.

## A Error Messages

There are four types of errors that a Lsd user may encounter. First, the Lsd system may be configured wrongly for your system. These errors appear either when you try to compile LMM (Unix systems only) or when you try to compile a Lsd model program. For example, you may have installed a Tcl/Tk distribution different from the ones specified in the system.

The second type of errors concern faulty equations' code, with errors that prevent the C++ compiler to produce a Lsd model program. For example, one may have written a line without the terminating semicolon. These errors appear in LMM when you try to compile and run the Lsd model program after having edited the equations.

The third type of error appears when one misuses the Lsd model program interfaces issuing commands that are impossible to execute in the present context. For example, you may have tried to run a new simulation run immediately after another run, therefore with the Lsd model program containing the last time step simulation data instead of a fresh data configuration.

The last type of error concerns mathematical or logical inconsistencies becoming apparent during a simulation run. For example, one variable goes to zero, and another equation uses this variable as denominator in a division.

The next paragraphs list the four types of errors and suggest possible causes and available solutions.

### A.1 Configuration errors

These errors concern only Unix users. If you are a MS Windows user, the only possibility for having a misconfigured system is that you removed part of the Lsd installation. Just restore the original installation and the problems should be over.

#### A.1.1 /usr/bin/ld: cannot find -ltcl8.3

If you are a Unix user and try to compile with the batch file you may receive this error message.

This means that you don't have installed the Tcl/Tk library or, more likely, that you have the version different from the 8.3.

Fix: Edit the `comp.linux` try to remove the version numbers altogether, or place the correct version number in place of the 8.3. Remember that you will have the same problem when compiling the Lsd model programs. You will have to update the system options with the same fix that works for compiling LMM. Use in LMM the menu **Model/System Compilation Options**.

#### A.1.2 undefined reference to '\_gxx\_personality\_v0'

This problem emerges with the latest version of the GNU compiler, such as the one distributed, for example, with the RedHat 8.0 and Mandrake 9.0 (and presumably also sub-

sequent distributions). Edit the `comp.linux` and replace "gcc" with "g++". With an editor (or with LMM) open the file `makefile_base.txt` in the directory `Lsd/src` and replace all instances of "gcc" with "g++".

### A.1.3 Other undefined reference ... errors

On different system may be necessary to link other libraries, like `socket`, `X11` etc. Normally, the undefined referenced function should explicate which library is missing. Insert in the `comp.linux` file the `-lmy_library`.

## A.2 Equations' programming errors

These errors appear when you try to compile a Lsd model program with a grammar error in the equation file. In this case LMM issues an error message and a new window appears in the background, labelled *Compilation Results*. Note that the error message may signal that, although LMM could not compile the new Lsd model program, an old one exists. In this case you have the choice to run the former Lsd model program, although, of course, it will not contain the updated equations' code.

The following errors are listed according to the lines appearing in the *Compilation Results* window

### A.2.1 `fun_XXX.cpp:99: parse error before ...`

The equation file contains a grammar error at or, more likely, just before the line number indicated (99 in the example). Typically, it may be a missing semicolon terminating a command line. Note that frequently an error in one location causes a long series of apparent errors in the subsequent lines. Therefore concentrate only in finding and fixing the very first error, and then try to re-compile. Likely, the other errors will disappear.

A very rare case is the following. If the line number corresponds to the latest opening curly bracket `{`, normally the one for the function `close_sim()`, then this means that there is an extra opening curly bracket not matched by a closing one. In this case, remove the opening bracket in the `close_sim()` function, making it like:

```
void close_sim(void)

}
```

Then, locate the cursor just before the latest closing curly bracket and press `control+m`. The LMM editor will show the matching opening curly bracket likely to be the faulty one.

Either remove the stray opening bracket or insert the missing closing bracket. Before re-compiling re-insert the opening bracket in the `close_sim(void)` function.

### A.2.2 `lsd_gnu.exe: Permission denied`

You are a MS Windows user and you are trying to compile a Lsd model program while an old version of the program is running. The system is not able to overwrite the Lsd model program file and therefore the error message is issued.

Close the running Lsd model program and re-compile.

### **A.2.3** fun\_pippo.cpp:99: label 'end' used but not defined

There is a closing curly bracket } not matched by an opening one. The compiler noticed this at line 99 but it is located before that line.

Search the extra closing bracket and remove it, or place the missing corresponding opening bracket.