Laboratory for Simulation Development – LSD

Marco Valente

marco.valente@univaq.it

March 1, 2014

Contents

Chapter 1

Why using LSD?

Writing a simulation model entails two operations, conceptually distinct. Firstly, one needs to design the model, using whatever support seems suitable: mathematical formulation, pseudo-code, verbal description etc. Secondly, the output of the first step, call it an "abstract" model, must be translated into a computer program, using a programming language. In theory, any programming language is potentially able to implement whatever computation is required by a model. However, in practice, languages differ, even radically, in respect of the possibility to implement in reasonable time particular classes of models. We may broadly classify different languages across two hypothetical scales of simplicity of use (e.g. how much training and skills are required to implement a given model) and power (e.g. execution time, maximal dimension of the model). The distribution of languages presents an inverse relation between these to indicators: simple languages are very limited, while powerful languages are difficult to use and required specialized programming skills.

LSD breaks this hypothetical trade-off, offering a very simple-to-use language able to generate professional grade simulation programs whose speed and dimension are limited only by the computational power embedded in the available hardware. Moreover, LSD also offers features specific to implement research-oriented simulation models, features that contradicts the principles of programming for end-users. Below we briefly summarize the most relevant properties of LSD.

1.1 Simple to use <u>AND</u> powerful

The first property is the simplicity of use. Modelers without prior skills can implement simulation models with a few hours of training, enjoying powerful interfaces to produce even sophisticated analyses of massive amounts of data. The structure of a LSD model is designed around how people naturally think of a model, not on how the model is actually implemented. Essentially, users are invited to consider their model as a set of difference equations that the system automatically computes for a number of steps. As a general philosophy, the LSD system requires users to input only the very necessary information (e.g. individual equations), while the system automatically retrieves details necessary for the execution implicitly available (e.g., the scheduling of equations' updating). Hence, building even highly complex models requires very little learning time.

Second, users demanding high performance simulation programs can exploit the fact that LSD is implemented as a set of tools built on top of C++. Hence every library and computational structure compatible with this language can be included in a LSD model, effectively making LSD the most generic language available. Also inherited from the underlining C++ is the power of LSD models at execution time. A pilot model can be designed and tested with, say, tens of entities using the graphical interfaces endowed automatically in any LSD model. When this is done, a click of the mouse generates configurations with millions of copies, fully exploiting the computational power available. Finally, the very same code for a model implemented on a personal consumer (e.g. Linux, Mac or Windows) can be transferred to a high performance computing center for massive amount of resulting data, exploiting the possibility to generate command line versions of LSD models executed in batch mode.

While simplicity of use and power of execution are the features normally considered when

evaluating a generic programming language, two other aspects are specifically relevant to research-oriented simulation models: flexibility to modification, even radical, and accessibility of the code at run time.

1.2   Open-ended simulation models

Any programmer is taught that the very first step in a software project consists in designing the structure of the future program as a function of the careful analysis of users' requirements. Modifying such structure at a later stage is very hard and extremely dangerous, introducing instability and poor functionality. The typical result of a badly designed software is that adding incrementally ad hoc adjustments produces increasingly complex irrational code, reaching the point that the easiest choice is to scrap the project altogether and start from a clean sheet. However, research-oriented simulations, by definition, only rarely enjoy the possibility to clearly predict the directions of development of a model. Typically the researcher starts with an initial prototype and, depending on the result, will decide how to expand it. Hence, researchers using simulation models are bound to either limit to implement models that produce perfectly known results, or are subject to recurrent *complexity crises* forcing them to loose all work done to start again a novel project that, in effect, have only minimal differences with the original one.

LSD greatly reduces the risk of getting one's code entangled in layers of corrections. One reason is that many of the decisions concerning the technical implementation of the code are automatic, so even radical changes costs nothing in terms of modeler efforts. For example, inverting the order of execution of variables can be generated with a single character in an equation code. More in general, LSD favors naturally a modular design. Each variable is associated to a piece of code, typically very simple, providing the expression to compute the value for the generic copy of that variable at a generic time step. At simulation time, all pieces of code for the necessary elements are arranged so as to produce a working computational structure without need of the modeler intervention, unless errors occur (and, in this case, a full report on the origins and possible fixes are provided). Consequently, for example, the same element in a model can be defined as parameter in a first version, then turned into a variable computed according to a structural function, or finally associated to the result from the highly complex elaboration performed by external libraries. The modeler does not need to care for possible effects of the different choices on the rest of the model, as the system automatically re-arrange the computations as necessary to the current state of the model[1]. The effect of the LSD natural modularity is that changing the content of an existing model is a very smooth process, rarely causing problems. Because of the very structure of a LSD mode, users are strongly invited to adopt a very gradual approach to model building, implementing one equation per time, and testing its proper working before adding new elements. Code from different models can be seamlessly merged relying on the automatic controls for missing elements, inconsistencies, or any other possible problems. For example, in two separated models can be implemented the demand and the supply of a market. Once these models have been properly tested in isolation they can be later merged without any effort even in case the interaction among the elements of the two models necessitate radical modification at run time, since these will be automatically provided by the system.

In conclusion, the intrinsic modularity of LSD and the automatic filling of implicit information permits to modify any part of the model at any stage, simplifying both the adjustment of an ongoing project and the re-use of code developed for other purposes. These features are particularly valuable for simulation models developed for research purposes, whose content and overall structure cannot be planned in advance as with software projects developed for a well-specified purpose.

1.3   Controlling simulation models

Programming languages are devised to implement efficiently code generating services to the user of the program, but not to show what is happening within the execution of a program. Indeed,

---

[1]Indeed, during the same simulation run the same element can be "frozen" as parameter and turned temporarily to a variable to execute specific pieces of code, in effect changing dynamically the very structure of the simulation program as result.

we use computers exactly because they execute at extreme speed many operations without the requirement for the user to monitor how these operations are performed. Inspecting the state of a program, and understanding how exactly it reached a certain condition, is a very difficult task, requiring both skills and special tools devised for the purpose. For general programming projects (and therefore also for simulation models), the internal inspection is obviously required in order to fix errors, normally called debugging. But simulation models have also a further requirement when used in a research project. In fact, in general programmers know the desired output of the program, and the need to inspect the inner working of the program arises only with the software behaviour is not the expected one. Conversely, the very reason for implementing a simulation model for research purposes is that we do not know with certainty what results should be produced. The inspection of the inner workings of the simulation program is therefore a necessity not only to fix errors, but also to investigate the motivations for a specific aggregate behaviour.

A standard programming language is designed considering that only the developer is able to access its internal code, for example using debugging tools. The user of the program, on the contrary, can control the program only making use of interfaces specifically developed by the programmer, and is generally forbidden the access to the internal working of the program. This has the cost for the programmer to write code for the interfaces, but has the advantage that the users don't have the possibility to mess up with the program unless using the interfaces prepared by the programmer, and therefore the being subject of the limitations embedded in those interfaces.

For research simulations this is a serious obstacle to the exploitation and diffusion of a model. Either the modeler devotes precious time to the tedious job of adding layers of interfaces, or it is impossible (or extremely problematic) to perform certain tasks. Even relatively small models, for example, can include dozens (or, worse, a varying number) of parameters affecting their results. Without appropriate interfaces it is impossible to even know the values used in a given configuration, not to mention changing them. Similarly, the results produced by a simulation run concern dozens or hundreds of series. Lacking appropriate tools it is impossible to access these series, and even less to investigate the process of generating these series at specific moments. The result is that a typical simulation model distributed to users includes the possibility to modify a few options among the many potentially relevant to explore, and produces as output only one among many possibly interesting results. Moreover, the events during a simulation run are generally impossible to access, apart a few indicators decided once and for all by the modeler. Consequently the whole system is, in the ends of the user, a black-box producing a trickle of data, possibly affected by a few initial settings. Any scientific claim concerning the simulation run properties need therefore to rely on the trust one has in respect of the modeler claim (and his capacity to have succeeded in implementing the claimed model). This is a decisively poor practice among researchers, and LSD offers powerful solutions to the problem of controlling a simulation model.

LSD models do not require users to write specific code for interfaces. Rather, a professional suite of windows, controls, options, etc. is automatically attached to any element at the time of its creation. Using these interfaces both the modeler and any subsequent user can access, both to observe or to modify, any possible relevant piece of information concerning that element. The interfaces can be divided in three classes depending on the stage they can be used.

Before a simulation run graphical interfaces allow to generate, edit, move or delete any element. For elements requiring numerical values, such as parameters, the system offers a wide variety of sophisticated initialization tools, ranging from the trivial manual entry to the comprehensive setting of million of entities according to elaborated procedures.

During a simulation run users can interrupt the model, inspect any element, proceed step-by-step, edit the value of selected elements, and complete the simulation run, so as to perfectly understand every event taking place at any level.

At the end of a simulation run (or during an interruption), every value produced during the simulation is available for graphical and statistical elaboration. The LSD interfaces are particularly suited to deal with the massive amounts of data easily produced by modern computers, allowing users to select which datum to use.

Across all the stages of building a model, a fairly complete error catching system prevents the crash of the program and supplies any information concerning errors or inconsistencies, suggesting possible fixes.

In conclusion, a model implemented in LSD offers the modeler unique tools to explore any

possible property of the model and support any related claim by merely instructing others to replicate the very same operations. Skeptics of a given result have the possibility to observe in detail the claimed property, and use the very same interfaces used for the claimed discovery to challenge its existence, generality, robustness, etc.

1.4 Simulation Model elements

There are many different languages to implement a simulation model, and even more formats to express them. In the following we propose a generalized format to define the content of a model. We claim tha the elements necessary to fully describe a model, whatever language is used to implement it, are the following:

- **Variables**
  - Label: a text identifying uniquely the type the variable.
  - Equation: a routine computed once and only once at each time step returning the value for the variable.
  - Initial values: values used in the earlier time steps to be used as past values for the variable.

- **Functions**
  - Label: a text identifying uniquely the type the function.
  - Equation: a routine computed any time, and only, the function is called by other elements of the model.
  - Initial values: values used in the earlier time steps to be used as previously computed values for the function.

- **Parameters**
  - Label: a text identifying uniquely the parameter.
  - Initial values: value assigned to the parameter.

- **Objects**
  - Label: a text identifying uniquely the object.
  - Content: list of variables, functions, parameters and other objects contained in the object.

  We call the **structure** of a model a set of objects hierarchically connected each containing a set of variables, parameters and functions. The structure of a model defines univocally the computational content of a model, but it is still not sufficient to produce results. For this we need to add to the structure the numerical specification for each element.

- **Initialization**
  - Number of objects
  - Initial values for lagged variables and functions, and for parameters
  - Contextual initialization (e.g. num. of time steps, pseudo-random number series, etc.)

The joint definition of a structure and of the initialization provides a **configuration**, which substantially provides a fully specified simulation model producing always the same results independently from the language used for the implementation. Before exploring the LSD interfaces we give provide below a slightly more detailed definition of the model elements.

1.4.1 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. Variables generate values by means of an algorithm computed once, and only once, for each time step, and returning a single value.

Some variables may need to be initialized before the start of a simulation run. 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.

1.4.2 Functions

In some cases a simulation requires to produce numerical values that are independent from time, but are simply functions that produce values upon request by some other events in the model. The different with variables depends on the timing of their computation, that is the use of the associated algorithm used to compute a numerical result. While a variable needs to have one, and only one, numerical value for each time step (and therefore its algorithm is computed once, and only once, at every time step), a function needs to provides a fresh result from its algorithm only when requested. This may, possibly mean several times in a single time, or none.

1.4.3 Parameters

Parameters are elements that do not change value of their own accord. In most of cases parameters are assigned an initial value before the beginning and maintain it throughout the whole simulation run.[2]

1.4.4 Objects

Most cases modelers find convenient to define a model element (say, a variable) and implement a whole set of copies of the same variable, sharing the same name and equation, but producing independent values.

While vectors are a common format to implement multiple copies, they have several drawbacks advising against their use in programming. In the past most programming languages moved to adopt an object-based approach. Intuitively, objects allow the equivalent of matrices with rows containing potentially a different number of columns, or other matrices of varying dimensions.

The best way to think of objects is to consider their difference with vectors. Suppose you need three elements: X, Y and Z. If you wish to have many copies, N, for each element, then you would need to define three vectors, each of which needs to be independently managed.

	$Object_1$	$Object_2$	...	$Object_N$
$\vec{X}$	X_1	X_2		X_N
$\vec{Y}$	Y_1	Y_2		Y_N
$\vec{Z}$	Z_1	Z_2		Z_N

Table 1.1: Vectors (listed horizontally) are sets of elements of the same nature, while objects (listed vertically) are collections of elements of different nature. In both cases we can express sets of copies of the same elements, but objects are more convenient for programming in general, and simulations in particular.

Using an object-based approach you define *objects*, that is, containers, each storing one copy of the elements. Creating many copies of the objects ensures that the numerosity of the elements is correct, and facilitates also the access of elements associated to the same copy of the object.

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.

An object structure permits to implement more easily 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 by changing the number of this objects. Instead, in a vector-based language it would be required to increase the dimension of all the vectors for the elements defining a firm, and those defining departments.

---

[2]Exceptionally, however, parameters may be overwritten during the calculation of the equation for some variable or function. This is typically the case of equations producing several results in a single computation. Since variables can store only one value, any additional result is stored in appropriately defined parameters.

1.4.5   Data required for a simulation run

A simulation model described in terms of its elements and its code can produce different results depending on the numerical values used to initialize it. This paragraph lists the class of values composing the initialization of a model.

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. object Market containing object Supply and Demand; Supply contains Firms and Demand contains Consumers, etc.), the number of objects must be specified for each group, subgroup and so on.

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. These values will be used in the earliest time steps of a simulation, where the "past" values could not be computed, and must be provided by the modeller.

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[3]. 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, if the same seed is used, or different ones for different seeds.

1.5   Overview of the LSD package

The LSD format for a model is identical to the format defined above, providing suitable interfaces to supply each piece of requested information, and automatically inferring from these all the technical details necessary to produce a working simulation program implementing the abstract model.

A LSD model is defined by three sets of elements:

- **Equations**: the only code modellers need to provide consists of independent chunks of code expressing how the values of the variables in the model must compute their value. As in a difference equation model, the equations are expressed as the computation to be performed in a generic time step for a generic copy of the variable.
- **Model configuration**:
  - **Model structure**: the modeller needs to define the labels the objects, parameters, variables and functions. A structure is a generalized model, still lacking the numerical definitions required to run a simulation.
  - **Initial data**: To run a simulation is necessary to complete the model structure with the values necessary to start a simulation run: number of objects, parameters' values, number of time steps, etc. A given model structure may be used to generate many configurations using different numerical values.

All the above components of the model can be produced in a variety of ways (eventually, they consist of text files, so that any text editor may be used). However, an extensive set of tools and interfaces assists the modeler in inserting quickly and efficiently the required information, minimizing the possibility of typing mistakes and easily allowing modifications.

The computational part of a model, that is the equations, are written using an editor specifically built for LSD, called LSD Model Manager, LMM. Using LMM modelers write the equations as short chunks of code, and the turn these equations in a LSD model program. LMM takes care of the compilation, the process of turning the C++ code (both the system code and the model-specific equations) into the executable. In case of failure, compilation issues detailed messages on which part of the code contains errors. Figure 1.1 describes these steps.

---

[3]Computer languages provide several random functions, generally derived from the uniform [0,1] random function.
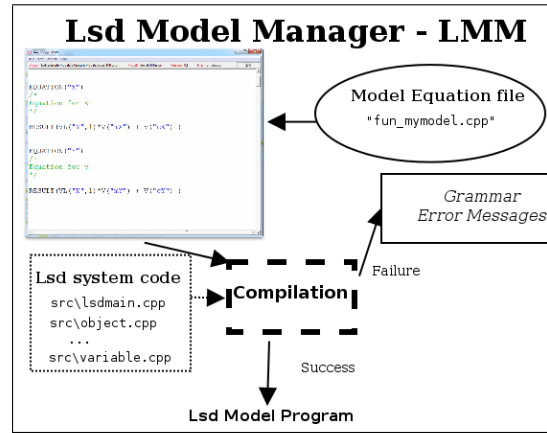
Figure 1.1: Overview of the functionalities provided by LMM, a special-purpose editor for L^{S}D equations. LMM takes care compiling the model equations and tries to produce a L^{S}D Model program, using the L^{S}D system code. On success it runs the resulting program.
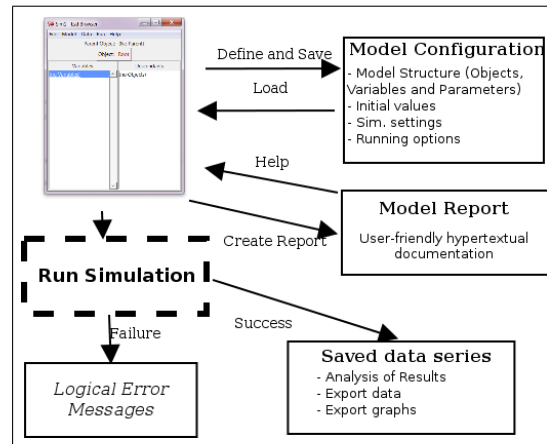
Figure 1.2: Overview of the functionalities the L^{S}D model program. They concern the definition and modification of configurations, generating and managing results, documenting the content of the model.

After a successful compilation LMM automatically starts the resulting program, a L^{S}D model program capable of computing the equations. This program is used for any activity on the model besides those concerning the equations. It allows the definition of the model configuration, run simulations, analyse the results, etc. (see figure 1.2). The same program also manages files storing different configurations (e.g for different initializations of the model), the documentation and the export of results. The interfaces of the programs are all identical, so that a user experienced in one model will be immediately able to explore and exploit a different model, focusing exclusively on the content of the model.

In the following we provide a broad description of the functions contained in LMM and in the L^{S}D model programs.

1.5.1   L^{S}D Model Manager - LMM

LMM is a program helping modelers to perform all the operations required to generate a L^{S}D model program. In particular, it manages the technical details of the C++ files related to the compilation process, so that users do not need to pay any attention to the underlining C++ layers, and have friendly interfaces for specific purposes, such as compiling optimized code.

Managing simulation projects   A simulation project generates, besides the model itself, result files, documentation, several configurations. A research project may be composed of several, inter-

mediate models, each of those relevant to be saved for potential future re-use. Finished projects may need to be reviewed, or continued. In practice, this means to deal with dozens, or hundreds of files, which easily generate enormous confusion unless properly organized.

LMM assists users in the proper housekeeping of simulation projects. When a new project is created, LMM generates a new directory and all the files necessary to start working on model. File names and directory locations are automatically managed, minimizing the possibility of confusion. Model projects are presented by a browser that, although relying on the directory structure on the disk, presents existing models showing their extended names and descriptions, so to easily find the desired project even in a disk crowded with models. Moreover, it is possible to generate groups of related models, or generating an identical copy an existing project to test modifications without risking the current version. LMM allows also to compare different models, showing the differences in the code between two different versions of the same project.

During the work on a simulation model, LMM automatically generates all the technical files required to use the C++ underlining machinery. These files, containing, for example, the compilation options, can be ignored by the users or modified. Furthermore, any relevant information (e.g. compiler's error messages) are presented and saved automatically by LMM.

Generating the model's code   Concerning the main role of LMM, the generation of a model's code, LMM offers an editor particularly suited for LSD models. For example, it contains short-cuts that LSD modellers can use to insert automatically (i.e. error-free) the most frequently used keywords for LSD equations, so that users need only to choose the desired function and type only model-specific text (i.e. the names of the variables). The LMM editor offers automatic indentation, text coloring and parenthesis matching features, besides an extensive on-line help concerning the equations' grammar, useful to develop LSD equations' code.

LMM also manages to transform the file of a model equations into a LSD model program, that is, to compile the equations and build the program containing them. This operation consists in using several programs to generate executable files from C++ code, adding the libraries required by the operative system, etc. LMM avoid users to pay any attention to these operations, though it allows expert users to modify any compilation option (e.g. optimization levels, inclusion of external libraries, etc.).

The output of the compilation depends on the whether errors have been found during the compilation process. If the equation file contains illegal code (typically, typing mistakes or missing parenthesis), then a window presents the user with the list of errors and any information available to fix them. Otherwise, a successful compilation generates a LSD model program which is automatically launched.

1.5.2   LSD model programs

LSD model programs are are used for any operation concerning a simulation model besides the writing of equations. Here is a short summary.

Define, save and load configurations   An extensive set of graphical interfaces allow to define the label of the model elements and any required information. The interfaces are easy to use and automatically adapted to represent the the information already contained in the model. For example, they prevent the definition of a new element with a label already used for another element. The number of values to be introduced is automatically determined, depending on the number of objects present in the model. Models with large number of elements (requiring, for example, millions of parameters to initialize) can use automatic functions to generate commonly used patterns, or load data from external files.

Configurations are stored as text files, which can be loaded, edited and renamed at any moment.

Document the model   LSD model programs are themselves tools to observe the model efficiently and in any detail. The central window focuses on a single object, presenting all the elements contained, while a graphical representation shows the full structure of the model, allowing to easily moves across different objects. User can request any detail for any element, for example asking

how many variables make use of a given parameter, their labels and position in the model, and even showing their very code (but not, as repeatedly said, modify the code, since this is compiled into the very LSD model program).

The description of a model can also be generated in a format that dispense from the use of LSD altogether, for example to generate an appendix to a paper. With a single command it is possible to upload as comments to each model element all the information contained in the model equations and in the model configuration. The comments are then used to generate a HTML file listing all the element of the model, the numerical initialization and their relations as hyperlinks. The structure of the file, called model report, allows the inspection from different perspectives, like a brief overview based on short comments only, or the detailed computational content of the equations. Therefore, even non-programmers can appreciate the content of large and complicated models.

Initializing models   Assigning initial values can be a rather messy business when the model contains objects at several hierarchical levels and requiring thousands or millions of values. LSD offers highly efficient interfaces that are, at the same time, extremely simple to use and able to perform highly sophisticated elaborations.

Controlling simulation runs   During a simulation run users can observe a graphical representation of a few series, interrupt the simulation on request or under pre-determined cases, inspect or modify the state of the model, perform a full analysis intermediate results, and continue the simulation.

Moreover, configurations may impose multiple runs each independently starting at the end of the previous one, saving result files for later use.

Managing simulation errors   When all the elements required for a simulation run have been defined (the system prevents incomplete simulations from running, signalling the missing elements), the LSD model program turns into a run-time environment controlling for errors. Events like a division by zero, the request of values from a non-existing parameter, logical inconsistencies, etc. generate an extensive list of information on the conditions generating the error and possible fixes. Also, any data produced up to the emergence of the error can be retrieved and analysed.

Managing simulation results   Results from a simulation models come in two, related forms. Firstly, and obviously, they take the form of the time series generated during a simulation run. Secondly, and frequently crucially, scientifically relevant knowledge is gained investigating the internal dynamics of the model, understanding how the model managed to generate particular, unexpected states.

Concerning the first type of results LSD model programs are endowed with an efficient module to produce graphical and statistical elaborations of the series generated. This module is particularly suited to deal with vast amount of data, ranging in the tens of thousands of series, each composed by tens of thousands of elements. It is possible to generate time series, cross-section and scatter (2D and 3D) plots, distributional histograms, and various descriptive statistics. The module allows to export graphs and data, in standard formats compatible with mostly used word processing and statistical packages.

Concerning the internal investigation during a simulation run, users can interrupt a simulation at any moment (or instruct the system to do so under specific conditions), analyse teh state of any single element of the model, use the analysis module on the partial data and return to the simulation, continue the simulation step-by-step, and modify every element.

1.6   Technical requirements

LSD is designed to run on Windows, Linux and MacOSX platforms, without requiring other software

than those distributed, or otherwise commonly found on standard installations[4] 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 apply for some accompanying package.

LSD model programs require a standard GNU C++ compiler, using only standard libraries, and the Tcl/Tk windowing language. Moreover, if available, LMM provides access to GDB for debugging models at source level[5]. 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 `makefile` file. Therefore, Linux users must ensure to have installed the compiler and Tcl/Tk. Both packages, if not already present, can be usually installed using the local package manager. Although not strictly required, it is strongly suggested to have installed GDB and GNUPLOT.

The Windows distribution includes all the software required, namely the C++ compiler (GNU compatible) and all the accompanying libraries and software (e.g. makefile, linker, libraries etc.) . The MinGW distribution (Minimal GNU for Windows, www.mingw.org) 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.

1.7 Installation

1.7.1 Windows platforms

The installation consists simply in unpacking the LSD files that are structured in a root directory (e.g. `C:`
`LSD`) and several subdirectories for the models, manuals, compiler and source code. The root directory needs to be located in a directory not containing any space in its name, nor to within parent directories with such names. For example, `C:`
`Documents and Settings`
`LSD` cannot work

When the installation is completed run the file `run.bat` in the installation directory. This will run LMM (LSD Model Manager) that allows to create new models, or select existing models.

Control that the root directory and all its descending directories must have the write permission. If you copied the directory structure from a CD, for example, you may need to set the properties on the hard disk so to be able to write in the directories.

If you move the LSD directory structure in a different location a warning will appear. When LMM starts follow the instructions to fix this problem.

1.7.2 Linux and Unix-based plaftorms

Unpack the distributed file, which may be a zip or a tgz package, and move into the LSD root directory. The objective is to compile the file `LMM`, which probably requires some testing. Once this program runs, then the rest of the system requires only minor adjusting, if any.

First of all, ensure that you have installed Tcl/Tk (possibly, even the development packages). To do this try to run the program `wish`. If this program does not exist you need to install it (download the package from http://www.tcl.tk/software/tcltk/downloadnow84.tml and follow the instructions to compile and install the system).

---

[4]Unix users are expected to have a standard Linux box with installed the C++ compiler and its standard libraries. The LSD distribution for MS Windows includes all the software required.

[5]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.

If Tcl/Tk is installed on your system try the command: `make -f makefile.ln` in the root directory. If the compilation succeeds, then you have the file `LMM`. Type `.LMM` to run it.

If the compilation fails the most likely reason is that the required elements of Tcl/Tk are located in a non-standard place (different distributions locate the files in different places). This compilation error is characterized by the compiler issuing thousands of errors concerning "tclXXX" functions not found, besides others.

The required files are:

- `libtclX.Y.a`: Tcl library - `libtkX.Y..a`: Tk library - `tcl.h`: Tcl header - `tk.h`: Tk header.

where X.Y are the numbers for the major and minor revisions of the package.

The default locations for these files are `/usr/lib` for the two libraries and `/usr/include/tclX.Y` for the two headers. If the files are not in that position try `/usr/local/lib` and the equivalent for the headers.

If the files are not in the default locations, edit the `makefile.ln` file and change the variables in the following positions:

```
PATH_TCL_LIB=/usr/lib
PATH_TK_LIB=/usr/lib
PATH_TK_HEADER=/usr/include/tclX.Y
PATH_TCL_HEADER=/usr/include/tclX.Y
```

If you needed to edit the makefile, when LMM starts, select a model and choose **Model/System compilation options**. Edit the same lines above shown by LMM, and any subsequent generation of LSD models will use the same information.

Note that the version numbers for the packages of Tcl/Tk may change in the future (currently is available the beta version 8.5). LSD tries in general to upgrade to the most recent version of Tcl/Tk, to avoid users to maintain obsolete libraries, though this needs some time.

Besides problems with the Tcl/Tk library, in some cases there are also other types of problems, caused by the local GNU compiler to require specific libraries. Compilation error messages generally make clear the missing library. Both the makefile for LMM and the system options within LMM allow to set additional libraries, if required by your system. Set them in `DUMMY=`, which is an argument passed to the compiler when linking the final executable.

1.7.3 MacOS

On Mac OS systems slightly different installations are required depending on the version of the system. Moreover, Mac users can opt for either a native graphics or use the X11 environment, in effect using the Unix layer at the core of the Mac OSX. Detailed information are available in the LSD forum at `www.labsimdev.org`.

Chapter 2

Implementing LSD models: Example 1

Implementing a simulation model for research purposes is a process prone to errors. We can divide the possible errors in two classes. Firstly, we may simply write the wrong code or values, so that the model implemented is different from the one we wanted to implement. Secondly, we may discover that, though implementing the model we originally designed, it is not appropriate for the purposes of our research, and therefore we need to modify the original idea.

In either case, an error needs to be firstly spotted and then fixed. Identifying an error can be very difficult: a large model, with dozens of routines and thousands of variables produces massive amounts of data, that are likely to be analysed only statistically, at aggregate level. Unless the error generates evident absurdities (say, negative market shares), it is well possible that the faulty code goes unnoticed.

Even in the case we identify an error and the required solution, say, replacing the code for a variable, the effects on the rest of the model may be huge, producing an avalanche of further changes on the rest of the model forcing, in practice, a complete re-writing of the whole model.

LSD provides very powerful tools to assist users in both respects. LSD models are automatically endowed with a large series of interfaces to access the state of the model in many different ways, facilitating the identification of problematic code. Also, the very structure of a LSD model is made of independent chunks of code, minimising the possibility that a few changes require the re-writing of large portions of the model. Still, even though LSD allows technically to find and fix problems at any stage of a model development, it is far easier to adjust a model

In both cases, it is good practice (and, in many case, of capital importance for the success of a project), to develop the model gradually, adding few element at a time, testing the (theoretically trivial) intermediate results, and proceed adding further complications. Without this approach, implementing at a single stroke hundreds of lines of code and dozens of values, we can be guaranteed to generate a long list of errors, whose combined effect make practically impossible to identify their original source. Moreover, even in case we managed to generate an error-free model, we are likely trapped in the black-box problem, since we cannot trace the properties of the results to the specific assumptions implemented in the model definition. In this section we will build a model step-by-step, so to have the possibility to discuss any aspect of modelling with LSD.

In this section we implement from the scratch a new model, which eventually will represent a discrete version of the replicator dynamics model. The process of model construction will be described step-by-step, with the aim of familiarizing readers with the major interfaces and operations. The steps described have also the purpose to show how a typical simulation project may proceed, by adding a few elements at a time, testing the results, editing the model, and extending it. As initial stage we start by implementing a model with one single variable, say X, computed as a random value.

2.1 Create a new model project

A LSD model requires several files, and the user is generate more, e.g. for different configurations, results, graphics, etc. To create a model it is then necessary to create firstly a directory where to place the basic files (essentially the equation file). The various models located in an installation can be organized in *groups*. The installation originally contains two groups: example models and work in progress. Users are obviously invited to place their models in the second group. Within a group the user can obviously create other groups, typically for sets of related models. Any group or model create its own directory that can be inspected, but whose content is safe not to alter unless using the LMM tools

To start the system launch we consider you already unpacked the installation file. Click on the `run.bat` file to start the system.[1].

The program starts offering automatically to open a LSD model, with the alternatives being to use LMM as text editor. Opting to use models the system activate a model browser showing all the models present in the installation, which also permits to create new ones.[2]
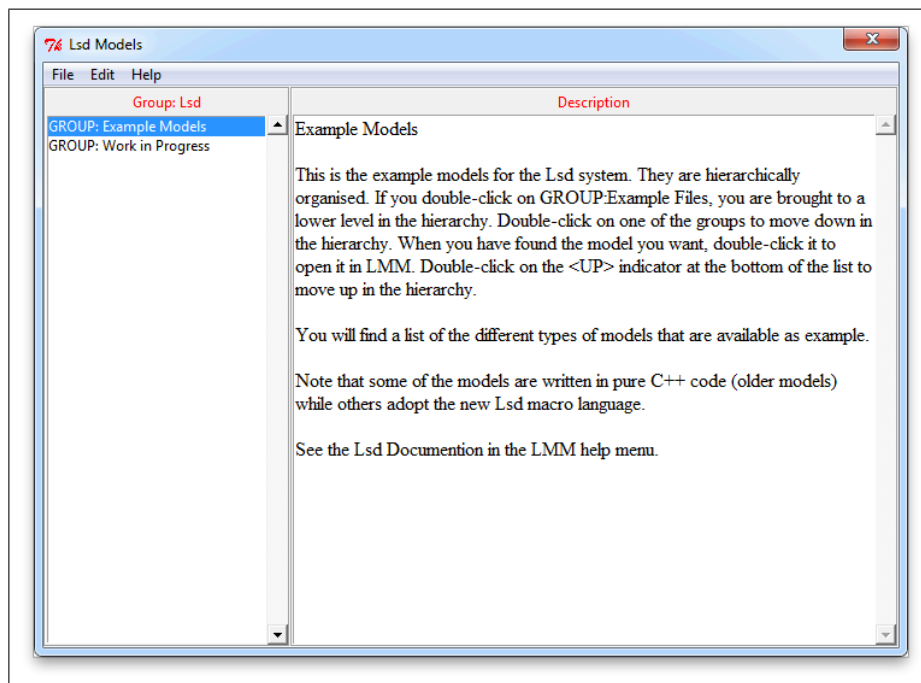
Figure 2.1: Model browser, initial screen

Using the mouse pointer or the arrow keys choose a group, e.g. "Work in Progress", and then uses the module's own menu **Edit** and then **New Model/Group**. You will have the choice between creating a new group or insert a new model. Opting for the latter you will be requested a few details to identify the model. Firstly a model name and a version number, which will be used solely to label the model for users. The third field, the directory name, needs to be a non-existing directory and cannot contain spaces, punctuations, etc.

Confirming on **Ok** will create a new directory (or a warning in case of error) with all the necessary elements required for an "empty" LSD model program. The most relevant is the file that LMM will use assuming it contains the code for the equations of the model. The filename is conventionally labelled as `fun_XXX.cpp`, where `XXX` is the model directory name. Users may change the file name, but this is potentially dangerous since LMM considers only changes to the equation file to be included in the model.

---

[1]For non-Windows users is also necessary to compile the `lmm` file. See the instructions on the installation in this document on in the accompaning files. For these system the system is launched directly the `lmm` executable

[2]The browser can be activated, besides at the start time, by choosing the LMM command **Model / Browse Models**.

At the end of this procedure the the LMM editor window shows the equation file for the new model. Although this is pure C++ users are invited to use keywords specifically designed to make simpler the expression of the most frequently used LSD expressions.

After the successful creation of the new model, LMM shows in the top bar the reference of the model (label and file name). Moreover, it automatically opens the equation file name, assuming that the user needs to start from there. The equation file name appears in the LMM editor as shown in figure 2.2.
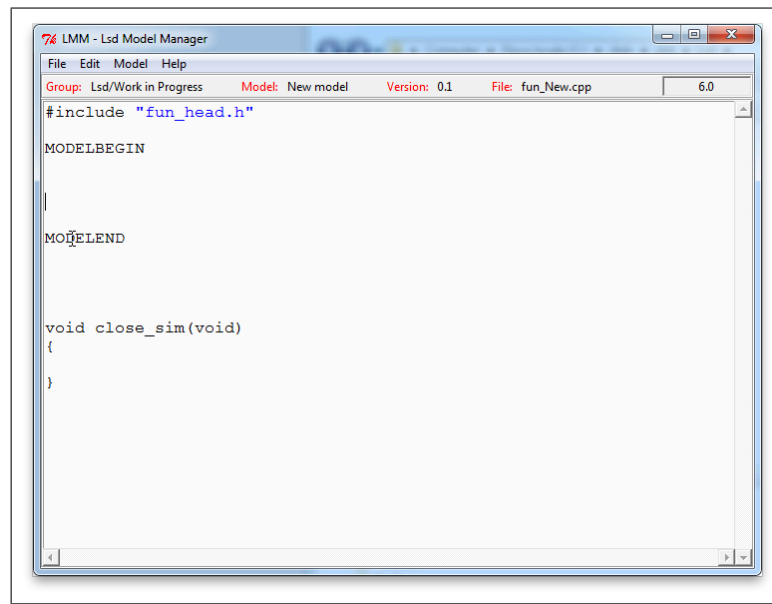
Figure 2.2: Empty equation file for a new model

The equation file contains at the very first line a call to include the definition of all the LSD specific command (file `fun_head.h`). The two following keywords, `MODELBEGIN` and `MODELEND`, are the markers stating the initial and final lines within which the user is allowed to insert the code for the equations. The last command, `close_sim()`, can be ignored for now.[3]

At creation the file contains no equation, but it is anyway technically sufficient to create and run an LSD model program, although, obviously, without equations the LSD model program will not be able to execute a simulation run.

2.2 Introduction to LSD equations

Users define in the equations of the model all the operations that need to be executed by the model. Equations are pieces of code are associated to a label; during a simulation run, whenever the system needs to compute the value for a variable, it passes the control of the program to the equation file. Here the system searches for the piece of code associated to the variable to be computed, and executes its lines, returning to the simulation (i.e. searching for the next variable to compute) at the end of the line of the equations for that variable. Therefore, each equation must necessarily indicate at the very least two basic elements: the name of the variable it refers to, and the final value to be used as result of the equation execution. In between is possible, of course, to place any

---

[3]TECHNICAL NOTE: In the rest of the text notes like this one will report on technical aspects, of possible interest for advanced users, but not relevant for the standard use of LSD.
Many of the commands used in LSD equations are obviously not 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 initialize 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. Obviously, all memory used by the system is automatically dealt with by the system.

number of intermediate lines containing legal C++/LSD commands, typically used to elaborate the value to be returned.

If LMM is not showing the equation file, use the menu **Model/Show Equations** to have LMM re-opening the equations' file. It is very important that you never open 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, which keeps on using the old, un-edited, equations' file, and therefore the LSD model program will not include them. When LMM is requested to show the equations' file it reads the name of the file used to create the LSD model program, and therefore there is no risk of editing the wrong file.[4]

Place the cursor of the LMM editor in any point of the file 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. The equation file is a simple text file, so that any text can be simply typed in. However, typographical mistakes are very common and their correction time consuming. Hence, LMM offers the opportunity to activate small interfaces that ask for the model-specific information, and then automatically insert the (error-free) text for the most commonly used commands. These interfaces are called *scripts* and are available for several commands, including all the variations for specific options.
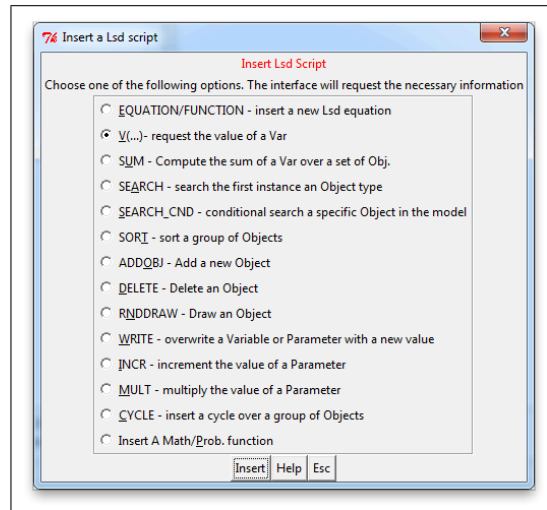
Figure 2.3: List of the scripts available to automatically insert LSD commands into the equation file.

To use a script place the cursor where you want to insert the text (in our case, anywhere in between the keywords for the start and the end of the equations, adding lines as necessary. You can then choose three different ways: menu **Edit / Insert Lsd Script**; click with the secondary (right) button of the mouse; press with two fingers the keys `Ctrl+i`. In any case you will be shown a list of the available scripts available, as in figure 2.3.

In our case, we need to insert a the code for a new variable, thus select the first option in the list (or press the key **e**) and press the key **Return**, or click on **Insert**. Notice that almost every operation necessary to write the equations can be performed *without* the use of the mouse, but only using the keyboard. Getting experienced in using the keys and the available shortcuts increases dramatically the speed of typing the equations in respect of using the mouse.

Following the choice of inserting a new equation you will be asked to type the name of the variable (choose X). Notice that focus is already in the cell for the variable name, so you just need to type the variable name. Pressing **Return** the focus is shifted to the button **Ok**, so another hit on the same key will conclude the script operation.

The script provides with a skeletal (and yet incomplete) code for the variable X.

---

[4]Users can modify the name of the equation file changing the content of the compilation model options, using the relative option in menu **Model**.

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

RESULT( )
```

The line `EQUATION("X")` marks the beginning of the equation, that is, the start of the code that any variable labelled X in the model will begin to execute whenever it needs to compute a value.

The editor places the curse so as to invite you to type in the lines immediately below the header. Any test inserted in between the symbols `/*` and `*/` is considered as comment in C++ (and highlighted in green in LMM) and therefore ignored by the program. This specific comment is particular relevant, because it is considered as the "official" documentation for the variable, and the system will automatically use this comment here any time the documentation of the variable is requested.

The last line, `RESULT()`, indicates the end of the equation and must be assigned, within the round parentheses, the numerical value returned by the equation and assigned to the variable.

Let's write the possibly simplest equation. In between the parenthesis of the `RESULT()` type the command `RND`. This is a LSD command producing a different random value drawn from a uniform random function between [0,1]. Therefore, the complete equation's code is:

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

RESULT(RND)
```

When an equation file is edited it must explicitly be saved, since the compiler does not read the editor window, but only the content of the file. Therefore, save the equation file (menu **File/Save** or shortcut **Ctrl+s**) and, to compile, use menu entry **Model / Compile and Run** (shortcut **Ctrl+r**). This will create (and execute) the new LSD model program containing the new equation.

If you try to compile without saving, the system asks whether you want to continue without saving, or if your want to save the current file before compiling.

At this point LMM should show the new program just compiled (figure 2.4), which is automatically launched. 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. Read the newly appeared **Compilation Results** window for indications on the line number where the error is located. See also the appendix 7.2 (pg. 142) for help on fixing the error.

2.3   Defining LSD model elements

All LSD model programs are externally identical, because their difference consists only in the equations they embody that have any visible effect on their interfaces. The LSD model program we just produced is able to compute a value for a variable labelled X. However, this potentiality can be exploited only defining a model where an actual variable X is defined[5].

The LSD Browser provides the interfaces for creating the elements of the model, but, before continuing, let's give a look at the browser.

The LSD model program browser (figure 2.4) is a window showing the content of one type of object in the model and providing access to all functions of a LSD simulation model, apart those concerning the computational content (i.e. the equations). The main part of the windows is composed of two lists, empty at the start. The one on the left shows the list of numerical elements (variables, parameters and functions) of the object pointed to by the browser. The right-hand list contains the labels of the objects contained in the object shown, again empty at the start. The only object necessarily present in any LSD model is a slightly special one, called **_Root_**. This object is the only one that cannot be multiplied in multiple copies, like any other one. Moreover, it is

---

[5]Of course, it is possible to write code for variables that are not used in a simulation. These pieces of code will not have any effect on the simulation runs, since they will never be activated.
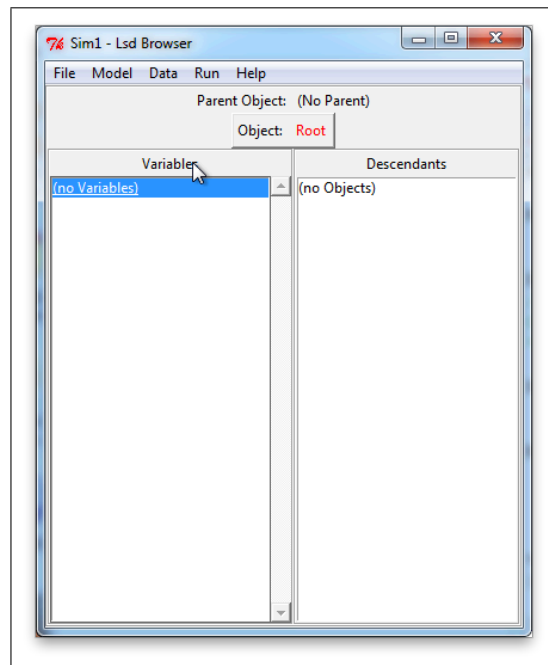
Figure 2.4: LSD model browser, the program embedding the equations of a specific model.

always the first object at the beginning of a time step to be scanned in search of variables needing updating. Hence, any variable placed here will surely be the first to be computed.

Just above the lists of elements are located the name of the object shown (**Object: Root**) and, one position above, the name of the object whose currently shown object descends from (none, as the **Root** is conventionally assumed to be the only object not contained in any other object).

Finally, the menu bar contains the sets of commands for the operations available: file management, model structure, setting or observing data, executing simulation runs, help.

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**. Therefore, at the moment, it shows only one object, the just created **MyObj**.

When naming elements of LSD models users can use any printable character, and the capitalization is relevant, so that **MyObj** is different from **myobj**. It is forbidden to use points, spaces, quotations and other word-separating characters.

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's symbol; you can just use the arrow keys on the LSD browser 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 click on this label to "move up" the browser showing the **Root** again (or press the letter 'u').

Eventually, we managed to have the Browser showing the content of a just created object, which is, obviously, empty[6]. 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

---

[6]Be sure that the browser points to the right object. If, by mistake, you add elements to the wrong objects it is possible to shift en element to a different object. If this is the case now, however, you better start the process from the scratch. Empty the model (menu **File**) and create again the object.

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)** and functions by letters **(F)**.

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 can assign a different file name using menu **File/Save as**.

2.4 Running LSD simulations

If you have executed all the steps described above, you can now 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 over-writing possible configuration files having the same name[7]. Pressing **Ok** will start the simulation.

The Log window shows a message on a new 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 value from the simulation 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.

The default option when defining a new variable is that its values need not to be stored for post-simulation analysis. Every value produced by a variable will therefore be stored in memory for the time strictly necessary to complete a time step, and then it is cancelled. This ensure that large models can be simulated without limitations due to memory constraints. Obviously, some of the data produced in a simulation can be saved, but the modeller needs to explicitly signal which element are relevant to be saved. Thus, we need to repeat the simulation after having set the option to store all values produced through time from variable **X**.

The repeat a simulation run we need to reset the configuration stored in memory from the status at the end of the simulation run to that used at the beginning. This is necessary because LSD prevents the direct continuation of a simulation, re-starting a run after it was completed. But since it saves any configuration before a simulation run, we can easily re-load that initial state of the model without the need to re-define the model from the scratch. To do so you may use the option **File / Load** and choose the only file with the extension `.lsd`. Since the name of the configuration we want to load is the same as the one previously loaded, you can also use the command **File / Reload** (shortcut **Ctrl+w**).

Notice that you really want to continue a simulation after it ended, you can save the configuration at the end of a simulation run and load it as any other configuration.

With a fresh configuration we can not tell the system that we want the results saved for post-simulation analysis. 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. 2.5.

This window provide access to a set of options concerning a variable. For the time being we focus on the three checkboxes after the header with the name of the variable: **Debug**, **Save**[8] and **Run Time Plot**. Check on all the three of them and click on **Continue** to return to the Browser.

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

---

[7]As a rule the configuration of a model is saved in a file before executing the simulation. This avoids that possibly interesting results cannot be reproduced because one forgets the configuration that produced them. However, this rule risks overwriting previous files, hence the warning message.

[8]There are two **Save** options. The second, saving data in separate files, is rarely used and is ignored in the tutorial for brevity.
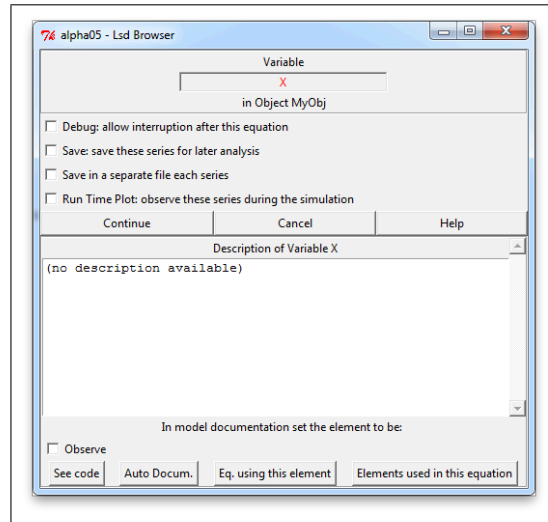
Figure 2.5: LSD variable options

plotting window (**Run Time Plot**)[9]. Notice that the two options are independent, so that we may save the results of a variable without plotting its values at run time, or, viceversa, observing its values without saving them for post-simulation analysis.

2.5 Results of LSD simulation runs

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. Being a very short simulation it is likely that before being able to see it the window containing the graph will be covered by the LSD browser. If necessary, use the icons to highlight the window, or minimize the LSD browser.

The Run Time plot window is meant to provide a quick approximation of the simulation results. A more precise presentation of the data produced during a simulation run is produced with the Analysis of Results module, which is devoted to produce a detailed analysis of all the data saved from a simulation run. To access the module from the Browser, choose menu **Data/Analysis of Results**, and the window shown in figure 2.6 will appear.
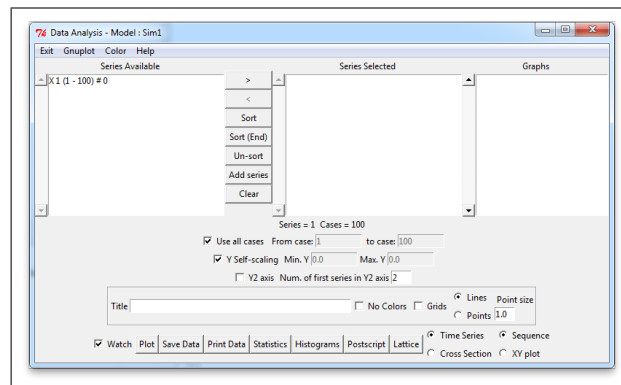
Figure 2.6: Analysis of Results window

This window allows several ways to present and elaborate results.

---

[9]The **Debug** option allows to control the internal computation of X during a simulation run. We will see that later.

The main body of the window is composed by three lists: **Series Available**, **Series Selected** and **Graphs**. 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.

The series available (in our case only one) are indicated by their label and other indicators, that we ignore for the moment. Move the only series in the leftmost list (**X**) to the the middle list double-clicking on it. Leave the default options as indicated. Now 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 area under the pointer; when the pointer crosses the line of the graphs the label of the variable appear in the left-bottom corner; double-clicking any part of the graph the main control window comes to the foreground.

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 many graphs). Press the button **Statistics** and observe the **Log** window (search for its icon in the icon bar of the screen). You can see some descriptive statistics concerning the series selected.

The Analysis of Result module in L^{S}D 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 L^{S}D. 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 L^{S}D model program, and return to LMM to update the equations for a new L^{S}D model program.

2.6 Extending a L^{S}D model equations

The equation we have written is pretty basic. We have defined **X** as a variable returning random values in the range [0,1][10]. 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:

$$X1 = minimum + (maximum - minimum) * X \tag{2.1}$$

Equation 2.1 shows that when $X = 0$, than $X1 = minimum$; if $X = 1$, than $X1 = minimum + maximum - minimum = maximum$; for intermediate values $X1$ varies proportionally to X.

Let's implement an equation for **X1**. If you did not do that yet, close the L^{S}D 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. L^{S}D determines, during a simulation, the order of execution of the different variables, so the order in which their code appear in the equation file is irrelevant.

As done before, choose **Edit/Insert L^{S}D 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[11], and then make logical or mathe-

---

[10]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 6.8.8 (page 126) describing the random functions available in L^{S}D.

[11]From now on we will consider functions as equivalent to variables, unless otherwise specified.

matical elaborations on them. To retrieve the value of an element and use it within an equation we need to use a LSD function called `V("label")`, standing for `V`-alue of the element with label "*label*".

Using the `V("...")` function, the equation for **X1** may be written as:

```
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, writing the elaboration in the `RESULT(...)` line is hardly a practical strategy for even minimally elaborated equations. For longer computations a single line will not be sufficient, and anyway the code will be un-readable. It is therefore a good practice to structure the code for the equation with a programming style that minimizes the computations required and improves the clarity of the code.

A more efficient style of coding (which becomes a necessity as soon as the equation is even slightly complicated), consists in collecting initially all the values necessary for the computation and storing them into local C++ variables, which are then used to perform the actual computations, possibly with many intermediate computations.

Temporary variables are repositories that live only within the computation of one single equation, and are re-created any time a variable starts to execute its equation[12]. In the equation file modelers 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:

```
EQUATION("X1")
/*
  A random value within 'minimum' and 'maximum'
*/
v[0]=V("minimum");
v[1]=V("maximum");
v[2]=V("X");
v[3]=v[0]+(v[1]-v[0])* v[2];
RESULT( v[3] )
```

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. Then we assign the result of the operation to a fourth local variable, `v[3]` and, eventually, its value is used in the result line.
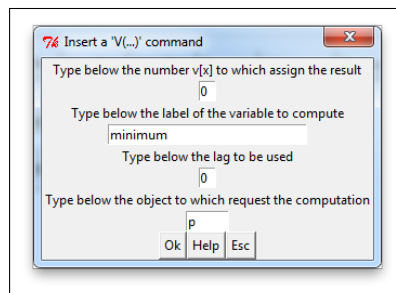
Figure 2.7: LMM script to insert the equation function `V(...)`.

To insert the `v[0]=V("minimum");` line and the others we have available a LSD script, which avoid trivial, but time-consuming, mistakes and keep tracks of the different local variables used. 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

---

[12]This point should be clear. The temporary variables we are going to use cannot be used to transfer values from one equation to another. They are strictly local storage places that needs to be loaded and can be used only within one equation.

`v[...]` and the type label for the value to request, that is ***minimum***. Leave the other two options to the default values. Notice that the script moves the focus sequentially through the different fields; you can quickly move the from one field to the next by pressing the **Enter** key without ever using the mouse.

Repeat the script for all the three elements, inserting a new line after each line is inserted. Notice that there is no difference in the equation code between using values from parameters or variables. Actually, a good modeling style consists in implementing earlier versions of the model with all parameters and only one or 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, even if the internal cycle of computations necessary to update all variables will be different, but this a task performed automatically by the system.

After having concluded the coding save the file with **File/Save** (or using the shortcut pressing at the same time the keys **Ctrl+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 will issue a warning. A new window will contain the message indicating approximately where the error has been found, so that you can fix it (see the section 7.2 on compilation errors.).

When a new LSD model program appears after the compilation we are ready to update the model configuration adding the new variable and the two parameters.

2.7 Initializing LSD elements

The new LSD model program is able to compute a new variable, ***X1***, but, for this having any effect in a simulation run, we need to add this variable, and the necessary parameters, in the model configuration.

First of all we need to retrieve the model configuration we have defined before: use **File/Load** and choose the LSD file (called `sim1.lsd` unless you did not change the default name). We need to add three elements to the model: parameters ***minimum*** and ***maximums***, and variable ***X1***.

Move the Browser to show the ***MyObj*** object and then add 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 for that element (see fig. 2.5 at page 22). In this window you can double-click on the red label of the element in the upper part of the window and a new window will appear, as shown in figure 2.8. This window allows to change the label for the element (assign an empty label to remove it altogether), change its position in the object structure, or change its nature (e.g. from parameter to variable).
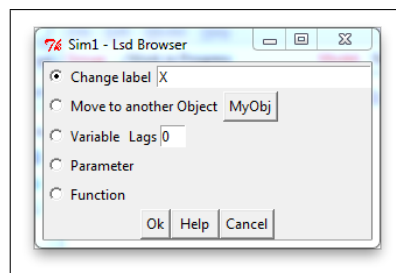
Figure 2.8: Options to modify an element. It is possible to turn the element into a parameter, variable or function; change the label of the variable; delete it altogether (assigning as new label an empty string); move it to a new object.

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 (try it, and then re-load the configuration to return to the current configuration).

To assign a value to the parameters in an object[13] 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. 2.9.
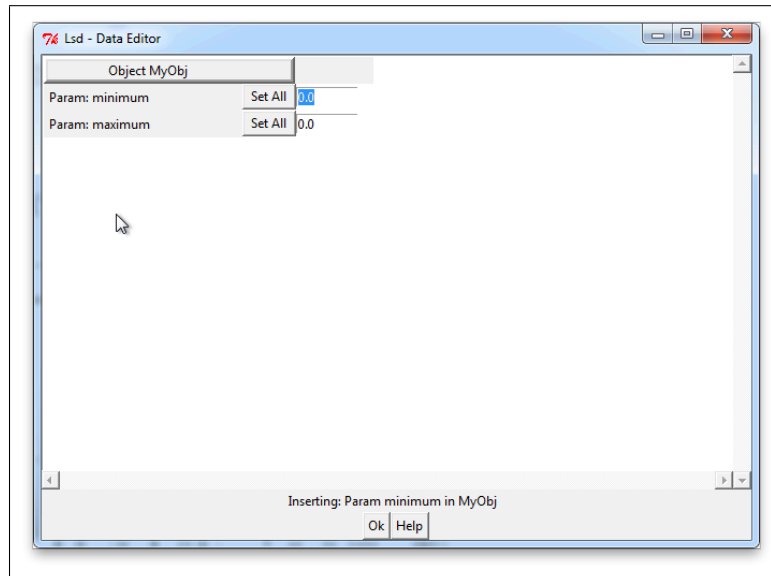
Figure 2.9: Initial values for **MyObj**

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 in the Browser 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 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, as expected, their range differ for the scale.

As a matter of exercise, we can produce a scatter-plot graph, where two values of the two variable at the same time step correspond to a point. In the Analysis of Results window check the box **Time Series** and the box for **XY plot**. Choose also the option **Point** to generate a graph with dots and without a line interporlating the points. Insert in the **Series Selected** list firstly the variable **X** and then **X1**. After clicking on the button **Plot** a new window asks whether you want to generate a high-quality or low-quality graph; press **No** and a new graph will appear[14] 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.

After having produced the graph, exit from the analysis of results module and return to 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 **Ctrl+w**.

---

[13]LSD permits to assign initial values only to the elements of one type of object per time.

[14]Scatter plot graphs are generated using an external package, Gnuplot. LSD allows either to incorportate the graph as image within a LSD graphical window, or to use a Gnuplot window. In the former case you loose some of the quality, but you can manage the window as any other LSD window. Gnuplot windows are instead managed by its own commands. Test both options, to appreciate the difference.

2.8 Setting the number of objects

We have worked with one single copy of the **_MyObj_** object. The main advantage of objects is that they can be multiplied in as many copies as desired, generating automatically all the copies of their content, including, if present, sets of copies of contained objects.

Open menu item **Data/Set Number of Objects/All types of Objects**. The Browser window will become as in fig. 2.10.
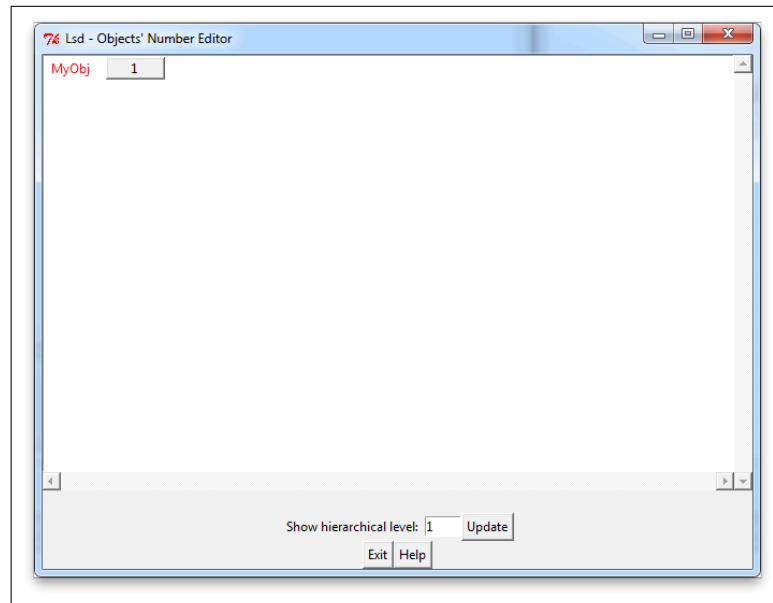
Figure 2.10: Setting the number of Objects

This table shows only one line, since there is only one type of object. 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. When adding new copies one generally is interested in changing the initial values of their elements.

2.9 Initializing multiple elements

Open the menu **Data/Init. Values** to control for the parameters' values[15]. Given the number of copies of **_MyObj_** now there are 10 copies for each type of parameters. Though it is possible to insert manually new values in each cell, the process is impractical as soon as there are a few tens of elements to initialize. To avoid this tedious work LSD offers the possibility to use one of several automatic functions to generate values for each line, that is, for each type of element to initialize.

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 **_minimum_**. Check on the option **Increasing**. In the box labeled "Numerical data ..." there are two entry cells: type in the top cell, marked as **Start**, -100 and, in the second entry marked **Step** enter 10. Press **Ok** to confirm and exit. Now **_minimum_** is set to -100

---

[15]If the initialization window is empty check that the browser is showing the right object. Initialization can be made only for one type of object.

for the parameter **minimum** in 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 the ten copies of **X1** oscillating on decreasing ranges.

2.10 Plotting multiple series

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. Run the Analysis of Results after the simulation. You will see that there are now 10 series for each variable saved, each identified with a different number placed after the name of the variable. You can choose as many variables you want to plot, by selecting them and then clicking on the > button, or double-clicking on each of them. The selection rule respect the usual criteria:

- Click and drag: all the variables touched will be selected;

- Click on one series, keep key "Shift" pressed and select another series. All intermediate series will be selected too.

- Click on several series keeping key "Control" pressed. Every series will be added to the selection.

As you can see, the series in the **Series Available** are listed according to the sequence of objects containing them, so that variables **X** and **X1** are listed alternating. Since frequently we need to select all the copies of one type of variable, the standard selection rules are rather cumbersome. There are other two ways to perform this type of selection quickly and effectively.

Firstly, it is possible to click on the button **Sort**. This will rank the series according to their increasing alphabetical order, therefore placing together all variables with the same label. Secondly, we can use a rather sophisticated selection function, which is extremely helpful when dealing with many thousands of variables.

Move the pointer of the mouse over one of the variables you want to select, and click with the right button of the mouse. A new window will pop up, as the one shown in figure 2.11.
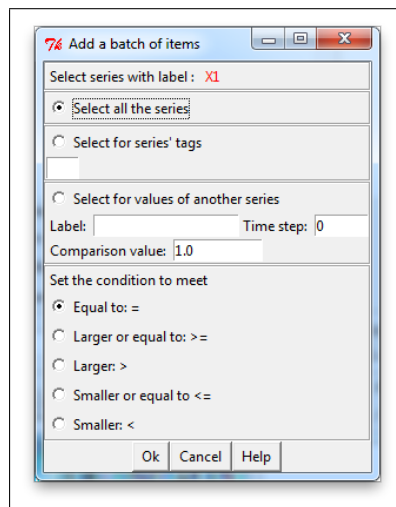
Figure 2.11: Selecting function in Analysis of Results window.

This windows allow to select and move into the **Series Selected** box all the series with a given label, respecting certain conditions. Let's ignore the options available (or see the **Help** button for details), and leave selected the top-most option, **Select all series**. Press **Ok** and all the series with the label will be moved into the central box. Pressing the button **Clear** will empty the **Series Selected** list box.

2.11   Statistics

We can now press the button **Plot** and generate a graph including all the series selected. The initialization we gave to the parameters let us expecting that the oscillations of the variables in the earlier objects should larger than those in the later ones. However, the graph shows independent random series, and therefore it is difficult to individuate whether the expected property does actually occur.

The easiest way to assess the properties of the series is to check the variance of the different series. Click on the button **Statistics**, and search for the **Log** window. The window will contain one line of statistics for each series selected, indicating: label and object's indicator of the series (along with the number of data considered); average value; variance; minimum; maximum; and standard deviation.

As expected the average values are all around zero, the expected values of all the random variables, while the variances decrease for the increment of the objects' indicators.

2.12   Comments on equations' code

The results we have obtained are quite obvious: each of the 10 copies of the object computed the equations producing independent results. LSD automatically induced that the copy of the variable **X1** placed in the ith object had to use the parameters contained within the same copy of the object, dispensing the modeller to insert redundant information as an index i.

This feature is extremely useful when the object structure of a model becomes even slightly elaborated, with many layers of objects. When an equation is computed the same code must be exploited by all the variables with the same label, included in different copies of the same type of objects. How can we be sure that the equation makes use of the "correct" elements, when the model contains many copies of each of them?

LSD "knows" the copy of the object containing the variable under computation. By default any element appearing in the code of the equation is searched within the same object containing the computing variable. If the required elements are not there, then LSD moves on to search in "nearby" objects, continuing until the whole model is scanned. This feature has many useful consequences. For example, as we have seen, the modeller needs not to specify within an equation where the elements required are stored, and the same code can be used in several object structures. Obviously, the modeller can, if necessary, force LSD to make use of a specific element, by indicating which object should be searched for an element, though this is rarely needed.

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("...") returns those copies. For example, the copy of **X1** in the 3rd object will compute its equation making use of the copies of ***minimum***, ***maximum*** and **X** contained in the same $3^r d$ object. As we will see, programmers have a wide variety of options to write code that makes use of values from users' specified 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.

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

2.13   Simulation 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. 5.6.

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
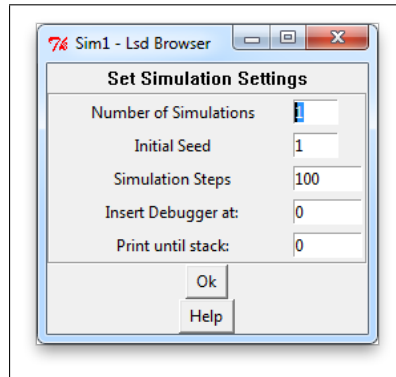
Figure 2.12: Setting simulation options: number of simulation runs, time steps per run, pseudo-random values, and debugging options.

after the simulation run. However, in certain cases one may be interested in knowing how the model behaves repeating several times the simulation using different random values, to test the robustness of some result. In these cases users can set the number of simulation runs to be higher than 1. The results of each simulation run (i.e. the data produced during the simulation by the variables marked to be saved) are stored into files (with extension .res) at the end of each simulation run. Moreover, the system generates 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 any 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 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 56th 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. 2.5 at pag. 22) completes its equation[16].

---

[16]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, and does not allow to access model data. Instead, using the LSD debugger is simpler and gives full access to any data of the model, including using the Analysis of Results module on the data produced up to the time of interruption.

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

2.14   Using lagged variables

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

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

That is, the value at any time steo 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[17]:

```
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 at n time steps before, lags. Note that `V("Var")` can be expressed also as `VL("Var",0)`.

Let's add this variable to the model we already implemented. 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. 2.13.
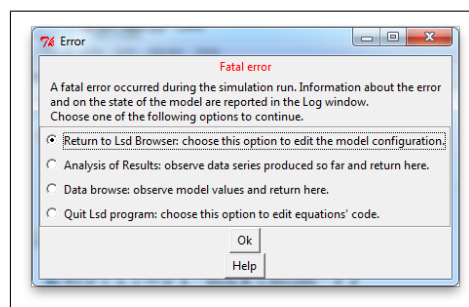
Figure 2.13: Simulation aborted for a serious logical error.

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

---

[17]At this point we assume the reader is able to remember how to use the menu **Insert LSD Scripts**.

legal from the computational viewpoint, but, in the specific context of model status, prevents to continue the simulation. 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 intuition let's follow the indication of the error window and read the messages in the log window:

```
...

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 suspects.

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 a time step is completed 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 indicate the correct lags for the variable at the time of its creation, 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. 2.13. We can:

- **Return to LSD Browser**: return to the Browser, terminating the simulation at this point.

- **Analysis of Results**: open the analysis of results window to analyse the data produced so far. understand why a model

- **Data Browse**: 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.

- **Quit the LSD program**: kill the LSD model program altogether.

Let's return to the browse and re-load the configuration, so that we can edit the new variable increasing the lagged values. 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. 2.8, pag. 25).

Select the option **Variable** and modify the number on the cell after the label **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.

Before being able to start a simulation run we still have to provide one piece of information. Having added a lag to a variable we need to tell the system which values it needs to use at the very first time step, that is, computing when computing the values $RandomWalk_1 = RandomWalk_0 + X1_1$. Since there were no time step 0, these values must be provided by the modeler. They will obviously affect the results, since they indicate the starting point of the random walks.

$RandomWalk_0$ 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[18].

Let' 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 (menu **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.

2.15   Multi-layered object 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, say **Demand** and **Supply**. Therefore, **Market** is "higher" in the hierarchy, representing a more aggregate element than **Supply** or **Demand**. In turn, **Supply** may 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 used so far is contained within another object in which to compute aggregate statistics, like the average value for all the random walk variables[19]. 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 **AggregateObj**. After confirming, verify that your model now includes **Root** containing **AggregateObj** 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 the newly created **AggregateObj** object type. On the right of the line for the new object there is a text signaling that other objects are contained within those objects. Click on the text to ask the system to show the additional objects[20]. and the window will create a new indented line.

This new line shows firstly a symbol '**+**', indicating that it is one layer below the objects at the highest layers, conventionally referring to the objects contained in **Root**. It will then indicate the name of the object the line refers to (in red), and the name of the copy of **AggregateObj** containing the group for the line. Eventually, we have the number of copies for the whole group, the button labeled with 10.

Click on the red label **AggregateObj** of the only line for this type of object and type 3 to the resulting window. After the confirmation you will be now shown three distinct groups of **MyObj**, one for each copy of **AggregateObj**. Notice that each group has exactly the same number of copies.

LSD constraints each copy of an element to have all the same structure, although the actual numerical content may obviously vary. We therefore are forced to have at least one descending object **MyObj** within each copy of **AggregateObj**. When creating new objects, as we have just done creating 2 additional copies of **AggregateObj**, the system takes one copy as templar to determine the numerical content (e.g. parameters and number of descending objects) of the new copies. Hence, having only one group, that one had be replicated.

You can change independently the number of copies for each of the groups of **MyObj**'s contained within each of the 3 copies of **AggregateObj**, or modify at once all of them. Click on the button with the number 2 of one of the lines for **MyObj**, for example the second. A new window will appear, like that reported in figure 2.14

---

[18]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 $RandomWalk_0$ and another for $RandomWalk_{-1}$.

[19]Actually, we already have a higher level object, **Root**. But, as already said, it is better to never use **Root** to contain any element but objects.

[20]Alternatively, you could type '2' into the entry for the hierarchical level to show and click on **Updated**.
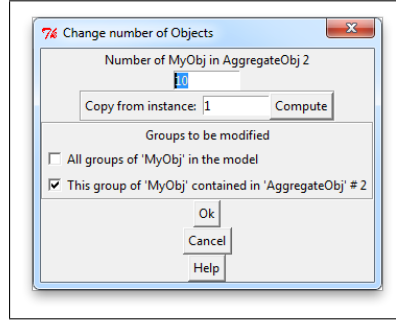
Figure 2.14: Setting the number of descendants. It is possible to modify a single group, or all the groups in the model. Moreover, it is also possible to choose which instance of the existing objects to copy initial values from.

This window refers to a specific group of objects **MyObj**, namely, the group contained in the second copy of **AggregateObj**. You can decide to modify only the single group of objects, or affect all groups of **MyObj**. Notice also that when you decrease the number of elements in a single group you have the possibility to decide individually which copy in the group to remove. If this does not matter, by default LSD will remove the "leftmost" objects, that is, the final ones in the group, under the assumption that they are those being added more recently.

Using the features of this window generate three 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**). We need to write the equation for this new variable, and then re-create a new LSD model program replacing this one. However, for the time being save the configuration (**File/Save**) but keep the program running. It is easier to write the equations having at hand the model structure to visualize. When the equation will be ready we can close the LSD model program and compile a new one with the new equation.

2.16 Equations for multi-layered models

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**:

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

where the index i refers to a succession of copies of the objects **MyObj**, and n to the total number of copies.

However, in LSD we don't make use of indexes, nor we need to know the number of copies before starting the computation. From the programming viewpoint we have the problem of ensuring that each copy of **AverageRW** is computed using 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 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 simple and intuitive way. The operations we are going to write is the computational equivalent of the

operations described below. It is anyway necessary define an element of programming language: the *pointer*. Pointers in programming are a special type of variables. A common variable is a label associated to a numerical value, and programmers can assign a value to the variable in a certain line and use this value later on. Pointers have the same function not for numerical values, but for objects: they are variables for objects, but numerical values. Hence, you can assign an object to a pointer, and the use the object *pointed to* by the pointer.

In writing the equations LSD offers both local variables (`v[0]`, `v[1]`, etc.) and pointers, called `cur`, `cur1`, etc. Hence, the pseudo code for computing the average of values in a group of objects is the following:

1. define two variables, `v[0]` and `v[1]`, both initially set equal to 0;

2. define a pointer `cur` and set it to point to the first **MyObj** in the group;

3. add the value of **RandomWalk** for the object in **cur** to `v[0]`;

4. add 1 to `v[1]`;

5. are there more objects, after the current one?

   - Yes: replace as `cur` the object next to its current copy, and go to step 3.
   - No: go to step 6.

6. assign to the variable as result the division `v[0]` / `v[1]`

In slightly more formal notation figure 2.15 describes the same operations.
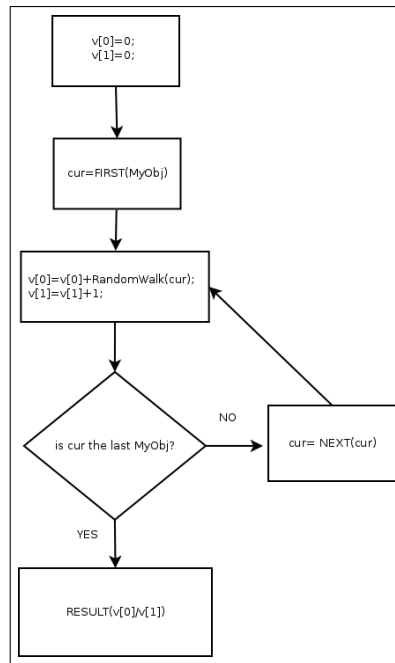
Figure 2.15: Flow chart to compute an average. Command `FIRST(cur)` indicates the first object in a group and `NEXT(cur)` replaces the pointer pointing to `cur` with its following object, like X_{i+1} in respect of X_i with vector notation.

The procedure defined by the pseudo-code above computes the average value sequentially, scanning one element per time and cumulating two sets of values for the denominator and numerator of the division.

In LSD equations commands like this are very frequent, thus the expression is very simplified. 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 **AggregateObj** 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 as many times as many copies **MyObj** can be found. Notice that the LSD "knows" the object containing the variable that it is computing at any time; call this object the object "under computation". LSD is therefore able, using the command `CYCLE`, to scan all and only the objects descending from the object under computation, ignoring the copies contained in other copies of **AggregateObj**. Therefore the number of repetitions depends on the number of copies of the objects whose label is **MyObj**[21]. During each repetition the cycle assigns to the pointer `cur` a different copy of **MyObj**.

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 copies of **MyObj**. The first line cumulates in `v[0]` a more complicated stuff: `VS(cur,"RandomWalk")`. This command 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. The difference with the general form `V("...")` is that the simple form assumes by default that the object where to search the element to evaluate is stored in the same object[22].

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 **AverageObj**. In these objects there is no variable **RandomWalk** so the system should start a search for an object containing the desired variable. But all the descending objects **MyObj** are equally "distant" from the object **AggregateObj**, where the equation for **AverageRW** is executed, and therefore the system cannot correctly find any specific copy of **MyObj**[23].

Using `VS("...")` 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" sequentially to all the copies of **MyObj** descending from the **AggregateObj** 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.

---

[21]If there are no copies of **MyObj** descending from the **AggregateObjext** containing the **AverageRW** whose equation is executed the internal calculations in the cycle are never executed.

[22]Within the code for equations modellers can access the object containing the variable under computation, represented by the pointer `p`. Therefore, the `V(``X'')` is actually equivalent to `VS(p,''X'')`.

[23]Using `V("RandomWalk")` in an equation for a variable in **AggregateObj** will return always the value of the very first copy of **RandomWalk**.

2.17   L^{S}D Simulation Manager

As we have seen a modeller defines the model structure and the equations, described by chunks of generalized code. However, a program needs a lot more of information to be able to run, determining what the processor must do and when. Users can safely rely on L^{S}D to fill any explicitly missing information required in order to have a simulation running correctly (i.e. coeherently with the model's definition). In particular, L^{S}D ensures that the variables' values used in the equations of other variables without lags (and therefore requiring their equations being computed before those where the present-time values are used), are updated in the correct order. Moreover, L^{S}D ensures that equations making use of elements present in many copies in the model choose the correct values.

L^{S}D exploits a default system to ensure that a simulation run performs the most likely operations, though modellers can, if necessary, over-run the default system. In this paragraph we give a hint on how L^{S}D manages to generate simulation runs out of the information modellers provide. Readers not interested in these technical details can skip this paragraph.

The core of a L^{S}D model program running a simulation is a system, called L^{S}D Simulation Manager or LSM. Its task is to assemble all available information concerning the model and produce an actual simulation run[24].

A simulation run is a sequence of time steps, within each of them all the variables must be updated (i.e. their equation executed) according to a specific order, that is, for example, which equation must be executed at the beginning of the time step and which can be executed at the end. The order of updating is important, since it determines whether the values of a variable used within the equation for another variable are those from the previous time step or those from the current one, and, in general, the results obviously differ.

Some simulation languages requires modellers to define explicitly a *scheduler*, that is, the ordered list of equations to be executed within a time step. L^{S}D uses a different approach, generating on the fly the schedule of the execution of equations by analysing the implicit temporal constraints as can be deduced from the very code of the equations. For this operation the LSM makes use of two bits of information: the general clock of the simulation run (called `t`) and a field contained in each copy of the variables indicating what was the simulation time when the variable was lastly computed, a field called `lastupdate`.

Let's see how the LSM exploits this information to ensure the correct schedule of updating within a simulation step. At the start of a simulation step the first operation performed by the LSM is to increase the time counter of one unit. After that the LSM begins scanning all the objects in the model and, within each of them, controls all the variables contained. For every variable considered the LSM compares the simulation clock with the field `lastupdate` of the variable. If these values are different (that is, `lastupdate` is smaller than `t`) than the clock, then the LSM begins the execution of the equation for that variable, otherwise (`lastupdate=t`) it skips the variable and moves to the next one.

While executing the equation for a variable the code may require the value of other variables in the model, typically through the command `V(...)`, as happens in the computations for **X1**, **RandomWalk** and **AverageRW** in our example model. If the values of the variables is requested with a lag, then the system can use the value as inherited from the previous time step. Otherwise, if the variables are requested as values computed at the present time, they may have it or not.

The LSM is able to distinguish this cases, comparing the lags of the values required in the equation's code, with the time clock of the simulation and the `lastupdate` field of the variables concerned. When a requested variable's value is available LSM returns the value to the equation under computation directly. When the value necessary to continue the computation of an equation is not available, it means that the variable did not computed its equation at the current time step as yet. In other terms, the scheduler should have computed firstly that equation, and later the one it (erroneously) has already started. In this case the LSM "corrects" its mistake by interrupting the computation of the variable to the stage it already reached, and moves to execute the equation for the required variable. When that computation is completed, it then returns to the interrupted

---

[24]Or, in case of errors, interrupt the computation, avoid program crashes, and provide as much information as possible concerning the problem encountered.

equation, provides the updated value of the variable, and continue the computation. When the computation for the equation is terminated, and the resulting value stored in the variable, its field `lastupdate` is updated in line with the clock of the simulation.

Note that the system may iteratively continue for several cycles: even the newly computed equation may require, in turn, other values not yet available. LSM can generate, if necessary, a long chain of half-computed, interrupted equations until it manages to compute the last one, and then rolls back all the interrupted computations completing them in reverse order.

Notice that the way the LSM arranges the order of completion of the equations is independent of the order for the starting the computations of the variables. Therefore it is possible to use any rule for scanning the variables of the model; in practice, LSD begins by (trying to) compute firstly the variables in the higher object (**Root** is always the first). When the scanning encounters a variable that have been already updated, because another variable requested its new value, it does not re-compute equation.

An important property of the LSM is that the actual order is decided at run-time. There may be cases where a variable's equation contains two alternative computations (distinguished by an `IF ... THEN ... ELSE` command), requiring two different order of updating conditional on certain events. The LSM will automatically generate different schedules of updating, depending on which condition applies at every step.

For a similar reason, LSM guarantees that the code for equations can easily be moved across different models. Depending on the equations for the variables within an equation, different models will generate different schedules, without the necessity for the users to pay attention to the schedule.

The LSM allows also to modify, possibly heavily, the schedule of a model in an extremely simple way. For ensuring that one computation (e.g. for variable **Z**) takes place before the completion of another variable (say **Y**), it is sufficient that the very first line of the code for the equation of **Y** contains the command `V(''Z'')`. The LSM ensures that any computation occurring after that line, variable **Z** will be updated, that is, its computation already performed.

Lastly, LSM is able to spot inconsistencies in the model, avoiding never ending cycles where one or more variables require both to be computed before the other. For example, the "models" made of two equations: $X_t = Y_t$ and $Y_t = X_t$ is not computable, in that it would generate an infinite bouncing between the two equations. As we will see the LSM identifies this potentially fatal errors and issues messages helping the modeler to fix the problem.

The equations we have written for our example model are the computational translation of a difference equation model that may be expressed with the following equations:

$$
\begin{aligned}
\text{AverageRW}_t^i &= \sum_{h=1}^{n_i} \frac{{}_t\text{RandomWalk}_t^{i,h}}{n_i} \\
\text{RandomWalk}_t^{i,j} &= \text{RandomWalk}_{t-1}^{i,j} +_t \text{X1}_t^{i,j} \\
\text{X1}_t^{i,j} &= \text{minimum}^i + (\text{maximum}^i - \text{minimum}^i)\text{X}_t^{i,j} \\
\text{X}_t^{i,j} &= r.v.U(0,1)
\end{aligned}
\tag{2.2}
$$

The LSD representation for these equations, as we have seen, can neglect the time suffix t, which is redundant (but includes the lag indication), 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 how LSD replaces the indexing system.

In our case, as in practically any model, there are many copies for each variable, one in each copy for any type of object. Every copy of the same type of variable executes the same computations, though, obviously, make use of 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 an information stored into the variables: which object contains it. When the equation is activated all the LSD functions, like `V("...")`, "know" the copy of 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.

In practice, L^{S}D dispenses from the use of indexes by using the more flexible concept of "membership" to an object. If a variable's equation requires another element the system will provide the copy of the element contained in the same object or, if it is not there, in the "closer" object containing the variable under computation. This system ensures that modellers can avoid to specify exactly where each element can be found, and also allows to use the same code of an equation in different model structures, where the same elements are stored in different objects.

This automatic system to retrieve elements system is useful most of the times, but not always. There are cases where the modeller needs to indicate in an equation a specific copy of an element chosen according to elaborated rules. For these cases L^{S}D provides a set of functions allowing modelers to express any possible strategy.

2.18 Extending the model: quality and sales

We are proceeding gradually adding new equations one by one. This style of proceeding is highly valuable in avoiding "complexity traps" generated when one tries to write a complete model before testing any of its components. L^{S}D favors this style since the simulation program "adapts" the simulation run by itself, depending on new elements introduced, changes in the lags of existing variables, etc. Given the speed with which a run can be produced, possible corrected, and analysed, it is possible to test the effects (and the correctness) of each and every variable added to the model.

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 one **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. In practice, we need two new variables, and their equations, expressing an indicator of competitiveness based on the quality of the firms, and the levels of sales. In formal 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 higher than the average, its relative quality is positive, boosting its sales, otherwise it is negative, decreasing sales.

The second equation states that the level of sales for a firm is inertial, changing slowly in the the direction of the sign 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.

Note that the equation for sales relates sales rate of changes to the relative quality. The absolute level of change in sales depends also on the past level of sales.

Let's see the equation's code for **RelativeQuality**:

```
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 new 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")? This is a variable contained in **AggregateObj** 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 **AggregateObj** copy containing the copy of **MyObj** whose **RelativeQuality** is executed. This automatic retrieval of data in "higher" objects 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 of interpretation:

```
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");
v[3]=v[0]*(1+v[2]*v[1]);

RESULT(v[3])
```

Now we can compile the model and, if there are no grammar error in the equations' code[25], we can use LSD model program to make the necessary modifications to the configuration of the model:

1. load the configuration file (**File/Load**);

2. move the Browser to show the object **AggregateObj**;

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.

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. When a mis-matching error occur you may need to change either the label in the code of the equations or the label in the configuration.

Another frequent error by beginners is to forget that LSD model programs cannot execute the code of the equations merely because it has been written. After coding new equations, the modeler needs to shut down the existing LSD model program and compile a new one. Only after a re-compilation the code for the new equations can be exploited.

---

[25]If there are errors, choose not to run the existing LSD model program and see the second line in the window **Compilation Results**. There should be a line number, indicating approximately where the error has been found.

2.19 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 **AggregateObj** 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, assuming identical initial conditions for all the firms in the model.

Any user of LSD running this model with the above initialization will produce exactly the same results. For example, in figure 2.16 it is reported the series for the relative quality levels over the first 100 time steps.
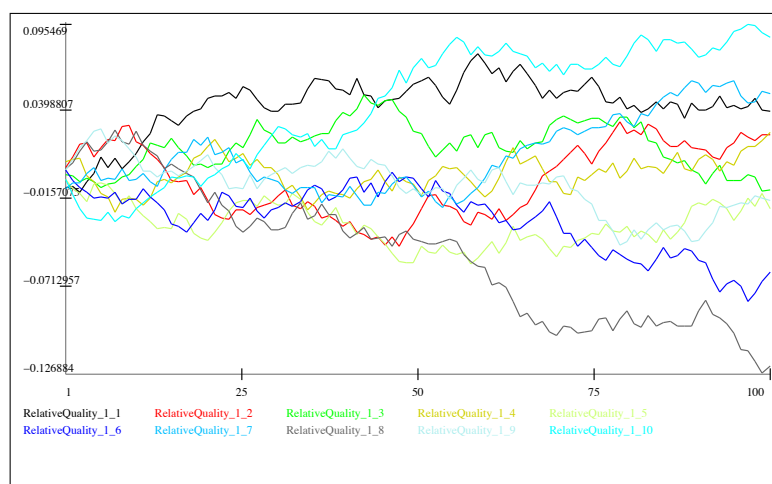
Figure 2.16: Relative quality over time steps from 1 to 100.

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 as to control exactly what happened during (pseudo-)stochastic events. 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**.

We can control whether the model produces, as expected, a relation between the values of sales and **RelativeQuality**. In Analysis of Results select all the **RelativeQuality** and all the **Sales** series. Then check the options for **Cross Section**, **XY plot**, **Points**, and press **Plot**. The new window appearing will ask which time step you want to consider the data, besides other options. Leave the existing default value and press **Ok**. The resulting graph will show the points whose coordinates are given by the values of the couples of **RelatieveQuality** and **Sales** for each **MyObj** at time step 100. As we can see, the points indicate a roughly increasing relation, as expected.

Is this model a good representation of the purported effect of quality on sales? If we analyse the model results we see that, actually, something is wrong, if not technically (the model does exactly what we told it to do), but the "scientific" interpretation of the results is inconsistent.

In fact, the equation for sales is implemented as a function of the *relative* quality, in that, implicitly, we would like to have some firms increasing their share of the market and some decreasing them. This representation would implicitly suppose that the total level of sales, the dimension of the market, would remain constant. Is this the case?

Though a mathematically skilled reader would immediately induce the answer from the functional form of our variables, the laziest may instead prefer to exploit the model to investigate the matter. For example, we may try to generate the increments in sales of the different firms, $DiffSales_t = Sales_t - Sales_{t-1}$. We know how to introduce and write the equations for such a variable. However, let's use another way, to introduce a new command for the writing of LSD equations.

When the model computes the level of sales we already have available the past level of sales, so that we may directly compute the values for the variable we are interested, call it ***DiffSales***, during that computation. Though LSD forces the code for a variable to generate only one single result, it is possible to use a special command to write values on any element of a model. Consider this modified code for the equation for ***Sales***:

```
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");
v[3]=v[0]*(1+v[2]*v[1]);

WRITE("DiffSales",v[3]-v[0]);

RESULT(v[3])
```

The new version of this variable does not modify the result produced by for the variable ***Sales***. But, although obviously an equation can return a single value for its variable, the code for the equation may contain any C++ (and LSD-specific) statement. In the above example we requested the equation for sales to write a value (the difference between the lagged and current sales) onto an element of the model, called ***DiffSales***. We can now introduce a parameter in ***MyObj*** labelled ***DiffSales*** that will contain the sales' differences for each firm in the model.

It is worth to notice that the use of a WRITE command is, in general, non necessary, since a variable with a specialized equation may do the same job. Generally, the reason for using the command WRITE(...) is to speed up the implementation of the model, or the execution of simulation runs. This command needs also to be used with caution because it does not allow the automatic management of updating. In fact, being a parameter, LSD does not "know" the time when ***DiffSales*** is written; therefore, if the model were to use the value of ***DiffSales*** in other equations, it would be necessary to manually ensure that the values are used in the correct way (before or after it is updated by equation for ***Sales***). In our case we do not run any risk, since the parameter is used only for statistical purposes and does not affect other results of the model.

Let's compile the new simulation model program and load the existing configuration. Move the Browser to show the object ***MyObj*** and add the parameter ***DiffSales***. You need to open the interface to initialize the elements of these objects (menu **Data/Init. Values**) because, being a parameter, LSD requires it to be initialized, even though the initial values are never used in the model. Just open the initilization window and close it, which is sufficient to signal that the default value (zero) is accepted by the user. Remember also to check on the option to save these values, by double-clicking on its label in the Browser.

2.20   Generating new series

Run the simulation and open the Analysis of Result module. You will find the series for the new parameter ***DiffSales***; select them and plot their results after having reset the options to the default values (**Time Series**, **Lines**, **Sequence**). If the assumption that any increase in one firm's sale should be matched by an equal decrease by other firms' sales were true, the average value of ***DiffSales*** across all ***MyObj***'s should be null. Is this the case? The graph is not conclusive in this respect, so we need to find a more precise tool. We may re-run the same simulation adding a new equation computing the average, but here we use a different method: generation of new series from those saved in Analysis of Results.

The module we use so far only to to show the results saved from simulation runs has the possibility to create new series from manipulating the available series. Clicking on the **Add Series** button in between the two main listboxes list the possibilities offered to add new series. Choose the option to **Create one series ...** and you will be shown a window as in figure 2.17.

Figure 2.17: Generate new series as elaboration from the currently selected ones. The new series will appear as a single statistics computed over time (first option on the top) or across series (second option). Choose the statistics to be computed, the name of the new series and a tag index to be used.

Leave the first option selected to generate a series across times steps. Choose **Sum** as elaboration. Note that the label text reflect the origin of the series and the elaboration chosen (you are free to change it). Pressing **Ok** the new series will be added to the list of available series. Empty the central list and plotting a graph of this new series; it will be obvious that the sum of the differences is not null across the simulation steps, and therefore we need to modify the model if we want to force the dimension of the market simulated in our model to remain constant.

2.21 Replacing a variable

Given our interpretation of sales improving according to the relative quality in respect of the average, what we need to change 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 for a general consumer, 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 tools to fix it offered by LSD.

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.

The code for **WAverageRW** is:

```
EQUATION("WAverageRW")
 /*
 Weighted average value of all the RandomWalk
 values using Sales as weights
 */

 v[0]=0; v[1]=0;
 CYCLE(cur, "MyObj")
 {
  v[3]=VS(cur,"Sales");
  v[2]=VS(cur,"RandomWalk");
  v[0]=v[0]+v[2]*v[3];
  v[1]=v[1]+v[3];
 }

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

In order to use this new variable, we need to upgrade the equation for ***RelativeQuality*** (see the equation's code in paragraph 2.18 at pag. 39). 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 ***AggregateObj***. Here add the new variable ***WAverageRW***, and set its options so that to save it for post-simulation analysis. We are now ready to run the simulation, but the an error message will soon appear.

2.22   Dead-lock errors - Spotting and fixing temporal inconsistencies

The simulation aborts immediately. 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, since they do not respect the grammar of the language used. Typically, you can mistype a command or forget the semi-colon at the end of a line. These errors are recognized at compile time, and they need to be fixed before the program is created.

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 by brute computational force, and the machine applying its typical mechanical attitude of actually trying.

As an example of a dead-lock error, consider a set of three variables with their equations, each using one of the other variables:

$$X_t = f_X(Y_t)$$

$$Y_t = f_Y(Z_t)$$
$$Z_t = f_Z(X_t)$$

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 interrupts 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 provided by the subroutine.

Let's see how this work in the example above. Suppose to start trying to compute firstly 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 freeze computers because 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. LSD recognizes dead locks before the operative system and interrupts the simulation providing all the information required to fix 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. In mathematical terms those set of equation 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 all satisfied*. In other terms, mathematics interpret a set of equations as conditions, or constraints, to be satisfied, without any indications on how the variables may be computed.

On the contrary, in computational terms a set of equations is interpreted as instructions to be executed, computing the values on the right-hand side of the equation to be stored in the variable indicated on the left-hand side, without any prior indication of the properties of the outcomes of those computations

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 legal 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 LSD Model Manager began the updating of the variables in 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[26]. 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 weighted 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. The obvious one 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 four options:

- **Return to LSD Browser**: return to the browser as if the simulation was terminated normally.

- **Analysis of Results**: move to analyse the results produced so far.

- **Data browse**: show every copy of the objects in the model, and every element within them.

- **Quit LSD model program**: kill the LSD model program.

In our case we don't have data to analyse, because the simulation crashed at the very first time step. Also the **Data browse** 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**.

2.23 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 how to go to fix it.

---

[26]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, re-using the previously computed value.

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 pattern 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 to their joint effect through time. The fact that a dead lock error occurred is a signal not much (and not necessarily) that your individual equations were wrong in the first place, but that the system as a whole was incoherent. And this is a potentially useful result: however negative, acknowledging an error is the first step to solve it.

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 assume different types of behaviour by consumers and producers. A dead-lock error, in this case, 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 consistently a given phenomenon, modelers are forced to devise rigorous algorithms, and therefore to have a precise idea on how the world really functions. And one of the undeniable properties of real-world events is that they take place in real time, and therefore the timing of the simulated events have a relevance, as much as the timing of real-world events is important. In practice, finding a solution to a dead-lock error is not a difficult problem, considering how real-world examples actually function.

In our case we can easily solve our the problem by modifying one of the three equations concerned with the error, **WAverageRW**, **Sales** or **RelativeQuality**. The change should consists in replacing the request for the present-time value of the variables `V(...)` with the lagged value `VL(...,1)`[27].

In such a simple model the different alternatives are likely to have similar effects. To minimise the changes to the present version of the model, we assume that the average quality is computed using *past* sales, instead of present time values.

Note that the change modifies the ordering of execution of the variables within a time step, but we need not (and, actually, can not) express this change explicitly, since the LSD Simulation Manager will take care of modifying the order of execution as necessary.

The only modification we need to do to the model consists in modifying the code expressing equation for **WAverageRW**:

```
EQUATION("WAverageRW")
/*
Weighted average value of all the RandomWalk
values using Sales as weights
*/

v[0]=0; v[1]=0;
CYCLE(cur, "MyObj")
{
 v[3]=VLS(cur,"Sales",1);
 v[2]=VS(cur,"RandomWalk");
 v[0]=v[0]+v[2]*v[3];
 v[1]=v[1]+v[3];
}

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

---

[27]Note that in the equation for **WAverageRW** we are using the values of sales from a specified object, `VS(cur, "Sales")`. Obviously, in this case we should use `VLS(cur, "Sales",1)`, requesting a lagged value from a specific object.

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

2.24 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 distributional element assigning ***Sales***. Let's see how the distributional element work, by transforming the ***RandomWalk*** variables so that they remain constant throughout a simulation run and having a different values for each firm.

To do this we can simply edit the structure of the model editing the elements ***RandomWalk*** and turning them into parameters, assigning to them different values. A second way is to maintain ***RandomWalk*** as variables and squeezing the limits of the random oscillations to zero, that is, assigning 0 to ***minimum*** and ***maximum***. This would produce a constant value of 0 for all the ***X1*** variables at any time step:

$$X1_t = minimum + (maximum - minimum) * X_t = 0 + (0 - 0) * X_t = 0$$

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

$$RandomWalk_t = RandomWalk_{t-1} + 0 = RandomWalk_0$$

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.

Save this configuration with a different name, say **sim2.lsd**, in order to not overwrite the previous one. Running a simulation you will see that an orderly pattern is clearly visible, but the eventual distribution cannot be seen because the simulation terminates too early. Reload the configuration (**Ctrl+w**) and change the simulation settings (menu **Run / Sim.Settings**), inserting 1000 steps as limit instead of 100.
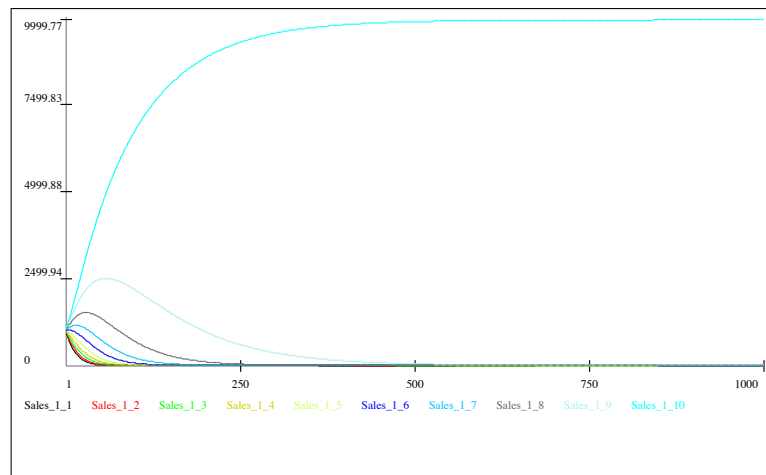
Figure 2.18: Sales time sequence with constant qualities.

Now our model represents a group of firms of different, but constant, 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. 2.18. We can see that the 10th firm gains in the end all the sales, while the others decrease to 0. This is obvious because the 10th firm has the highest quality. The interesting aspect of the simulation consists in the patterns of the different sales' variables.

Suppose you want to interpret the result according to the explanation: the dynamics of sales depend on the quality of the firms. This seems to be confirmed by the pattern of the worst firms (always decreasing) and of the very best one (always increasing). However, the second and third firm, at least, have their sales initially increasing and then decreasing: don't they have the quality constant as the others? Though the answer can easily be found logically, we can use this question as an excuse to explore another tool of LSD: how to find detailed evidence of the micro-events within a simulation run.

2.25   LSD Debugger

In many cases (and we may even claim the most interesting ones) we have a simulation model producing unexpected results, difficult to justify on the basis of the equations. This happen because, however simple may be the equations, the non-linear interactions among many elements through time are very hard to predict, and this is the very reason for performing simulations. Obviously, it may even be the case that our model is simply wrong, containing errors.

In both cases it is very difficult to investigate the behaviour of the model on the basis of the time series of results alone, as they are provided by the Analysis of Results, or by a sophisticated statistical analysis. This is because reconstructing the dynamics producing the results requires a careful analysis of the events within a simulation run. For this reason you do not necessarily need a lot of data, actually too much data may be as confusing as too little. What is necessary is to access data in the appropriate way. We may even need to run a counter-factual experiment modifying what we suspect to be a crucial value. In general, we need to "freeze" the simulated world represented by the running model, and potentially investigate all of its elements, gaining an understanding that is much more detailed than that available looking at the resulting series alone. It is the same difference as having available the census data of a country at aggregate level along, or having the chance to reach each and every person's state: to understand how aggregate patterns emerge you need sometimes to access micro-data.

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[28]. Let's see how to use the LSD debugger. Quit the Analysis of Results (**Exit/Exit**) and re-load a fresh configuration (press the keys **Ctrl+w**).

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. That is, which equation will interrupt the simulation. To ensure that no other variable is already to be debugged, use menu **Run / Remove Debug Flags**.

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 **RelativeQuality** 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 as soon as the first copy of a variable **Sales** complete its equation.

Run the simulation, and the model should interrupt showing the window reported in figure 2.19.

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; see the

---

[28]The name is due to the use of this function for spotting errors, that is bugs, in programs, when they generate unexpected results. In this case it is necessary to run the program step-by-step to reproduce and identify the sources of the error. In a simulation program, a "bug" may be either an error or simply an unexpected result.
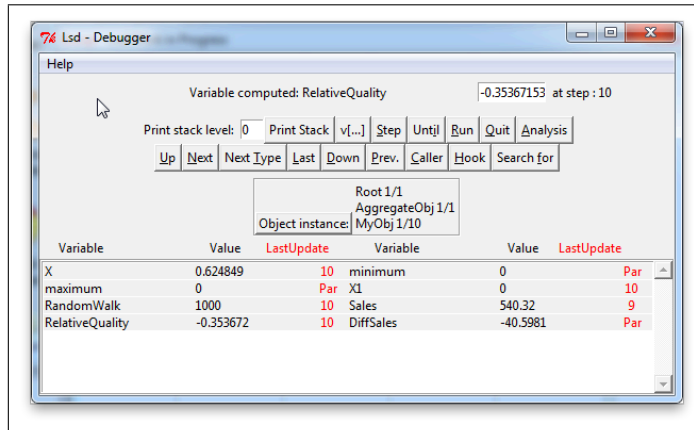
Figure 2.19: Debugger set on the variable **RelativeQuality** at the 10*th* time step.

menu **Help/LSD Debugger Help** for a complete presentation for this interface. Let's see firstly the items contained in the page.

The top part of the figure, the header of the debugger, reports the label of the variable just computed when the simulation was interrupted (**RelativeQuality**), the value resulting from its computation (-0.353671), and the time step of the simulation (10).

Below the header there is a row of buttons controlling the simulation run. These buttons operate and provide information on the dynamics of the simulation. For example, button **Run** would tell the model to continue the simulation until the end, **Quit** aborts the simulation at current time step; **Step** will continue the simulation until the next variable marked to be debugged is updated. 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")[29]; you can also use the arrows to move around the model.

Below the buttons you find the indication on the specific copy of the model you are observing. Contrary to the LSD browser normally used, the debugger browser does not show only the generic content of an object type, but shows the actual content of a specific copy of an object. Hence, you are shown the ordinal number of the specific copy, including the ancestors of the object up to **Root**. Moving "right" across the set of **MyObj** objects you will see that their ordinal number changes.

Below the buttons you observe the how sequence of ancestors of the object currently shown (which is indicated along the label **Object instance:**). Note that each element is indicated as the ordinal number of the object within its group.

The remaining part of the window shows the actual content of the object: the list of 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 lastly updated. 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 did not receive their newly computed value (e.g. variable **Sales** in the figure). Comparing the value **LastUpdate** 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, we interrupted the simulation when the **RelativeQuality** of the first **MyObj** has been completed. Therefore, for example, we will have that copies of **Sales** in the other **MyObj** will not be updated for the current time step, as well as all the **RelativeQuality**'s variables.

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**. We see that all the values for **RelativeQuality** (concerning time step 9) 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

---

[29]The underlined letters in the buttons' text indicate the keys available as alternative to click on the buttons.

WAverageRW. Given our set up (*minimum* and *maximum* set to 0), variables *RandomWalk* never change. Actually, it is *WAverageRW* that does change. At time 10 this variable has the value of 1547.2, as we can observe moving the debugger in *AggregateObj* 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 *WAverageRW*. Therefore, the values of quality for some firms was above average in the beginning, and therefore their sales level increased. But later *WAverageRW* increased so to overcome the levels of quality for each firm but the 10th, so that eventually all these firms have decreasing levels of sales.

Once a property of the model has been identified it is necessary to find the a suitable format to describe it. While the modeller has a deep knowledge of the model (and of the modeling tool), it is necessary to find a synthetic and clear format to communicate the relevant knowledge to an audience that does not have available the same tools and skills. Let's see how to prepare a clear presentation to answer this question.

Complete the simulation run, clicking on the button **Run** and open the Analysis of Results. As we mentioned before we wanted to understand why sales dynamics of some firms are not monotonous, by change sign even though their qualities are constant.

Let's consider the case of the 9th series (the next-to-best) which increases its sales until about time step 60, and then decreases it. We can produce a graph containing its sales and quality, as well as the weighted average quality. This graph shows that the change of direction of sales for this firm takes place exactly when the firms' quality crosses the average quality.
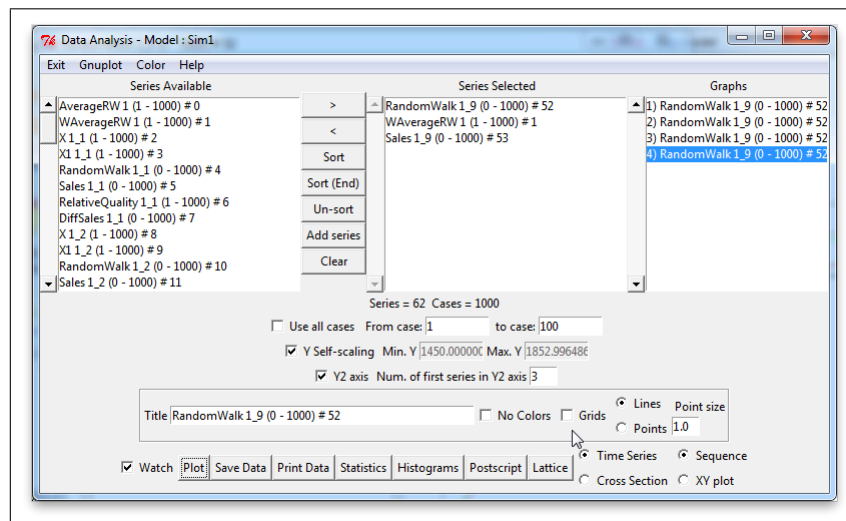
Figure 2.20: Options for the Analysis of Result creating a graph using a restricted period of the simulation and using two different scales, with the second scale being used for the third series.

To generate this graph we need to make use some of the features of the Analysis of Result window. Firstly, moves in the **Series Selected** box the series for the quality of the firm (*RandomWalk* with the tag 1_9) and the average quality (*WAverageRW*). Finally, add also the sales for the same firm (*Sales*, with tag 1_9). We need then to plot a graph restricted to the first 100 time steps, and using two different scales: one for the two qualities and one for the sales. These options can be obtained using the checkboxes and entries in the middle of the Analysis of the Result window, as indicated in figure 2.20

Once the graph window has been produced we can edit it by adding, removing or editing labels. To add a label keep the shift key pressed and click on the graph window. Type into the resulting entry the text desired and confirm. To edit an existing window click on it with the right button of the mouse, and edit the text or format. To remove a label do as for editing it and assign an empty label. See the help menu for further details.

The resulting graph is reported in figure 2.21. Note that the graph windows can be saved as encapsulated postscript file for use in any word-processor (again, see the help on details).

The model we developed so far is driven by the functional form we impose on the variable
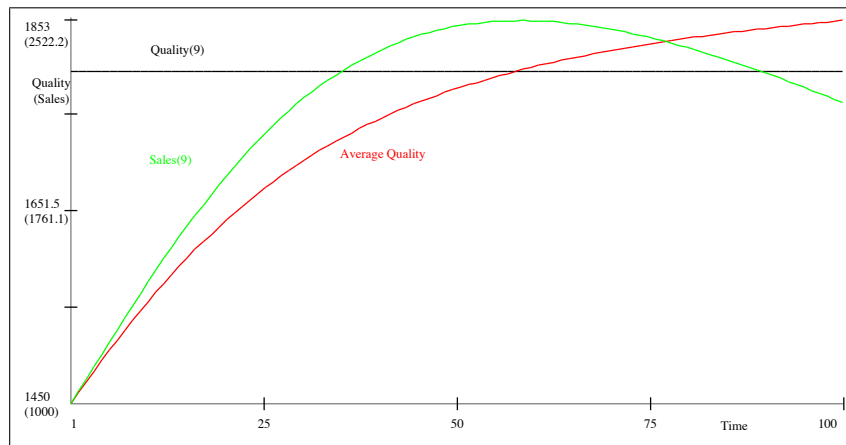
Figure 2.21: Graph produced with the options as in figure 2.20 and edited by changing labels.

Sales. This equation represents implicitly the behaviour of the consumer, who are supposed to act in such a way to generate the prescribed dynamics of sales.

This modeling approach is generally used for mathematical models, since it can exploit convenient mathematical properties of the chosen function. But it also introduces strong rigidities on the type of models that can be generated. In our case, it is impossible, for example, modify the model to obtain a sales level positive for all firms. The model we built is, essentially, a model explaining the pattern to a monopoly, not a model of competition. For such purposes we need to devise a different model, where consumers are explicitly represented and their different preferences used to generate sales levels which are persistently differentiated, but not converging to a monopoly.

We will need to "deepen" the model, replacing an imposed dynamics (our equation for sales dynamics) with a derived dynamics, where sales are the result of consumers' behaviour. The resulting model will be more difficult to deal with mathematically, but will be more flexible in the type of representation. From the programming viewpoint we may continue to update the element of the model we have already generated, but it is more convenient to start a new model from the scratch to avoid confusion caused by mismatched labels or spending time re-naming many elements.

Chapter 3

Implementing LSD Models: Example 2

This tutorial will skip on details of the elementary operations and will focus on relatively advanced features of LSD coding: use of functions; manual control of precedence; optimization of code.

3.1 Model Content

The model implemented in this section is the computable version of a model originally[1]implemented as a standard dynamic model, where a system of differential equations is "solved" providing asymptotic solutions under very strict assumptions. We will reproduce the same results by means of a computational model, which automatically allows also the possibility to observe the path leading to the limit values and provides the basis for infinite extensions.

The model addresses the issue of consumers who are not able to evaluate the quality of products they buy, neither before nor after the purchase, because of undetermined reasons. The only information available to consumer is that their product needs replacement every once in a while because it "breaks down", i.e. for some reason does not provide any longer its purported services. The actual quality of products, unknown to consumers, is the probability that a given product will break down at any unit of time. Consider **ProdUsed** a variable indicating which product a consumer is using. Its dynamic can be represented as follows:

$$ProdUsed_t = \begin{cases} ProdUsed_{t-1} & \text{, if } isBroken(ProdUsed_{t-1}) = 0 \\ Purchase & \text{, if } isBroken(ProdUsed_{t-1}) = 1 \end{cases}$$

where the expression $isBroken(i)$ indicates a stochastic event based on the probability that product i breaks down. This may be expressed as:

$$isBroken(i) = \begin{cases} 1 & \text{, with probability } BD_i \\ 0 & \text{, with probability } 1 - BD_i \end{cases}$$

where **BD_i** indicate the quality of product i, hidden to consumers.

The model is completed by defining the action of consumers activated when they need to replace their broken product, defined above as **Purchase**. The model assumes that consumers rely on the only information available to them: the relative popularity of different products. The model assumes therefore that products are branded and consumers are able to recognized the brand of products, though not able to compare quality levels across different brands. Hence, they tend to choose more popular products with higher probability than those with lower market shares.

Crucially, the model assumes differences in *trust* among consumers in respect of the reliability of the market shares to correlate with products' qualities. Markets where consumers have little faith that market shares are a good indicator of products' quality will tend to choose with more

---

[1]Smallwood and Conlisk, 1979, "Product Quality in Markets where Consumers are Imperfectly Informed", *Quarterly Journal of Economics*, 93-1.

even probability any brand. On the contrary, markets where consumers trust market shares as a reliable indicator will concentrate their purchase only on the most popular.

The model implements this assumption by assuming that the probability of choosing any given product i is the following:

$$\text{Prob}(Purchase = i) = \frac{ms_i^a}{\sum_{j=1}^{N} ms_j^a}$$
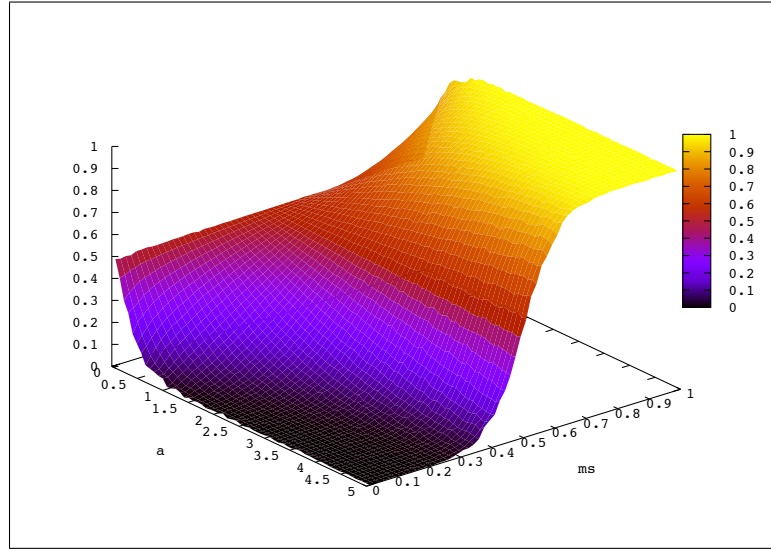
Figure 3.1: Probabilities obtained by biasing the market share of one product in respect of different values for a.

The relation between market shares and probabilities depend on the parameter a. The higher a the more extreme are differences between probabilities, while lower a indicates flatter probabilities. Figure 3.1 shows a visual representation of the effect of a. The graph reports the probability resulting for a hypothetical option among two alternatives for different values of the market share of the option and of a. Increasing values of a clearly show the increasing polarization of probabilities. At the two extreme we have that for $a = 0$ the probability is uniformly distributed irrespective of the market share. On the contrary, for high values of a the product enjoying even minimal advantage obtains a probability of being chosen virtually to 100%.

Given the two rules on the "technology" (when the product breaks down) and on the consumers' behaviour, the model explores the question on whether the market as a whole is able to identify the best product (least probability of breaking down), even though no single participant to the market owns such information.

In the following we are going to implement a LSD implementation of the model able to answer this question, choosing a version able to illustrate a few advanced features of the system.

3.2 Model structure and core equations

The result produced by an equation in LSD can (and frequently does) depend on the copy of the object containing the element. That is, the very same code of an equation associated to an element in, say, **Firm** may produce a different result if moved to **Market**. This is due to the fact that LSD fills automatically many details left un-specified in the code by inducing the intention of the modeler from the position of the variable in the model structure. Relying on automatic rules not only requires far simpler code from modelers, but it is also central in the possibility of modifying existing models and re-using code in different models, since the system automatically re-arrange the code to changing conditions.

Because of the importance of the model object structure, let's start by defining the objects into which we will place our computational content. The structure is pretty straightforward: we define

an object ***Market*** containing a ***Supply*** and ***Demand*** side. In turn, ***Supply*** contains several objects ***Firm*** and ***Demand*** several objects ***Consumer***. The structure of the model is reported in figure 3.2.
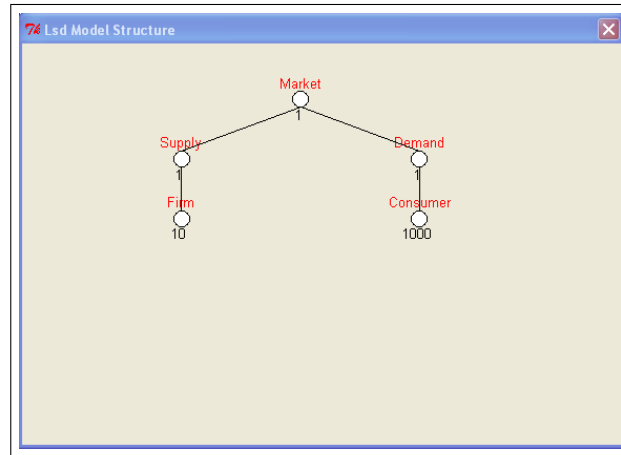
Figure 3.2: Object structure of an model. Object ***Market*** contains ***Supply*** and ***Demand***, which, respectively, contain ***Firm*** and ***Consumer***.

Let's define a "brand" for each firm, that is a way to identify a product from others. To do so we can simply define a parameter in the objects ***Firm*** initialize with a different value (say 1, 2, etc.) for each copy. Insert parameter ***idFirm*** in the object ***Firm*** and assign as initial values increasing integers starting from 1 (use **Set All** and option **Increasing**, using 1 as starting point and 1 as step). Also, assign to the same objects the "quality" of products to each firm, that is the parameter indicating the probability of breaking down for users of each firm's products. Insert parameter ***BD*** and assign 0.1 to the first firm, and then increase the value by 0.01 to each following firms (as beforem **Set All** and **Increasing**, using starting value 0.1 and increasing by step 0.01).

Let's start by writing the code for the equation concerning the product used by consumers. The variable will indicate at time t the id of the product used by the consumer. Its equation can be expressed as follows

```
EQUATION("ProdUsed")
/*
Determine the product used by the consumer
*/
v[0]=V("IsBroken"); //breaks ?
if(v[0]==1)
  v[1]=V("Purchase"); //yes, buy a new product
else
  v[1]=VL("ProdUsed",1); //no, keep on using the previous one
RESULT(v[1])
```

The code is pretty straightforward, but a few comments may be useful. Firstly, we avoided tackling the main problem by delegating element ***IsBroken*** to produce the crucial information on whether the product breaks down. Once this is resolved, the equation is a mere conditional statement. Note that the logical condition "is equal" is expressed with the double equal sign == in C++. If the condition is true the first line is executed and the *else* command is skipped, and the opposite in case the condition is not verified.

Similarly, if a new product must be chosen the equation relies on another element of the model to make the necessary operation, ***Purchase***, which will have to return the ***idFirm*** of a newly purchased product. Alternatively, if the product is not broken, the variable ***ProdUser*** will continue to have the value as at the previous time step.

3.3 Finding model data in LSD equations (equation for *Is-Broken*)

Let's see the code for the variable ***IsBroken*** whose main problem concerns the possibility to access the appropriate data from the model. We start by writing an intuitive, but wrong, version for the equation, and then, fixing the errors, we will eventually reach a correct version. The incremental versions of the model will provide the excuse to discuss the different ways modeler can control the data used by equations for their computation, that is, which copy if a given element, among many, is used within the code for an equation. The system provides, by default, a certain copy depending on the location of the element associated to the equation. However, the user can overrule the default system.

The computational content of the equation for ***IsBroken*** is not very complex. The difficult part concerns the access to the necessary data. Consider the following formulation:

```
EQUATION("IsBroken")
/*
Check if the product is broken. INCREMENTAL VERSION 1.0
*/
v[2]=V("BD"); //should be the b.down probability
if(RND<v[2])
  v[1]=1; //product broken
else
  v[1]=0; //product not broken
}
RESULT(v[1] )
```

Provided that the local variable `v[2]` contains the probability of breaking down, the code is pretty trivial. The conditional statement `if(...)` emulates a random event. The LSD function `RND` produces a random value uniformly distributed in [0,1] every time it is executed. Hence, for example, if `v[2]=0.1`, the condition `RND<v[2]` will be true or false 10% and 90% of times it is executed, respectively. In the first case it returns 1, and in the second 0, providing the required result for the equation of ***ProdUsed***.

3.3.1 Automatic data retrieving

The code for the equation, however simple, cannot produce the result we expect because the system is not able to produce the correct value of ***BD*** in the equation. In fact, without any specifier (i.e. using only `V(...)`), the system decides on its own where to retrieve elements requested during a simulation run. Consider the structure of object we built for our model (figure 3.2); whatever object you decide to place variable ***IsBroken*** in, the result will not be the one desired.

To understand why, we need to consider how LSD retrieves automatically the data within the equations. Consider the equation for a variable, say ***Y***. When the code encounters a command requiring data from the model, e.g. `V("X")`, it starts searching an element of the model with label ***X*** in the same copy object containing the copy of variable ***Y*** under computation. This object is referred to by a pointer, `p`, that, contrary to `cur`, cannot be assigned by modelers but it is set by the system for each equation. If the object contains the element, i.e. if the object containing ***Y*** contains also ***X***, the system returns the value from this copy. Alternatively, if the object containing ***Y*** does not contain element ***X***, the system starts a search scanning potentially all objects in the model, according to a precise order. Below is the strategy adopted to automatically search elements necessary for an equation.

1. Search in the object `p`.

2. Search in all objects descending from `p`.

3. Search in the object containing `p`.

The search routine is recursive, that is, replicates the same strategy in every object visited, with the obvious control that it does never pass twice in the the same object. The first object found during the search containing ***X*** the search stops and the value of the element is returned.

3.3.2   Manual data retrieving

Considering that parameters **BD** are contained in objects **Firm**, and given the search routine, wherever you placed **IsBroken** the result of the equation as written above will be wrong, that is, it will not produce the intended result. The problem is that the copy of parameter **BD** to be used cannot be automatically delivered by the automatic system, but needs to be the one indicated by the content of the past value of **ProdUsed** of the object consumer using this code. In other word, the same code for **IsBroken** used by the same copy of object consumer will need to deliver a different copy of **BD**; this means that the code needs to specify "manually" how to retrieve that copy.

The following equation shows how the modeler can by-pass the LSD automatic retrieval system and find a value for a specified copy of an element, in our case a specific firm.

```
EQUATION("IsBroken")
/*
Check if the product is broken. INCREMENTAL VERSION 2.0
*/
v[0]=VL("ProdUsed",1);
cur=SEARCH_CND("idFirm",v[0]);
v[2]=VS(cur, "BD");
if(RND<v[2])
  v[1]=1; //product broken
else
  v[1]=0; //product not broken
RESULT(v[1] )
```

Let's see firstly the new command introduced in this code. The first line of the equation loads in `v[0]` the unique identifier of the product owned by the consumer in the previous time step, **ProdUsed** with one lag. Notice that **IsBroken** is computed during the computation of **ProdUsed**, thus any line in the equation such as `v[0]=V("ProdUsed");` would cause a dead-lock error; but here we are asking for the *past* value of a variable under computation, and this value can be obtained without triggering the computation of the equation.

The second line contains the new command, `SEARCH_CND("Label",val)`. In general, this command implements a search for the element called **Label** following a search strategy identical to the one discussed before for the automatic retrieval system. However, the search does not stop as soon as on copy of the element is found, but continues until the copy of the element has a value exactly equal to **val**. If the system does not contain any element **Label** with value `val`, then the simulation stops issuing an error. Conversely, if one such an element is found, the command returns the object containing this element.

Besides the new command, the code of the equation is straightforward. The local pointer `cur` will contains the copy of **Firm** whose parameter **idFirm** equals the past value of **ProdUsed** of the consumer. The copy of the **BD** parameter will be the one contained by the selected **Firm**, and hence the random event will be computed as before.

3.3.3   Functions vs. Variables

This implementation 2.0 of the equation works provided a crucial condition is respected. The condition is that the line `v[0]=VL("ProdUsed",1);` is able to return the (past) value from the copy of the variable of the consumer we are computing. To ensure this we need to locate **IsBroken** within the same object as **ProdUser**, i.e. in object **Consumer**. This choice is not, however, particularly elegant. We likely wish to have a large number of consumers, and the least number of elements we place there the smaller will be the memory requirements. Besides, the simulation needs not to store the information produced by **IsBroken**, since we are only interested in which product a consumer is using. Hence, we may consider to place the element **IsBroken** somewhere else, for example in object **Supply**, so that every consumer may exploit its code, but not own an individual copy. This choice, computationally more efficient, generates however two problems.

The first problem derives from the choice of having a single element **IsBroken** and many variables **ProdUsed** asking its value. This implies that the equation for **IsBroken** needs to be re-computed several times at each time step. In fact, suppose that the first consumer uses **IsBroken**, adopting the probability of breaking down of its currently owned product. At the same time step, the second consumer will need to re-compute the same code. But LSD prevents variables to re-execute the code for their equation twice in the same time step. Hence, the result provided by the first use of **IsBroken** will be passed to all consumers, causing the wrong result that all consumers will have their product broken, or not broken, at every time step. To avoid this result we need to define **IsBroken** not as a LSD variable, but as a function. In LSD

functions are not constrained to be computed only once at every time step and to return the same value without recomputing the equation every time they are asked so in the same time step. On the contrary, functions have their equation re-computed again and again an element of the model (that is, an equation) requires their value, which is exactly what we need here. Nota that it is possible also to ask for *lagged* value of a function, such as, e.g., in `v[0]=VL("IsBroken",1);`, though the meaning is different in respect of lagged variables. When asked for lagged values of functions the system does not compute their equation, but returns the value computed at the latest execution of the equations' code, independently from the time step in which it had been computed.

3.3.4 Accessing the calling object

The second problem is that placing the element ***IsBroken*** in an object different from consumers, we cannot ensure that the very first line works properly. In fact, if the function is located in ***Supply***, or anywhere else but consumer, the copy of ***ProdUsed*** in the first line is not determined, and the automatic data retrieving is almost certain to fail to produce the correct value. We need to ensure that the copy of the ***ProdUsed*** used in ***IsBroken*** comes from the same copy of consumer that is forcing the computation. This result is not possible by relying on the automatic data retrieval, because this will always return the first copy found. Nor it is possible to use `SEARCH_CND()` because we do not have an identifier for consumers. So, we need to use a third mechanism available to identify a specific element among many copies.

```
EQUATION("IsBroken")
/*
Check if the product is broken. INCREMENTAL VERSION 3.0
*/
v[0]=VLS(c,"ProdUsed",1);
cur=SEARCH_CND("idFirm",v[0]);
v[2]=VS(cur, "BD");
if(RND<v[2])
  v[1]=1; //product broken
else
  v[1]=0; //product not broken
RESULT(v[1] )
```

The difference with the previous version is that the first line does not ask a generic copy of ***ProdUsed***; rather it specifies where the element must be searched in. Note the grammar of the usual command `V()`, which in this case it becomes `VLS(...)` combining both the modifier for lags as in `VL(...)` and for specific objects `VS(...)`. The implementation uses the pointer `c`, which is a system-set pointer (as for `p`), indicating the object of the element whose equation requested the current equation to be computed (the name derives for the c̲alling object). In our case, the variables ***ProdUsed*** trigger the computation for ***IsBroken***, so any time the equation is computed, the pointer `c` will indicate the consumer containing ***ProdUsed*** whose equation caused the computation of ***IsBroken***. Using the system-set object `c` is almost always restricted to functions, because they are always executed on request by other equations, and therefore there is always an object calling their equations. On the contrary, variables may be computed also because of the mere need to update their value, not because requested by other equations. In this case the pointer `c` is not set, and its use in the equation will cause an error.

To test the effect of using object `c`, use the LSD debugger to observe functions ***IsBroken*** and ***Purchase***. When the browser shows the result of these equations you can use button **Caller** to access the object calling the execution of these functions.

3.3.5 Accessing a randomly chosen object

We know that consumers choose a novel product (when they need to buy one) with probabilities proportional to the normalized value of the formula ms^a. So, we need to produce the equation for this variable, that we may call ***Visibility***.

```
EQUATION("Visibility")
/*
Visibility of the firm used by consumers
when purchasing a product
*/
```

```
v[0]=VL("ms",1);
v[1]=V("a");
v[2]=pow(v[0],v[1]);
RESULT(v[2] )
```

The only remarkable aspect of this equation is that you need to use *past* market shares, because this variable is used by consumers to make decisions that collectively will allow to compute present-time market shares. Using the value of ms_t to compute a variable necessary to compute itself will obviously produce a dead-lock. Also, note the grammar for the power function, which one of the mathematical expressions available in C++ and inherited in LSD.

The equation for **Purchase** uses this value to pick a randomly chosen firm and return its **idFirm**. The logical operations for random choices are rather complex: compute the number of alternatives; assign probabilities; draw a random value; etc. However, it is so frequently used that LSD provides a single line command performing all necessary operations on the basis of the the information provided by the modeler. Thus, the code for **Purchase** is very small.

```
EQUATION("Purchase")
/*
INCREMENTAL VERSION 1.0
Make a purchase for the calling object (supposedly a consumer).
RNDDRAW("Obj", "Label") is a Lsd function choosing randomly an
object called "Obj" probability equal to "Label".
*/
cur=RNDDRAW("Firm","Visibility");
v[0]=VS(cur,"IdFirm");
RESULT(v[0] )
```

The LSD command `RNDDRW("Obj", "Label")` can be used only in the equations for elements contained in objects up in the hierarchy that contain as descendants the group of objects to choose, `Obj`. The element `Label` refers to the elements whose (normalized) values must be used as probabilities. The command returns the pointer to the chosen object which, in our case, provides the id of the firm to be used as result of the equation. Note that the command `RNDDRW()` automatically normalizes the values used as probabilities so that, as in our case, the sum of **Visibility**'s is not, in general, 1.

Purchase needs to be located in an object containing the firms, so will be located in **Supply**. Moreover, the code of purchase serves all consumers, and needs to be re-computed every time a consumer needs to make a purchase. Hence, this must be defined as a function, not a variable, otherwise once chosen a product i for one consumer, all other consumers will choose the same product in that time step.

3.4 Using parameters as "passive" variables

We have seen that the model needs to compute the market shares of different firms. The equation for market share is pretty trivial:

```
EQUATION("ms")
/*
Market shares of a firm, computed as number of consumers
using the product (NumUsers) divided by total number
of consumers (TotUsers)
*/
v[0]=V("NumUsers");
v[1]=V("TotUsers");
RESULT(v[0]/v[1] )
```

Obviously, we expect that the variable **ms** will be located in objects **Firm** together with **NumUsers**, indicating the current customer base of the firm. Variable **TotUsers** will instead be located in an aggregate object, such as **Supply**[2].

The equation for **NumUsers** will be expressed as the net variation of previous period value after summing newly acquired customers and subtracting lost ones, i.e. customers owning the product of the firm but discovered it has broken in the current time step.

---

[2]The code for **TotUsers** is not reported, assuming the reader is able to implement the code for the sum of **NumUsers**.

```
EQUATION("NumUsers")
/*
Number of consumers using this product
Computed as NumUsers[t-1]+Sales[t]-Lost[t]
*/
v[0]=VL("NumUsers",1);
v[1]=V("Sales");
v[2]=V("Lost");
RESULT(v[0]+v[1]-v[2] )
```

While their meaning is trivial variables **Sales** and **Lost** pose a serious problem: how can we compute them?

In LSD variables express the changing values according to a computational algorithm, i.e. they represent an "activity" of the object in which they are located. But in the case of these variables we do not want to express any active computation by firm; on the contrary, they are sort of passive accounting book in which the model actors (the consumers) will write the result of their actions. In LSD it is possible to use variables without a specific equation, that rely on other equations to change their value. Actually, though they are conceptually variables in the model because they change value through time, they are technically implemented as LSD parameters, because they lack an equation. So, add the two parameters **Sales** and **Lost** in object **Firm** and then we need to refine previous equations to ensure that the information required to have **Sales** and **Lost** correctly computed in inserted.

Consider, first, the case for **Lost**. This variable needs to express the number of customers of a given firm whose product broke down. Interpreting the variable as an action by the firm, this would need to write the code to scan each and every consumer controlling whether its product broke down. This is obviously highly inefficient, because we would need to scan all consumers for each firm. Besides, we already know whether a consumer has its current product broken, and therefore we can piggyback on the code for **IsBroken** to play two distinct roles in the model: to consumer provides the information of breaking product, and to the firm provides the information on a consumer lost.

Consider this fourth version for **IsBroken**:

```
EQUATION("IsBroken")
/*
Check if the product is broken. INCREMENTAL VERSION 4.0
*/
v[0]=VLS(c,"ProdUsed",1);
cur=SEARCH_CND("idFirm",v[0]);
v[2]=VS(cur, "BD");
if(RND<v[2])
 {
  v[1]=1; //product broken
  INCRS(cur,"Lost",1);
 }
else
  v[1]=0; //product not broken
RESULT(v[1] )
```

The additional code contains the command `INCR("Label",val)` that modifies the content of an element in the model. As for most of the LSD commands there are several variations of the command grammar; in our case we use the `INCRS(pointer, ...)` version that allows to specify a given object where it should be applied, as in `VS(...)`.

The command is among among a few that changes the value of the element indicated by `Label`. The command increases the value of the element by adding the value `val` to the previous value. In short, the amended code not only provides the consumer with the information on whether its product broke down, but also tells the firm that it had lost a customer, updating the counter of the consumers who had the same experience.

We can use the same system for **Sales**. Since **Purchase** already gets the firm chosen by a consumer, we may use this code to update the (passive) variable.

```
EQUATION("Purchase")
/*
INCREMENTAL VERSION 2.0
```

```
Make a purchase for the calling object (supposedly a consumer).
RNDDRAW("Obj", "Label") is a Lsd function choosing randomly an
object called "Obj" probability equal to "Label".
*/
cur=RNDDRAW("Firm","Visibility");
INCRS(cur,"Sales",1);
v[0]=VS(cur,"idFirm");
RESULT(v[0] )
```

3.5 Manual scheduling: semaphores

Whenever the value of one variable, say X_t, is used in one equation the modeler generally does not need to ensure that the variable is already updated. The system ensures that X have executed its own equation to get updated before using its value in another equation. However, this system cannot work for passive variables, such as **Sales** and **Lost**. The system considers them as parameters, and does not attach any time tag to their values, being constant or changing. It is the modeler then that needs to ensure that the simulation cycle is such as to schedule all the operations involving these elements in the correct order.

If we ran the model as it is coded now the result would be disastrous. Suppose, for example, that you initialize **Sales** to 0. At the first time step all firms get their copy of **Sales** reflecting the number of customers who bought their product. For example, assume one firm gets its **Sales** equal to 100. Consider the content of **Sales** at the end of second time step. These parameters have been increased by the sales in the first step, and again increased by those in the second. Hence, **Sales** and **Lost** represent not the values at each time step, but the *cumulated* values, which is clearly not what we want to implement.

This is due to the passivity of these elements, reflected in their nature as LSD parameters. The system does not have the means to control when the parameter is modified by the equations of the model, hence it is up to the modeler to manually construct a computational schedule ensuring that the desired values are correctly computed.

If we want that **Sales** and **Lost** reflect the behavior of consumers at a single time step only, and not the cumulated values from previous periods, we need to ensure that the very first operation at the beginning of a time step consists in resetting to 0 the values of **Sales** and **Lost**. This is done by the following equation.

```
EQUATION("InitTrade")
/*
Ensures that Sales and Lost
are set to 0
*/
CYCLE(cur, "Firm")
 {
  WRITES(cur,"Sales",0);
  WRITES(cur,"Lost",0);
 }
RESULT(1 )
```

The code contains a member of the command `WRITE("Label",val)` which search in the same object as the computed variable the element `label` and replace its value with `val`. The version used in the code (`WRITES(...)`) searches the element in the object specified by the pointer in the first field.

The element **InitTrade** is of no interest as far as the results of the model are concerned, as reflected by the fact that the result value is a constant (may be any value but it is necessary to specify a number, 1 in the example). The equation is purely a service element that only prepares the stage for the passive variables to get properly computed.

In general LSD modelers do not need to worry on the precedence of computations, that is, the schedule of the equations of the model because the system is able to use the variables time tags to arrange a coherent order of computation, if it exists (or issue a deadlock error message). But this is not the case when we use passive variables. The system may execute the equation for **InitTrade** *after* consumers made their decisions, therefore preventing firms to read the values they need. So, we need to tell the model that **InitTrade** needs to be computed before any other equation. There are two alternative systems to obtain this result.

Firstly, modelers can exploit the way LSD produces its own automatic schedule of operations, i.e. how it design one of the possible order of execution of the model equations. The general rule is that LSD starts trying to update the elements in the top objects of the hierarchy: first **Root**, then its descendants, and so on. Hence, placing **InitTrade** high in the hierarchy we have a high probability that its equation is

computed before anything else. But this solution dangerous. If the very first equation executed at the beginning of a time step requires an updated value from another variable, the system turns to compute the latter before completing the former. Hence, even if we painstakingly control that this does cause disruption in our model, we still leave the possibility that any future change to the code will overlook the need for **InitTrade** to be computed early on in the schedule. So, if you produce an extended version of the model, or one user wants to re-use the code, the priority is no longer ensured. These types of errors are particularly nasty because the system does not issue any error: simply the results are different from what one expects.

The second solution is, contrary to the former, highly reliable and robust to changes of the model. It consists in exploiting the automatic scheduling system by explicitly preventing the wrong precedences to occur. That is, we can tell the system that it can proceed with a certain operation only after another one has been completed. In our case we need to we can obtain this result by editing the equation for consumers as follows:

```
EQUATION("ProdUsed")
/*
Determine the product used by the consumer
*/
V("InitTrade"); //the semaphore
v[0]=V("IsBroken"); //breaks ?
if(v[0]==1)
  v[1]=V("Purchase"); //yes, buy a new product
else
  v[1]=VL("ProdUsed",1); //no, keep on using the previous one
RESULT(v[1])
```

The code is identical to the former version but for the very first line. It asks the (useless) value of the variable **InitTrade**. Obviously, it is not its value we are interested into (which is not even assigned to any local variable). We just want to ensure that the system proceed to the following lines only after **InitTrade** is computed. If the copy of **ProdUsed** is executed when **InitTrade** have not been updated at the present time step, then the necessity to compute its value triggers its equation. Since it is a variable, any subsequent call for its value will not trigger another computation, but the previously computed (useless) value will be returned. In essence, **InitTrade** acts as a *semaphore*, stopping the traffic of computation until a given equation completes its own.

Our model needs a second semaphore, again because of the presence of passive variables in the model. While **InitTrade** ensures that the cumulation of data starts at each time t from 0, without carrying erroneously data from the past time step, we need similarly to ensure that we use the computed values for **Sales** and **Lost** only when their values are correctly computed. In fact, we use **Sales** in the equation for **NumUsers**, and we need to avoid that a firm uses the passive variables before all consumers have made their choices. So, we build a second semaphore, call it **EndTrade**, and place it in the equation for **NumUsers**.

```
EQUATION("EndTrade")
/*
Semaphore signaling the end of
consumers activity
*/
CYCLE(cur, "Consumer")
  VS(cur,"ProdUsed");
RESULT(1 )
```

This variable, located in **Demand**, appears as updated to the system only if all consumers completed their choice. Hence, we can edit the equation for **NumUsers** as follows in order to ensure that the levels of **Sales** and **Lost** for the firm reflect the decisions of all consumers.

```
EQUATION("NumUsers")
/*
Number of consumers using this product
Computed as NumUsers[t-1]+Sales[t]-Lost[t]
*/
V("EndTrade");
v[0]=VL("NumUsers",1);
v[1]=V("Sales");
```

```
v[2]=V("Lost");
RESULT(v[0]+v[1]-v[2] )
```

3.6   Initialization by code

Any LSD model program is automatically endowed with interfaces allowing rather sophisticated initializations, i.e. assigning number of objects and initial values to parameters and lagged variables. However, there may be cases in which standard initialization is time-consuming and error prone. In these cases it may be fare easier to write the code of an initialization routine assigning values to the model element at the very start of a simulation run.

In our model we could run the model as it has been defined so far, but the initialization would be relatively annoying. For example, and primarily, we need to ensure that the value of parameter **TotUsers** is identical to the number of copies of objects **Consumers**, meaning that every time we change one value we also need to change the other.

There are also other constraints we need to respect in the initialization. The model is designed to maintain a rigid coherence between data from different objects, namely consumers' and firms' data. That is, the values in **NumUsers** need to correspond to an equal values of **ProdUsers** showing the identification of the firms. So, for example, if the firm with **idFirm**=3 has the value of **NumUsers**=154, you need to ensure that in your model there are 154 consumers whose variable **ProdUsers**=3. The model ensures that a coherent configuration is maintained through time, but at the very beginning it is the initialization that needs to ensure we have a coherent condition to start from.

The manual initialization using the LSD interfaces is feasible but not practical, unless to force very peculiar conditions (e.g. all consumers start using the same product, which will start with 100% of market shares). Thus, we better write the code for an equation whose variable exist only at the beginning of a simulation run and is then removed. As we will see the time necessary to write the code for a generic equation is a fraction than those required for a single coherent configuration by manual initialization.

The equation implementing the code for initialization is generally placed in **Root**. We already mentioned that it is good practice not to use this object to contain elements because of the specificity of this object. However, for the same reason **Root** is particularly convenient to store an initialization routine. The object **Root** is the only one in a model that cannot be replicated in many copies. Moreover, the automatic update of variables start always from **Root**, hence the first variable in this object will certainly be executed before anything else.

The equation needs also to be valid for any possible configuration of the model. Though we have used so far only configurations made of a single **Market**, in general it is possible to run simulations with many copies of **Market** to be simulated in parallel. Hence, the **Init** function should be designed for the generic cases of multiple descending objects **Market**.

```
EQUATION("Init")
/*
Initialize the model coherently
*/
CYCLE(cur, "Market")
 {//cur is the pointer for markets
  VS(cur,"InitTrade");
  v[0]=VS(cur,"TotUsers");
  cur2=SEARCHS(cur,"Demand");
  ADDNOBJS(cur2,"Consumer",v[0]-1);
  CYCLES(cur, cur1, "Consumer")
  {//cur1 refers consumers in market cur
   v[1]=VS(cur,"Purchase");
   WRITELS(cur1,"ProdUsed",v[1], t);
  }
 }
PARAMETER
RESULT(1 )
```

Let's see the code for the equation. The equation is composed by two *nested* cycles. The external one scans all objects **Market** in the model, repeating for each of them the same set of operations. The first line within this cycle fires the semaphore for the market so that, at the very first time step, it is not re-computed again. At this time the semaphore is useless, since the values of **Sales** and **Lost** will not contain the values from the previous step.

The second line reads the number of consumers in the model as expressed in ***TotUsers***. The following command searches the object ***Demand***. The SEARCH() command returns the object specified among the descendants from the starting object. The version used here start the search from the object contained in `cur`, that is, the market currently initialized. The results of the search is stored in pointer `cur2`. Next the code shows an advanced command; the line generates a number (exactly ***TotUsers***-1) of new copies of objects ***Consumer***, and place them into ***Demand***. This line obviously assumes that the model configuration is defined as containing a single copy of object ***Consumer***, and therefore it has to create the remaining to reach the total of ***TotUsers***.

The subsequent cycle is defined to scan all consumers in the market. Note the modifier `.S` used in the command for the cycle, whose generic format, `CYCLES(objStart, objCycle, "Label")` , has the following meaning: searches among the descendants from object `objStart` the group called `Label`. Then start a cycle using `objCycle` as recurring containers for all the copies of the group.

For each object ***Consumer*** the equation firstly asks for a computation of ***Purchase*** (note that it is requested to `cur`, that is to the market), and then imposes the resulting value to ***ProdUsed***. This command is pretty powerful, and therefore dangerous, because it alters not only the value of the element indicated, but, in case of variables, also the timing system used to decide the updating schedule for variables. Let's see in detail the meaning of the `WRITELS(cur1,"ProdUsed",v[1], t);)` . It needs to specify the object containing the copy of ***ProdUsed*** to be written onto, `cur1`, hence the modifier `.S`. It also uses the modifier `-L` normally used to mean lagged values. In this case, however, it indicates the time at which the variable concerned by the command must appeared as being computed. The last field of this command indicates that all the variables ***ProdUsed*** for consumers will show as time tag the value contained in the system variable `t`, which is a variable controlled by LSD indicating the current time step of the simulation run.

To ensure that the equation is computed only once for the whole simulation run, the last line of the equation contains a command that affects the very element whose equation is computed. The command `PARAMETER` orders to the element using the equation (which, having an equation must necessarily be either a variable or a function) to turn itself into a parameter. By doing so, the model ensures that the same code will never be repeated again in the time step following the first. In the rest of the simulation, the system passing through ***Root*** will find not variable ***Init***, but *parameter* ***Init***, and therefore will not bother to search for its equation. To avoid repeating the execution of the initializing equation is crucial, because otherwise we will have a new bunch of consumers at each time step, which is not what it is supposed to happen.

Now the model is ready to be simulated. Complete the configuration as reported in table 3.1

3.7   Testing models

LSD models are particularly useful because with a simple click it is possible to pass from a prototype model to a full-scale one. For example, you the model will work equally well if you change the number of consumers to 100,000 or millions.

In many case, in order to exploit the model, we need not much to simulate very large models, but to compare the results from a large number of different versions of the model, i.e. models whose configuration changes for only one option. This is similarly simple, albeit it requires a small change. If you use the configuration suggested above and you increase the number of ***Markets*** from 1 to, say, 100, you will have 100 independent simulations running in parallel. Initializing, for example, the value of parameter ***a*** to different values you will be able to appreciate the weird role it plays in the model.

When the configuration becomes large you will probably face time and dimensional constraints, which may be at least partly met by opimizing the model.

3.8   Optimization

The model we have developed can run in a few seconds when the consumers are in the range of the thousands, but can become increasingly expensive for large dimensions.

There are two main areas in which users may wish to optimize their code: reduce the memory requirements of the model in order to accommodate large models to hardware with limited memory, and reduce the simulation time of a given configuration. Both issues can be addressed by LSD exploiting its underlining C++ layer, so that the simulation program produced by LSD can be optimized to the highest levels and for all systems, including super-computing systems.

Besides the generic C++ optimization, there are also LSD-specific ways to improve the performance of simulation runs. We distinguish actions that can be taken while running a given model, from those that required a modification of the code.

Object(parent)	Element(type)	Initial value
Root(none)		1
	Init(0)	
Market (**Root**)		1
	InitTrade(0)	
	EndTrade(0)	
	TotUsers(P)	10,000
Supply(**Market**)		1
	IsBroken(F,0)	
	Purchase(F,0)	
	a(P)	0.2
Firm(**Supply**)		10
	idFirm(P)	1, 2, 3, ..., 10
	NumUsers(1)	0, 0, 0, ..., 0
	ms(1)	0.1, 0.1, 0.1, ..., 0.1
	BD(P)	0.1, 0.11, 0.12, ..., 0.19
	Sales(P)	0, 0, 0, ..., 0
	Lost(P)	0, 0, 0, ..., 0
	Visibility(0)	
Demand (**Market**)		1
Consumer (**Demand**)		1
	ProdUsed(1)	0

Table 3.1: Configuration for the Smallwood and Conlisk model. First column reports the object and their parente. Second column reports elements within the object and their type: (n) for variable with 'n' lags; (P) for parameters; (F,n) for functions with 'n' past values. Last column number of objects or initial values where necessary.

3.8.1 Optimization running options

The first possibility to generate faster code consists in using the LMM option to compile optimize code. See the LMM menu entry for system and model compilation options.

A given LSD model program can be run faster and using less memory by reducing the number of variables saved for post-simulation analysis. Also, having Run Time Plots reduces the speed of execution. During a simulation run, it is possible to click on the button **Fast** in the **Log** window to avoid any refreshing at run time, hence devoting all the computing power to the computation of the model (clicking **Observe** returns showing detailed information). Finally, it is possible to tell the system to ignore some objects as not relevant for the updating of the system. Clicking on the label of an object when shown in the main browser of a LSD model program you are shown the window for option (see paragraph 5.1.3). There you find the option **To Compute** checked on, meaning that the system will take care of updating the variables contained in that object and in its descendants. Checking the option off will instruct the simulation to ignore the variables contained in that object. Note that the variables may still be computed, if triggered by other equations. For example, suppose that you checked off the option **To Compute** for the objects **Consumers** in the example model. The simulation will still run properly because the equation for the variables **ProdUsed** are triggered by the variable **EndTrade**. The only limitation is that elements from objects not marked to be computed cannot be saved for post-simulation analysis.

3.8.2 Optimizing code: `hook`

Besides any generic trick to optimize the execution time of C++ code, which applies also to LSD, the specific nature of LSD models may generate computational inefficiencies. We discuss here how to remove at least some of these.

The primary strategy to reduce the time of execution of a LSD simulation is to reduce the number of steps necessary to fetch the data used in the equations. As a rule of thumb, any time an equation makes use of elements not stored in the same objects, and particularly when the command

SEARCH_CND is used, then it is likely that there is room for optimization.

In the following we will explain how to make use of particular tricks that are generally used only for optimization, albeit they may be useful for particularly complex models.

As a first step, generate a copy of the existing model so that your attempts will not destroy the work done so far. Shut down any LSD model program running and, in LMM, use the **Model Browser** to copy and the existing model and past it as a new one, changing the name of the directory containing the model (see par. 4.4.1).

The target of the optimization in the model is to eliminate the use of SEARCH_CND in the equation for ***IsBroken***. The alternative method we will use is to establish a "hard" connection between a consumer and the object ***Firm*** it is used.

LSD provides three types of hard connections linking objects: parents; siblings; descendants. Thus, any object is able to access to its immediate neighbours in the model structure, but requires computationally expensive search strategies crawling the model to reach objects further away.

Besides the links create by the system to form the structure of the model, LSD offers also a "free" link for each object that modelers can use to establish direct links cutting across the model structure. This link is implemented as a pointer, evocatively called hook. Thus, in the equations we can manipulate the hook of any object in order to establish or exploit links. Notice that objects can be accessed in the equations as pointers. In particular, the following pointers are available to modelers:

- pointer p is the object containing the element of the equation;

- pointer c is the object "calling" the equation, that is, containing the element whose variable requested the computation of the present equation;

- pointer root is the ***Root*** of the model;

- a number of un-assigned pointers, cur, cur1, etc. to be used in association to LSD commands returning pointers, such as SEARCH, SEARCH_CND, etc.

To access the hook of each of these objects it is necessary to use the C++ grammar obj->hook, as, for exampe, in p->hook or cur->hook.

Consider the following revised version for ***IsBroken***. Its goal is to replace the existing method based on SEARCH_CND with the hard link assumed set for objects ***Consumers***. That is, each consumer is associated to the firm producing the product the consumer is using.

```
EQUATION("IsBroken")
/*
Check if the product is broken. HOOK-based version
*/
//v[0]=VLS(c,"ProdUsed",1);
//cur=SEARCH_CND("idFirm",v[0]);
v[2]=VS(c->hook, "BD");
if(RND<v[2])
  {
  v[1]=1; //product broken
  INCRS(c->hook,"Lost",1); //here is the hook, again
  }
else
  v[1]=0; //product not broken
RESULT(v[1] )
```

The code reports the old code, commented out, as a reminder of what it is the functionality we need to replace. Instead of searching the object with a specific ***idFirm*** at every execution, this version relies on the existence and appropriate setting of the hook for the consumer asking the computation, which is indicated by c because the function ***IsBroken*** is always and only requested by a consumer. So, the whole search process is removed, relying on the fact that c->hook has the same content as the cur in the previous version. The modeler must pay extreme attention to ensure that the implementation is correct, because the errors in case of "broken" hooks are particularly devastating. In the best cases the simulation run

is interrupted signaling the problem. But it may also continue using the wrong firm, so that the results produced are technically valid, hence issuing no warning, but wrong.

Note that also the updating of the passive variable (i.e. parameter) **Lost** is done using the hook of the consumer to its product.

To ensure that the consumers are linked to the product they use we need to modify the equation for **Purchase**, where the consumer must be associated to the product they choose.

```
EQUATION("Purchase")
/*
Make a purchase for the calling object (supposedly a consumer).
RNDDRAW("Obj", "Label") is a Lsd function choosing randomly an
object called "Obj" probability equal to "Label".
*/
cur=RNDDRAW("Firm","Visibility");
INCRS(cur,"Sales",1);
v[0]=VS(cur,"idFirm");
c->hook=cur; //here is the assignment of the hook
RESULT(v[0] )
```

The code is actually not changed. The only difference is an additional line in which the object `c` gets its hook assigned to the chosen firm, an operation of negligible computational cost.

Now the model is ready to execute a generic time step of the simulation run: each consumer is `hooked` to its product allowing quick access to the data of the firm, and the procedure for purchasing updates the `hook` when consumers change product used. However, the simulation still has an initialization problem: at the very start of a simulation run the `hook` are not set, hence we need to update the equation for **Init** to ensure that the `hook` are set at the very first time step.

3.8.3 Optimizing code: `V_CHEAT`

At an early consideration it may appear that the problem of initialization is already solved. One may reason that, since **Purchase** sets the `hook` for the consumer making a purchase, and the initialization uses the **Purchase**, then the problem is already solved. In fact, this is not the case. To have a direct experience, try to run the model with the two equations we made so far. It will create an error because the `hook` of consumers are not set. Why is this so?

You can use the **gdb** debugger (see the manual for information) placing a break on the line `c->hook=cur;` in the equation for **Purchase**. Then run the simulation; **gdb** will interrupt the simulation at the first occasion in which the line indicated is executed. In the **gdb** console type the command: "`print *c`". After pressing enter the debugger will show you the content of the object pointed to by `c`. The information from the screen should be sufficient to clarify the origin of the error.

The problem is that when **Purchase** is executed because **Init** requires its value, in the equation for **Purchase** the object referred to by `c` is not a consumer, but the object containing **Init**, that is **Market**. This is the appropriate way LSD works: pass to an equation as `c` the object containing the element asking for its computation. Since, in this case, the element requesting the computation is **Init**, which is contained in **Market**, this is the object appearing as `c` in **Purchase**. However generally correct, in this specific case we don't need to use the generic system, but a peculiar one, therefore we need a workaround.

When using the standard command `V()` the system reads the object containing the element under computation (appearing as `p` in the equation) and assigns it to `c` to the called element. There is a similar command that allows the modeler to define arbitrarily which object will appear as calling object `c` in the equation for the required element, in effect cheating the system by letting it believe that the calling element is not the actual one. The command is called `V_CHEAT("Label",fake_c)`. The command searches for the element `Label` like a normal `V()`, but the equation for `Label` will find as calling object `fake_c`, instead of the real one (which cannot be accessed).

In conclusion, to fix the problem you need to cheat the equation for **Purchase** to believe that, when it is called by **Init**, it is not **Market** the calling object, but one of the **Consumer**. This is expressed as follows, revising the initialization equation we described above.

```
EQUATION("Init")
```

```
/*
Initialize the model coherently
*/
CYCLE(cur, "Market")
 {//cur is the pointer for markets
  VS(cur,"InitTrade");
  v[0]=VS(cur,"TotUsers");
  cur2=SEARCHS(cur,"Demand");
  ADDNOBJS(cur2,"Consumer",v[0]-1);
  CYCLES(cur, cur1, "Consumer")
  {//cur1 refers consumers in market cur
   v[1]=VS_CHEAT(cur,"Purchase",cur1);//here is the cheating!
   WRITELS(cur1,"ProdUsed",v[1], t);
  }
 }
PARAMETER
RESULT(1 )
```

Note that the command requires the modified version `VS_CHEAT(obj, "Label", fake_c` because we need to use the ***Purchase*** of the specific ***Market***.

Now the model will run correctly, and much faster, saving about 25% of the computational time in respect of the original implementation.

3.8.4   Optimizing large models: `turbosearch`

There are classes of models in which the search for specific objects needs to be taken frequently, and the number of copies among which to search is very large. Typically, this is the case for models on dynamic networks in which a large number of nodes are constantly explored to generate new links replacing old ones.

In these cases the use of `hook` may speed up the use of existing links, e.g. to exchange information, but does not affect the costs of accessing new nodes, e.g. to evaluate the creation of a new link. For these cases LSD offers a command that decreases dramatically the time required to access a generic copy among N, in effect transforming a $O(N)$ problem in a $O(\log_{10}(N))$ (i.e., a simulation run costing 1 hour to access elements would be reduced to less than 4 seconds).

The grammar of the command, still experimental, is reported in the manual pages.

Chapter 4

LMM interfaces

LMM is an auxiliary program supporting the management and constructions of L^{S}D models. The most frequent activities done using LMM are: identify the model to work with (or create a new one), write the code for the equations of the model, set the options to be passed to the compiler, and run the L^{S}D model program. LMM is not essential to generate a L^{S}D model program; any editor can be used to write the equations' file, and the L^{S}D model program may be generated using a standard makefile. However, LMM simplifies sensibly the usual activities, besides providing other functionalities similar to those provided by an IDE.
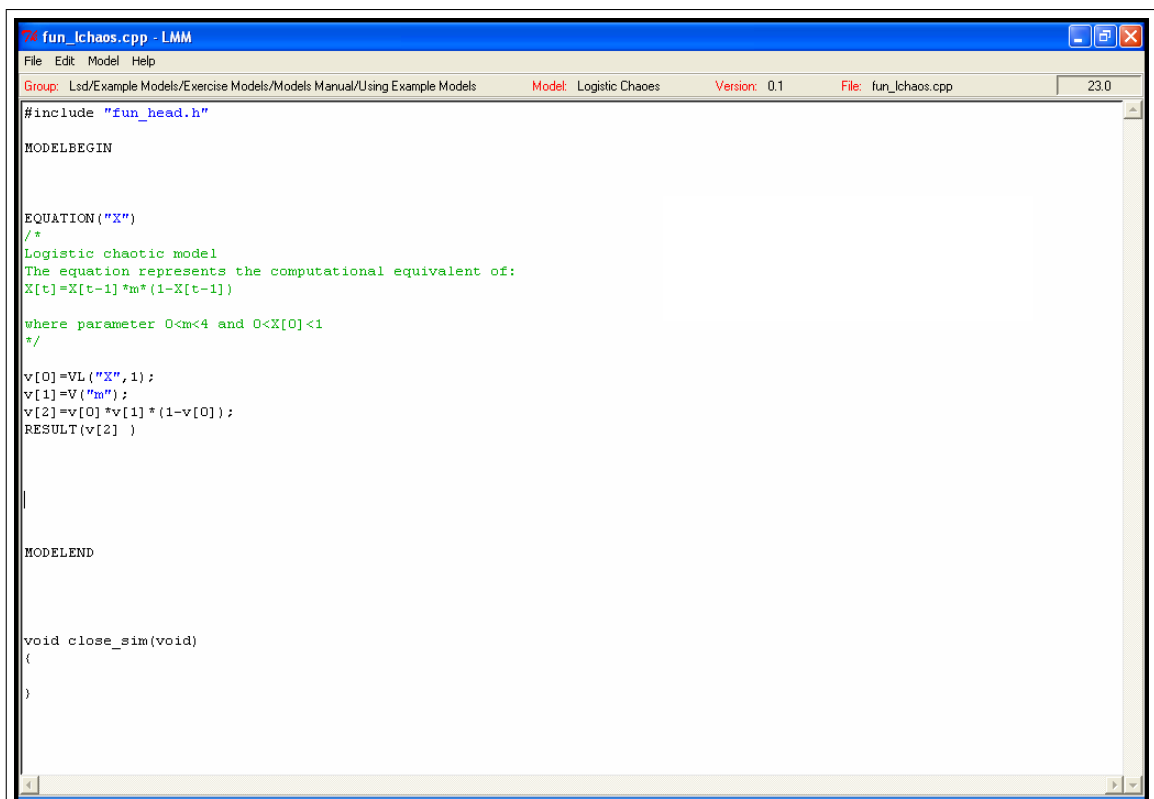
Figure 4.1: L^{S}D Model Manager - LMM

The section documents in detail all the functionalities provided by LMM. The first paragraph below describes the features for the editing environment, while the following describe the menu entries as they appear in the LMM window.

4.1 Editor features

The LMM window contains e menu row, a header, and a main editing window. The header indicates the group and the model currently used by LMM, and the name of the file loaded into the editor.

The LMM contains a standard editor for text files. The editor admits all the standard features concerning selection and movements across the text[1], like **Ctrl+arrow** to move the cursor across whole words or paragraphs.

The text for C++ files is colored in blue for strings of characters (recognized by the double-quotes) and green for comments. Moreover, the editor is endowed of with some event-driven functionalities and shortcuts to frequently used commands.

4.1.1 Click with the right button of the mouse

Right-clicking on a text makes a small menu appear, referring to the position of the text cursor where the mouse cursor is located.
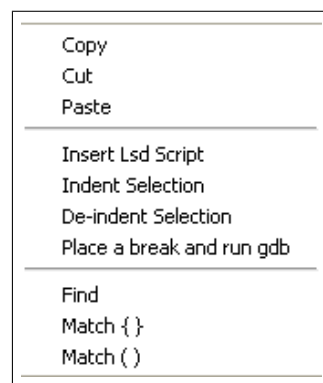
Figure 4.2: Right-clicking on the editor generates a short menu.

The commands offered by the menu depend on the position of the mouse pointer when the right button have been used. Most of these functions are identical to the equivalent commands accessible through the standard LMM menus. The **Place a break and run gdb** has an additional features. Besides running the gdb debugger on the model, it also places a break in the line indicated by the mouse when right-clicking the text. This will cause the model to be executed and the simulation will be interrupted at the break line.

4.1.2 Insert LSD Script

Pressing the key combination **Ctrl+i** the system will insert in the text at the position of the cursor one of the most frequently used commands for the equations. The resulting window can be operated with the arrows and offers users context-dependent help on the different commands, as well as suggesting default values.

4.2 Menu File

The standard menu **File** allows to open, create and save the text files loaded into the LMM editor.

Unix and MacOS users have also a menu entry allowing to set options for the external programs to be used as terminals (required for the gdb program, see menu entry **Model/gdb Debug**) and the web browser, used for the manual pages.

4.3 Menu Edit

The menu **Edit** offers the usual editing functions, plus special functions dedicated to the generation of C++ code.

---

[1]Some of these features depends on the operative system used, and may vary slightly.
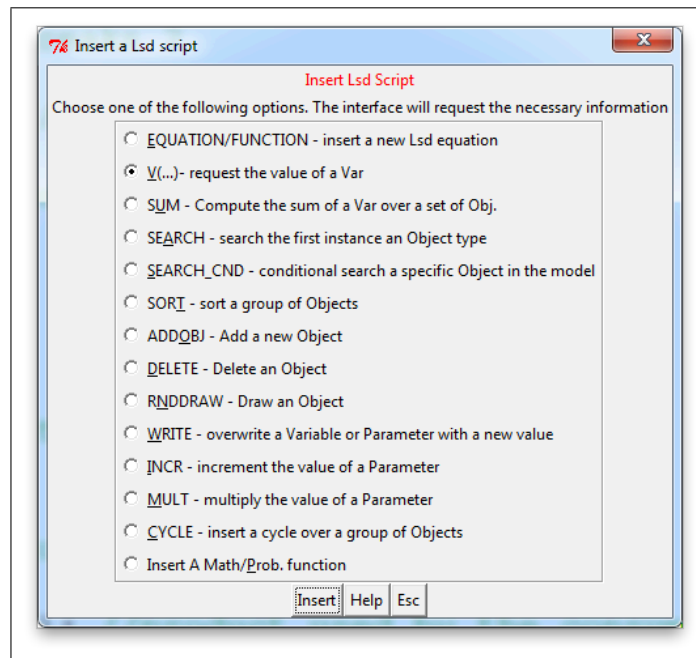
Figure 4.3: Pressing **Ctrl+i** the system automatically inserts one of the most frequently used L^{S}D commands at the cursor position.

4.3.1 Go to line (`Ctrl+l`)

Moves the cursor to the text line indicated by the users.

4.3.2 Match {} (`Ctrl+m`)

When the cursor is position just before a curly bracket { or }, this command scans the matching bracket, discounting the nested brackets.

4.3.3 Match () (`Ctrl+p`)

As the command **Match** {}, but concerning parentheses.

4.3.4 Insert { and Insert } (`Ctrl+(` and `Ctrl+)`)

Insert curly brackets in the text, useful for keyboard layouts missing these symbols.

4.3.5 Wrap/Unwrap (Ctrl+w)

This command alternates the text between being entirely contained in the horizontal width of the screen, or extending beyond the borders for long line. The wrapping concern only the visibility, and does not affect the actual content of the text file.

4.3.6 TkDiff

This entry run a program comparing different text files (supposedly different versions of the same file) highlighting differences between them.

4.3.7 Compare models

This command allows to select two files from different models, and launches the **TkDiff** with the two equation files.

4.4 Menu Model

This menu contains the commands concerning the model currently loaded in the LMM, if any (otherwise, most of the commands are not active).

4.4.1 Browse Models

Open the interface to select the model to work on, or to create a new model.
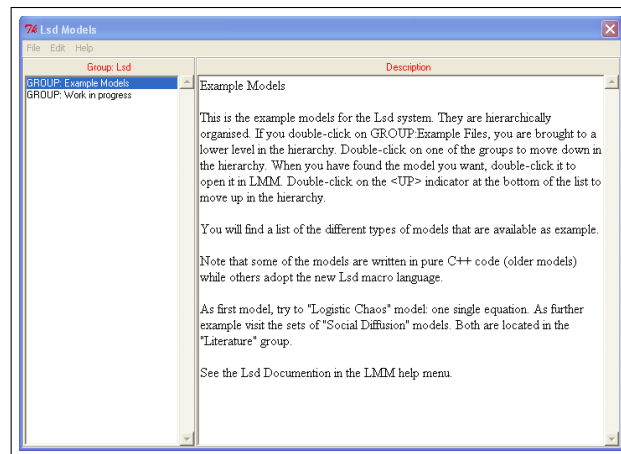
Figure 4.4: Menu **Model/Browse Models** activates this interface to select the model to work on, or to create new models.

The model browser allows to move across the models available in the disk where LSD is installed. Models are defined within **Group**'s, that is, directories, which can be explored with the arrow keys and pressing `Enter`, besides clicking with the mouse. Entry <**UP**> moves the browser to show the models and groups in the upper directory.

Besides selection, menu **Edit** allows to create a new group or model. From this menu it is also possible to **Copy** a model, which can then be **Paste**d in another or the same group. When creating a model the user is requested to provide a name for the model (a few strings), and the directory name associated to the model. Obviously, the name for the directory must be non-existent and must contain only non special characters (i.e. no spaces, quotes, etc.)

4.4.2 Compile and Run model

This entry actually executes three different operations in sequence. Firstly, it controls whether the text currently shown in LMM has been saved on the disk, comparing the text in the editor with the content of the file. If the text and the file differ, LMM offers the option to save, since the compiler reads the file, not the content of the LMM editor. Secondly, after this control LMM generates the LSD model program concerning the model. This process entails the constructions of the compilation instructions concerning the system code and the model-specific code. The compilation may fail because the equation file contains illegal code; in this case LMM shows the list of errors as recognized by the compiler.

Each line from the list of errors refers to a line in the equation file (that can be reached with **Ctrl+l**). In figure 4.5 the compiler found two errors. At line 13 there is a call to command `VL(...)` containing only one field and not two, as expected. Moreover, at line 17 the compiler gave up waiting for the semicolon ; ending a command, which actually should have been placed at line 15. This second error is a typical case in which the error occurred not in the line indicated but the preceding one. The window with the list of compilation error can be closed and re-opened using the LMM menu entry **Menu / Show Compilation Results**.

If the compilation succeeds LMM returns available and the newly compiled LSD model program is automatically executed.
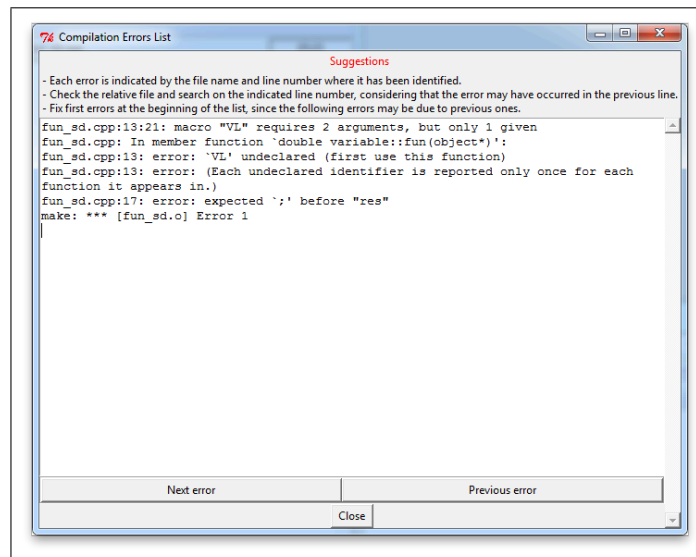
Figure 4.5: Window showing compilation errors as passed by the compiler. Each line refers to a line of the equation file, indicated by file name (`fun_sd.cpp` in the example) and the line number where the error was recognised.

4.4.3   GDB Debug

The gdb debugger is a program allowing users to execute a C++ compiled program line-by-line, normally used to investigate unexpected behaviours of a program. The gdb debugger is a command line program, whose main commands are described in the on-line help of the relative section in the LMM help.

This command runs the command-line GDB debugger loaded with the existing LSD model program. Notice that this command does not compile the LSD model program. Therefore, users must be sure to compile and run the model, then close the automatically run instance of the LSD model program, and only then run gdb.

Notice that for gdb to work properly it is necessary to compile the model with specific options. In LMM you can use the menu entry **Model / Model Compilation Options** and then assigns to the line `SWITCH_CC=-g`.

4.4.4   Show Equation File

This command opens in the LMM editor window the equation file used for the model. Users working on a model should always use this command to ensure that the file they are editing is the actual file used to generate the LSD model program. Obviously, renaming the file opened by the **Show equation** command should never be done, since the saved version is not used for the generation of the LSD model program.

4.4.5   Show Makefile

This command shows the `makefile` concerning the loaded model. This command is used only for reading the content of this file, which contains the complete set of instructions required to compile and generate a LSD model program. This file should not be edited, since the `makefile` for any compilation (**Model/Run model**) is re-created any time it is requested. Therefore, changes to one copy of the `makefile` have no effect.

To change the compilation options, see the compilation options, in this same menu **Model**.

4.4.6 Show Compilation results

This commands shows the error messages generated by the latest attempted compilation, in case it failed. The errors are listed by line numbers on which the compilation identify an error. Normally, the actual error is located just before the line numbers indicated in these error messages. See paragraph 4.4.2 for details.

4.4.7 Show Description

This command opens the text file containing the description of the model. Its name must not be modified. The model description is a user-generated text presented at various occasions when the model is observed by users, such as while browsing models and as header of the automatically generated model documentation. Normally, it should contain a summary of the aims and content of the model.

4.4.8 Model Info

This command shows the name of the model and its version number (which is associated to the name), allowing the users to modify it, permanently. Also, the user can observe, in read-only, the date of creation and last modification of the model's equations, along the full path of the directory containing it.

4.4.9 System compilation options

This command shows the section of the `makefile` concerning the options used to compile the LSD model program. Changes to these options concern the way all the LSD system code is treated, leaving untouched the way to compile the equations' of the model.

The options are expressed in terms of variables used in the `makefile`. These variables concern the location of libraries and files necessary for the compilation, as well as options on the type of executable to generate.

Users may have three reasons to edit these options. In case of installation problems, (i.e. failures to compile because errors not located in the model equations' file) they need to change one or more of the variables indicating the location of the necessary libraries and other files.

The second reason is to generate different types of optimized code. This choice is operated setting to different values the variable `SSWITCH_CC`.

The third reason is to link the executable to an external library. This option can be operated extending the `EXTRA_PAR` content.

4.4.10 Model compilation option

This command changes the compilation options specific to the LSD model program currently loaded in LMM. Three options are permitted: the name of the executable; the name of the equation file to be used; and the optimization options. For the latter use: `SWITCH_CC=-O3` for fastest code and `SWITCH_CC=-g` to use the gdb debugger.

4.4.11 Generate a 'NO WINDOW' makefile

This command generates within the directory of the model a reduced distribution of the current installation (i.e. the files in directory `src`) and of the model file that can be compiled using exclusively pure C++, omitting the graphical libraries used for the windows of LSD model programs. Versions compiled with these options can be compiled on any super-computing machine.

Compiling this reduced version generates a version of the model that is not able to create configurations, but only to execute simulations, even in batch mode. In this latter case at the end of each run the model saves all results in a result `.res` file before starting a new one, so that users may leave the program to run and collect later the results (including the `.tot` result files).

The NO WINDOW version needs to be compiled at command line, using the associated makefile named `makefileNW`. The resulting executable can perform two different types of batch runs. Firstly,

it is possible to run many repetitions with the same identical configuration and changing the pseudo random numbers. Secondly, it is possible to use a sequence different configurations.

In both cases it is necessary to prepare the configuration files for the batch runs with a normal version of the LSD model program. For multiple repetitions of the same configuration just prepare the configuration and specify the number of simulation runs and the initial seed. Save the configuration and run the NO WINDOW LSD model program from command line as follows:

```
$ ./lsd_gnu -f sim.lsd
```

This will execute the configuration in file `sim.lsd` as many times as indicated in the configurations. The results will be stored in files `simX.res` where `X` refers to the seed value used for the configuration. Moreover, the final values from each run will be aggregate in a special result file, `sim.tot`, where the "step" refers to different simulation runs.

The second version of the command has the format:

```
$ ./lsd_gnu -f sim -s 1
```

(where 1 can be replaced by any positive integer) the program expects to find many configuration files starting from `sim1.lsd`, `sim2.lsd`, etc. The program will execute all configurations as indicated by the number of repetitions in the very first configuration file, using the increasing indicators. Also in this case the results are saved independently from each tun and collectively in the `.tot` file.

4.5   Menu Help

This menu provides help pages for LMM and for the general use of LSD. The three entries in the menu concern help pages on LMM, the list of all the functions available to write LSD equations, and documentation on LSD. In particular, a set of slides form a course on LSD use starting from lessons for beginners in simulation and providing several examples for most LSD function.

Chapter 5

L**S**D model program interfaces

The L**S**D model program is a stand-alone executable containing the compiled version of the equation of the model. The L**S**D model program allows users to perform every function concerning the model besides changing the equations: create the structure of a model, insert numerical initializations and simulation options, run simulations, analyze the results, etc.

This section lists all commands provided by the interfaces, which change in accord to the content of models and to the operation currently performed by the user.

5.1 Browser window features

The browser window shows the content of one single type of object in the model and includes all commands necessary to define, modify and exploit a L**S**D model program.
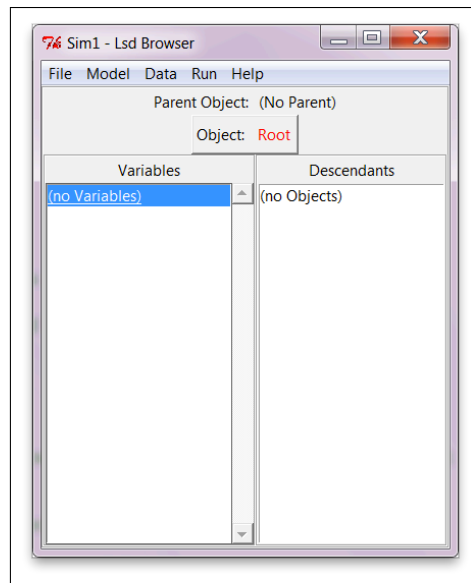
Figure 5.1: The browser window is the main interface of a L**S**D model program. It shows the content of one object and gives access to all commands to the simulation program, like defining a model, initializing its elements, or run a simulation.

The left-hand list shows the variables, parameters and functions contained in the object. The right-hand window shows the *descendants*, that is, the objects contained in the object. On the top of these two lists, the browser shows the name of the object shown, and the parent object of this object. Clicking on the descendants the browser moves to show the selected object, while clicking on the parent object the browser moves to show the parent object. It is possible also to use the arrow keys to move across the labels of the objects. The key **u** shows the parent object.

The browser window, besides its menus, allows observe and edit the elements contained in the object, by double-clicking, respectively, on the label of an element in the left-hand list, and on the label of the shown object.

5.1.1 Moving elements

Within both the list boxes for variables and descending objects it is possible to change the order of elements of the elements using the arrow keys up and down while pressing the control key. The order of the elements normally does not modify the computational content of the model.

5.1.2 Options for an element

Clicking on an element in the **Variables** list (variables, parameters or functions) the window changes to show the possible options concerning that element, as shown in figure 5.2.

The top part of the window is a a header indicating the nature (i.e. variable, function or parameter) and label of the element, besides the object containing it. Double-clicking on the label of the element it is possible to edit its label, or change its nature, for example, turning it from a variable into a parameter. It is also possible to change the position of the element, choosing a different object. Finally, assigning a new empty label to the object the element will be removed from the model.

Below the header there are up to four check boxes. The option **Debug** mark the element to be debugged, meaning that when the simulation is run in debug mode (see below), the simulation will be interrupted showing the state of the model at the very end of the computation of the equation for the element marked as debugged (if any). This option appears only for *Variables* and *Functions*.
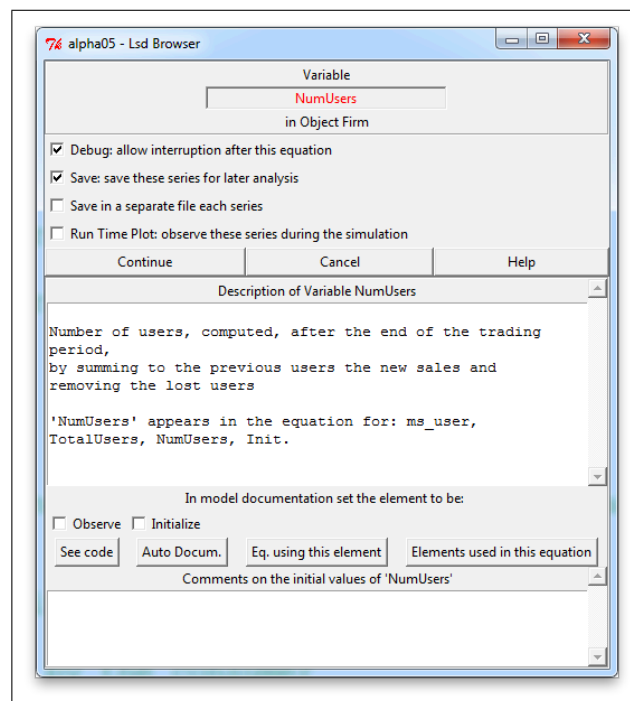
Figure 5.2: Options for an element, activated by double-clicking or pressing enter on the element's label.

There are several modes to activate the debug mode. Users can set the time step at which the debug mode must be activated (in either the **Simulation Settings** before running a simulation, or the button **Until** in the debug module). Otherwise, users can press button **Debug** in the log window at run time. In any case, when in debug mode, the simulation will be interrupted when the first element marked as being debugged is computed.

Option **Save** impose the system to store every value at each time step to be saved for analysis. Elements with this option unchecked will loose their values as soon as they are no more needed (normally, the subsequent time step of their computation).

Option **Save in a separate files each series** will store the values for this element in a separate file labeled after its name and a code referring to its position in the model structure.

Option **Run Time Plot** set the element's values to appear in the dynamic Run Time plot graphical representation while running the simulation.

Button **Continue** confirm any change made to the options and return to the browser.

Button **Cancel** reject any change made to the options and return to the browser.

Button **Help** opens the help page for this window.

The text window entitled **Description of the ...** can be filled with any text documenting the element, supposedly explaining its meaning to readers of the model documentation. See the button **Auto Docum.** below.

The option **Observe** allows to include the elements whose results should be relevant to observe my the model users.

The option **Initialize** is present only for the elements admitting initial values, that is parameters or other elements defined as being used with a lag. This option allows to include the element within the set of elements whose initialization is relevant for the model.

The button **See code** is present only for variables and functions. Press this button generates a new window containing the code of the equation for the element.

Button **Auto Docum.** fills the text for the description of the element. The text inserted depends on the nature of the element. If it is a ***Variable*** or a ***Function***, it is the text contained in the first lines of comment present in the equations' code immediately after the beginning of the equation's code. This comment is supposed to contain the relevant description for the element. For any kind of element, including parameter, the text is completed with the list of elements (if any) whose equations contains the element shown. This information is assumed to be relevant as documentation to understand which part of the model is affected by this element, but does not affect the model results

Button **List of vars. using '...'** generates a list of the elements whose equations make use of this element. This list can be clicked to move the browser to the elements selected.

Button **List of vars./pars. used in '...'** generates a list of the elements appear in the equation's code code for the element. This button appear only for variables and functions.

The final text window appears only for elements that may be initialized, like parameters and other elements declared with a lag. The text contained here describes how the element has been initialized. By default, the initialization functions fill automatically this window.

5.1.3 Objects' options

Clicking on the label of the object in the main LSD browser (figure 5.3) it is possible to change the name of the object or, assigning an empty label, to delete the object from the model. Removing the object removes also all the descending objects.
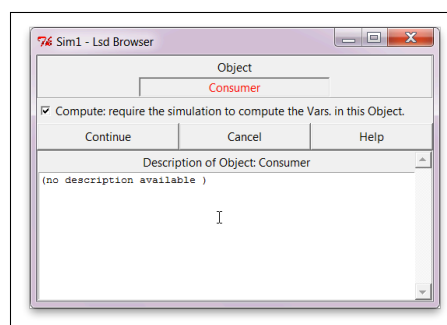

Figure 5.3: Options for an object, activated by pressing the label of the object shown by the LSD browser.

The check box **Compute** (set on by default) allows the L^SD simulation manager to visit the object at every time step and compute the variables contained in the object. Checking off this option prevents the L^SD simulation manager to update automatically the variables contained in the objects, including also the variables in the objects contained in the object. As a consequence, the simulation will be faster, but the modeller must ensure that, if any variable is contained in the object or its descendants, they are updated because explicitly triggered by the code for other equations. Moreover, any element contained in the objects not visited by the L^SD simulation manager will not appear in the elements saved for analysis of results, even though they have been marked to do so.

The text section in the option window defined as **Description of object ...** can contain any text describing the object, which will be used for the documentation of the model.

5.2 Menu File

The model contained in a L^SD model program can be stored in a file to be later reloaded. These files, called configuration files, store every relevant information containing the model. Loading a configuration file restore the L^SD model program in the same conditions as when the file had been saved (aside possible simulation results, if any).

The menu **File** concerns the opening and saving of configuration files. Note the **Re-load**, with its shortcut **Ctrl+w**. This command loads the configuration with the same name as that currently loaded into the L^SD browser. This is mostly used at the end of a simulation run to reload the configuration and run another simulation, possibly after editing the initial values or other settings.

5.3 Menu Model

This menu allows to modify the structure of the model, for example, adding elements to the object shown in the browser. This menu should be used only when building a model; data and simulation options are controlled by the other menus.

5.3.1 Add a variable (Ctrl+v)

Add a variable to the object shown in the browser (figure 5.4). Variables are elements associated to an equation, which is used at each time step to compute a value for the variable.
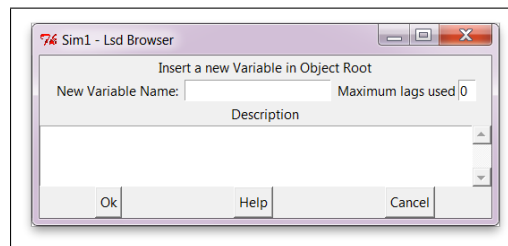
Figure 5.4: Adding a new variable to an object requires the label of the variable and a number of lags.

New variables are inserted by assigning a label and a number of lags. The label must be a text string without special characters, as spaces, quotes, etc., and must not be used elsewhere in the model. The label must match the variable's equation headers, and the matching is case-sensitive (i.e. X is different from x). When the variable is created it is not necessary that the corresponding equation already exist, but the model can be successfully executed only if the L^SD model program is compiled with the equations for all the variables and functions declared in the configuration.

The number of lags must be the highest lag by which the new variable is used in some equation of the model. If, for example, the variable value is never requested by other equations with its past values, then the number of lags can be zero. Conversely, if the model includes an equation where the new variable must provide its past value (at time $t-1$), then, the number of lags must be 1. If it is requested with its values at time $t-k$, then the number of lags must be k.

Notice that variables declared with lags must be provided with initial data, that is, the values to be used in the earliest time steps as past values, which would refer to a negative time.

If a variable is declared with a number of lags smaller than those used in the model's equations, the simulation run will be interrupted and generate an error. In these cases, it is possible to edit the variable's definition accessing the variable's options, and change its nature by clicking on the variable labels.

When inserting a new variable it is possible to provide a textual description of the variable, although this information is generally provided automatically using the text inserted in the equation's code (see the section on elements' options 5.1.2). The elements' descriptions are used for the various forms of documentation of a model.

5.3.2 Add a parameter (Ctrl+p)

Add a parameter to the object shown in the browser. Parameters are elements that do not change value of their own accord during a simulation run, although they can be written over by the code of some equation. Parameters can be saved as sequences of values, to be used in the analysis of results, if this makes sense.

The label of a parameter must be a string without special characters. As for variables, a textual description may be provided.

5.3.3 Add a function (Ctrl+n)

Add a function to the object shown in the browser. Functions are like variables, but for a crucial difference. Both functions and variables are associated to equations that compute their values. The difference is that variables are computed always once, and only once, at each time step, and the system ensures that this happens under every possible circumstance. Instead, functions are computed (i.e. their equation is executed) only, and every time, the function's value is requested. Therefore, a function may have its equation computed several times or never within a time step, depending on the calls by other equations, while a variable has its equation executed once and only once at each time step. In a sense, variables are used to time-driven dynamics, taking new values in synch at each time step, while functions are used for event-driven dynamics, when a computation is triggered by other events. In LSD models the two approaches are normally used in the same model, using the most appropriate for each part of the model.

Functions can be declared, like variables, with lags. However, the "past" values of functions do not refer to the number of previous time steps, as variables, but to the number of previous activations of the function. Variables cannot be saved, as they do not produce meaningful time series.

5.3.4 Add a descending Obj. (Ctrl+d)

Add a new type of object to the object shown in the browser. The new (empty) object will be located as contained in the object shown by the browser. The label of the object must be a string without special characters.

5.3.5 Insert a new parent

Insert a new object above the currently shown object, so that the latter will have the new object as parent. For example, if object *A* contains object *B*, and this command is used when the browser shows *B*, the new object will be located as contained in *A* and will contain *B*.

5.3.6 Change obj. Name

Modify the label of the object shown by the browser. Inserting an empty string delete the object from the model.

5.3.7   Set equation file label

Define the file name of the equation file. This field is automatically filled by LMM when generating the L̄SD model program, and it is subequently stored in the configuration file, normally transmitted in different configuration files without need for the user to change it. The file name is used only for presentation purposes, in that the L̄SD model program cannot modify the equations of the model. The equation file is used to generate the automatic documentation.

5.3.8   Ignore equation file controls

A copy of the equation file is contained in the configuration file, and it may be compared with the actual equation file of the model. If this option is not checked, the L̄SD model program compares the equation file of a configuration with that of the actual equation file. If the two files differ, a warning is issued.

5.3.9   Upload equation file

Store in the current configuration the equation file present in the model directory. This file is used only for presentation and control, not for actual computations.

5.3.10   Offload the equation file

Generate a new file exporting the part of the configuration containing the equation file. This command is used to transfer a model, by sending only the configuration file and, with this command, re-creating the equation file.

5.3.11   Compare eq. files

Generate a temporary file with the equation file stored in the configuration file, and compare (using TkDiff) this file with the actual equation file present in the model directory. TkDiff highlghts the difference in two text files, smartly managing lightly modified texts.

5.3.12   Generate automatic documentation

Any element in a L̄SD model is associated to a text that the modeller can use to document that element. This text is then used for the model report and, in any case, store in the configuration file for potential users of the model.

However most of the descriptions' content can be automatically derived, or copied from previously written text. Using this command the system deletes the description of the elements and replace them with the automatically generated ones.

For variables and functions the description is copied from the earliest commented lines located in the equation file, just under the line stating the beginning of the equation for that element. Normally, this is the very only text that modellers need to write (and keep updated) concerning an element.

For objects and parameters the system reports the list of the elements whose equations include the label of the object or parameter. This last part of the description is optional, so that users may update only the text associated to variables and functions.

5.3.13   Create report

Generate automatically the L̄SD model report, a documentation of the model in HTML format that links together all the information on the model elements and their interactions. See the paragraph on menu item **Help/Model report** for more details.

The report is given by default the title, as well as the file name, of the the configuration currently loaded. Users can choose different titles and different file names to store the report. The report contains several lists of elements. When creating the report it is possible to choose between lists presented as pull-down menus or simple lists of links. Moreover, the report normally includes an
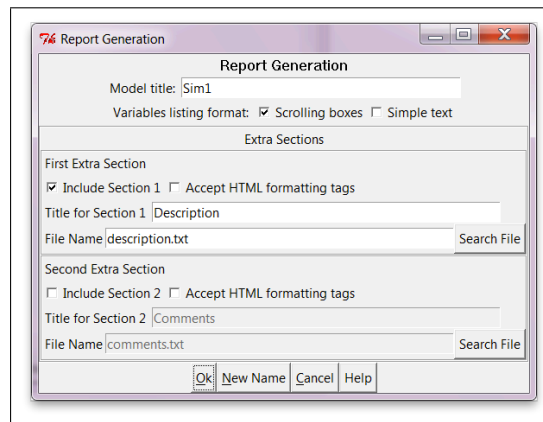
Figure 5.5: Creating a LSD report the user can set a few options concerning the report naming and additional files to insert in the report.

introductory text, and the user can choose whether to use one, two, or no file to copy the text from.

5.3.14 Create LaTex report

This commands generates a report in Latex format. These reports contain the same information as the LSD reports, but for the sets of initial values used. Latex reports are composed by sets of tables in Latex format containing the list of elements for each object, and the documentation for each element. Latex report should be edited as appropriate, and then inserted in a Latex document.

5.3.15 Find an element of the model (Ctrl+f)

Search the model with an element with the specified label. The browser will show the object containing the element.

5.4 Menu Data

This menu contains the commands related to the numerical content of the model. The data concern the initialization of the model before a simulation run, the analysis of results produced by a simulation run, and the observation of the current state of the model.

There are two types of data necessary to execute a simulation run and that can potentially affect the results. First, the user needs to specify the number of each type of object. Second, numerical values must be assigned to all the elements requiring a value at the start of a simulation run, i.e. at `t=1`: parameters, lagged values for variables and functions.

Below we list the different commands present in the menu and describe in detail the use of the modules triggered by those commands

5.4.1 Set number of objects (Ctrl+o)

This command allows to define the number of objects in the model. There are two options available. The first option (**All types of objects**) allows to determine the number of objects for the whole model structure. This command permits to customize potentially complex model structures, such as diverse numbers of objects for different branches of the model. For example, in a model with objects **Market** containing sets of objects **Firm**, allow to define many **Market**'s, each containing a different number of **Firm**'s.

The second option (**Only current type of objects**), allows to determine only the number for all the groups of objects of the same types as the objects shown in the browser. This second option is obviously more limited, but it is much faster to implement, and it is the only practical way to define very large number of objects whose sheer number would crash the initialization interfaces.

Notice that for very specialized initialization there is a third way to solve the problem of complex initialization. In many models it is possible to devote a L͜SD variable to implement the initialization.

See the section on 5.7 (p. 89) for the use of the module on setting the number of objects.

5.4.2 Init. values (Ctrl+i)

The initial values necessary to start a simulation are the values of parameters and lagged variables (and functions). Lagged elements require at least one value because they are requested in the equations with a lag, that is, for example, with their value at $t - 1$. At the very first step of the simulation ($t = 1$) it is obviously not available any value for previous (non-existent) step and therefore the system needs the modeler to provide these values. If a variable or a function is used with more than one lag, then the modeler must provide as many values as the lags.

Using this command the user activated a module of the L͜SD model program allowing to set the initial values for all the necessary elements contained in all the copies of one object type. See paragraph 5.8 (p. 91).

5.4.3 Sensitivity (parallel)

In some cases users wish to test the combined effects of a set of parameters on model results. For this purpose it is necessary to create many copies of the same configurations each containing a different combination of the values for the set of the chosen parameters. The L͜SD model program is able to generate these configurations starting from the set of values for each parameter inserted in the appropriate option as initial values (see paragraph 5.8.1 (p. 92). In that case user specifies for each parameter how many different values must be tested, and the system automatically generated all the possible combinations. For example, if you wish to test the results by the combinations of parameters α, β and γ for respectively 3, 7 and 2 values, than you will have to produce 3x7x2=42 configurations. Using this option the system automatically multiplies the objects involved generating 42 different copies each containing one combination of parameters.

This option is feasible only for a models of limited dimensions and/or relatively few number of values. For large models and/or large number of combinations the same results is obtained using the sequential version of sensitivity.

5.4.4 Sensitivity (sequential)

This option performs the same function as for the parallel case, but instead of generating a single configuration with multiple copies of the model structure, it generates multiple configurations each with the same model structure and with the different parameters combination. The resulting set of files can be used to run L͜SD in batch mode (see paragraph 4.4.11, p. 74) to be executed sequentially with separate result files generated at the end of each simulation run.

Contrary to the case of parallel simulation run, this method poses far less dimensional constraints on the simulation model.

The use of this command will also generate a text file (named `sensitivity_sq.txt`) reporting the elements used for the initialization, including how many values and the list of values for each element.

5.4.5 Analysis of Results (Ctrl+a)

This command activates the module **Analysis of Results** that allows the user to analyse the results of simulation runs. The data may come from the simulation just finished, from previously saved simulation results, or both. The module is very efficient being able to manage huge datasets and provides essential manipulation of data, such as graphical presentations and a few descriptive statistics.

See the section 5.9 (p. 93) for details the module of **Analysis of Results**.

5.4.6   Save Results (Ctrl+z)

This command generates a L^{S}D result file containing all the values saved during a simulation run. Users are requested to provide a name, to which the system attach the extension `.res`. This file can be loaded into the module **Analysis of Results** for later analysis.

The command generates also a L^{S}D configuration file (`.lsd`) with the same name as the result file, so that the user can store both the simulation results and the configuration that generated them.

5.4.7   Data Browse (Ctrl+b)

This command switch the browsing of the model from the standard browser to a detailed, object-by-object perspective. While the standard browser does not show the content and the number of object types, the *data* browser allows to navigate the model inspecting each single copy of the objects, and observe the actual values contained in each element of the model.

The data browse is mostly used at the end of a simulation to inspect the state of the model in any detail. The interface is identical to the L^{S}D Debugger, but for the commands referring to the control of the simulation run (e.g. **Run**, **Step**, etc.). See paragraph 5.10 (p. 105) for instructions on the L^{S}D debugger and data browser.

5.5   Menu Run

This menu contains all commands affecting the way a simulation is run not directly referred to the model content, as the number of time steps, number of simulation runs, etc.

5.5.1   Run (Ctrl+r)

This command starts a simulation run. Before executing the very first step of the simulation the system writes the configuration file for the model contained in the L^{S}D model program. This ensures that it is always possible to replicate the same simulation by re-loading the model from the configuration file. To avoid overwriting an existing configuration file users need to change the name of the configuration before running the simulation.

The system prevents to run a simulation using a model that just completed a simulation run. This is because the state of the model differ from its configuration file. If the user wishes to do so, it is necessary firstly to save the state of the model (at the end of a run) as a new configuration, and then load the resulting file.

Activating this command the system asks for confirmation of the execution summarizing the action it is going to perform. In case of a single simulation run the results will be stored in memory only, to be analysed at the end of the simulation with the analysis of result module. In case of multiple simulation runs the system will create many files, one for each run, containing the results for that run.

5.5.2   Set sim. settings (Ctrl+m)

This commands allows to configure options for simulation run not represented within a the definition or initialization of the model. The available options are the following (see figure 5.6):

Number of simulations This value determines how many simulations must be run in sequence. A sequence of simulations consists in many simulations run with the same configuration but using different sets of random numbers. Multiple simulation runs are therefore used to test the robustness of results against variations due to random events.

If this option indicates a single simulation run (value 1), then the system will execute the simulation and, at the end, will retain the simulation results in memory, to be analysed by the Analysis of Results module. If more than one simulation is requested the system produces the first simulation, saves the results in a L^{S}D result file, re-load the configuration and execute again a simulation run after increasing the seed for the pseudo-random number generator, in effect using a different set of random values.
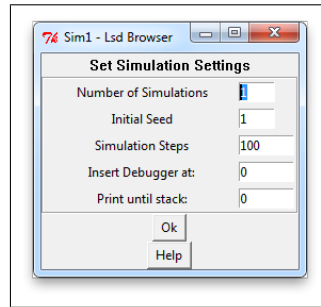
Figure 5.6: Simulation options, setting the environment conditions for the model.

The result files saved during multiple simulation runs are labeled after the configuration file, extended with the number of the pseudo-random seed generator. Therefore, any simulation saved during a multiple run can be reproduced using the configuration and the seed indicated. For example, if the configuration contained in the file `MyConfig.lsd` is defined to run 10 simulations with initial seed 2, the simulation results will be stored in files named from `MyConfig_2.res` to `MyConfig_11.res`. The result files can be loaded collectively in the Analysis of Results module analysing single simulation runs or comparing the patterns across different runs.

Collective simulation runs generate also a file (named after the configuration and extension `.tot`) containing the very final value for each element in each line, and as many lines as the number of simulation runs. This file can be loaded also in the Analysis of Result module, although for this case the "time steps" will actually refer to different runs.

It is worth notice that multiple simulation runs are rarely necessary in L^SD. Robustness against random variations may be produced simply multiplying the number of objects in the model. Multiple runs are necessary only when the model dimensions are so large that they wouldn't fit in the computer memory, but are small enough that it is not worth to run in batch mode.

Initial seed The random numbers used in stochastic simulations are in effect pseudo-random values generated by means of deterministic functions. This functions produce sequences of values that have the statistical properties of the random function requested. The *seed* for pseudo-random functions initialize the function so that it will generate a given series of (pseudo-)random values. Using twice the same seed will therefore generate the same series of values. When a simulation starts the L^SD model program initializes the pseudo-random functions with the seed specified in this box. Therefore, the same model using the same random values will replicate exactly the same result. Using different seed will instead use different pseudo-random values.

Simulation steps Number of time steps to be executed in the simulation run. Note that the modeler may use a command in the equations to close the simulation run in advance, typically under certain conditions. Moreover, users can choose to stop a simulation run at any moment.

Insert debugger at Indicate the simulation time step at which to enter in debug mode. At the indicated time step the model will start controlling whether the variables or functions computed have been marked as to be debugged. If one of these is encountered, the system interrupts the simulation just at the end of the equation computing the new value for the element. When the simulation is interrupted in debug mode (which may happen also when an error occurs, such as a division by zero) the system shows the Lsd Debugger interface, showing every detail of the model (see paragraph 5.10, p. 105).

Inserting in this box values lesser than 1 or above the maximum number of time steps have no effect.

Print until stack In some cases, generally for optimizing purposes, it may be relevant to know the exact order of completion of equations within a time step. This order is determined by the L^SD simulation manager at run time, depending on the state of the model at each time step. The L^SD simulation manager tries to compute the equations for all the variables in any object

of the model, starting from **Root**, and then going "down" in each object. Normally, equations for variables contained in "higher" objects are executed before those for variables of lower objects. However, in some cases this is impossible, since the equations for high objects' variables necessitate the updated value of variables contained in lower objects.

An equation (say X) being executed because of normal updating required by the LSD simulation manager is said to be executed at "stack 0". If such an equation requires the value of another variable (say Y) not yet updated, the equation for X cannot be completed. It is therefore interrupted (placed on the *stack*), and the equation for Y starts to be executed. This latter equation is said to be executed at stack 1. In turns equations computed at stack 1 may require the values of other values, which will be produced by equations computed at the next stack.

Reading the print-out of the stacks involved in a simulation step, and the label of the variables computed, allow to interpret the order of completion the LSD simulation manager managed to find (among the generally many feasible).

This option allows to specify the "depth" of the stack one is interested to read. Setting this value at, say, k will generate in the log window a line for each element computed at level k or below. Each line will indicate various information, including the equation result value, the "triggering" variable (which variable at stack level $k-1$ requested the value) and the identification tag of the objects containing the variables. Moreover, the lines will also provide the time necessary for the equation to be completed.

5.5.3   Remove debug flags

Variables and functions marked to be debugged cause the simulation to stop when the debug mode is inserted. This command removes all the debug marks, or flags, from every element of the model.

5.5.4   Remove save flags

The values generated for any element during a simulation run are normally deleted as soon as they are no more necessary to compute the subsequent steps. The modellers need to explicitly indicate which elements' values need to be saved for post-simulation analysis. This command removes all the flags for every element of the model possible marked to be saved. If a simulation is run just after this command, no data will be available for analysis.

5.5.5   Remove plot flags

During a simulation run it is possible to generate a dynamic graph for the values of some elements. Using this command all elements possible marked to be plotted dynamically during a simulation run are removed, and no dynamic graph, or run time plot, will be produced.

5.5.6   Show elements saved

This command generates a list in the log window for all the elements marked to be saved in the model. It has no effect on the configuration of the model.

5.5.7   Show elements to observe

Modellers can mark some elements of the model as series relevant to be analysed. This option is independent from the indications of which elements are actually saved during a simulation run, so that a modeller may indicate with the **observe** flags the meaningful elements for the model, while setting a different set of values to be saved.

This command generates a list in the log window for all the elements marked as worth to be observed in the model. It has no effect on the configuration of the model, but the list of elements worth to be observed is given a prominent role in the automatic documentation of the model.

5.5.8 Show elements to initialize

By defaul L^{S}D forces the modeller to assign initial values to all the parameter of the model and all the lagged variables. However, frequently the initial values for many of such elements have no effect on the model results, because, for example, the model generate its own "customized" initialization. The user can indicate the list of the elements whose initialization is mostly relevant for the model results, as marking them "to be initialized".

This command generates a list in the log window for all the elements marked whose initialization is considered as particularly relevant. It has no effect on the configuration of the model, though the elements marked as relevant to initialize are given a prominent role in the automatic documentation of the model.

5.5.9 Remove run time plots

The windows containing the dynamic graphs during simulation runs are "sticky" and cannot be removed by the usual commands for closing windows. This command removes permanently all the run time plot windows from the screen.

5.6 Menu Help

Users may need assistance on the use of the model in two respects. Firstly, they may need help on the use of the L^{S}D interfaces, commands, meaning of error messages, and any other possible aspect of the L^{S}D system in general. Secondly, they may need to understand aspects of the model not easily accessible by the L^{S}D browser. For the first type of issues the entry **L^{S}D help** gives access to the L^{S}D manual included in the L^{S}D distribution. Concerning the information on the model use the second entry **Model Report**, which opens the model report for the configuration used.

Figure 5.7: L^{S}D model reports are HTML files automatically created describing the elements contained in the model and presenting them in a variety of formats, from textual only to the actual code and values used.

L^{S}D model reports are HTML files containing all the information contained in the model, presented in different formats, and enriched by hyperlinks allowing to follow the model description according to the preference of the reader.

In a model report there is an introductory section, and three sections for as many lists of the model elements, with links among related elements within the same sections (e.g. parameters used in an equation) or to other sections (for the same element).

The introductory section includes a textual description of the model and list the main elements to initialize and those containing the main results. As for the other lists the entities are grouped in tables listing sequentially the elements of a single object. Links allow to jump to related objects and/or elements.

The second list provides for each variable and function the actual code used for to compute the element, including also the list of the elements whose values are used in the code. For any element, this list provides the list of equations making use of their values.

The third list includes all the initial values for the elements whose initialization may affect the model.

The structure of the model report is such that users with different programming experience can be provided with the best suited form of documentation. Moreover, the hypertextual format allows users to skip through technical variables, such as averages etc., and focusing only on the interesting parts of the model. Lastly, model reports are easy to transfer and use (basically, they are text files, opened with any web browser), and easy to generate.

If the model report does not exist, the user can open another HTML file, possibly referring to the report of another configuration of the same model. In generaly, however, it is necessary to create a model report from the scratch. This can be done using menu **Model**, which allows to create the model documentation automatically (endowing the model elements with all the information automatically accessible) and the create the model report.

5.7   Module **Set Objects' number**

This module determines the number of copies for the objects in the model.
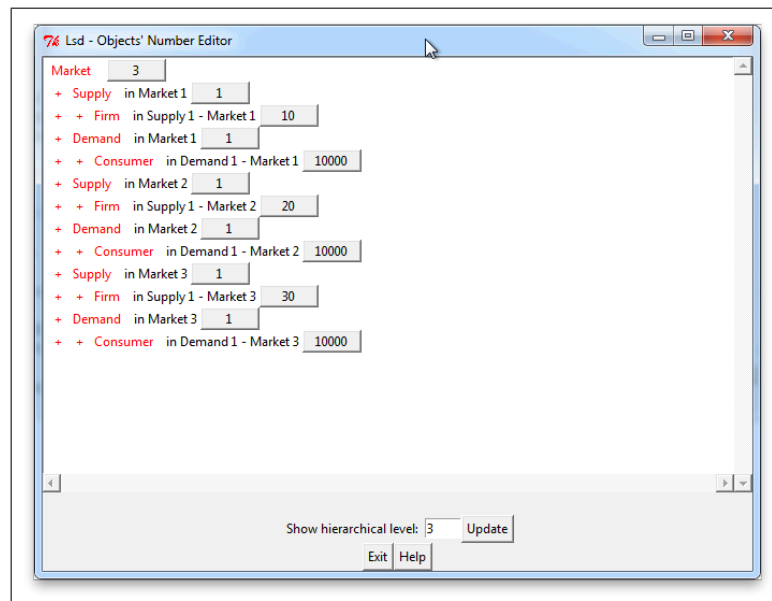
Figure 5.8: The main window to set the number of objects presents each type of object with their number of copies in the present model configuration. Clicking on this number it is possible to modify the number of copies. Note that the window may be set to hide the number of lower hierarchical levels, in order to identify more quickly the groups of higher level objects.

The module report the number of copies in different lines. The labels of the objects are indented on the right following their parent objects, that it, the objects they descend from. Given the number of objects up in the hierarchy, there will also be consequently groups of descending objects. Hence, each line refers to the ordinal number of the copy each group of object descends from.

Clicking on a group of objects, contained in specific copies of higher order objects, it is possible to set the number of objects for that "branch" of the model tree independently from the other groups of the same object in other parts of the model.

Besides the setting for individual groups of objects, the module offers also the possibility to modify at once every group of that object in different "branches" of the model structure. Suppose your model includes 3 copies for *Market*, each containing 1 copy of *Supply*. In the 3 resulting groups of objects there can be a different number of copies of *Firm*. Suppose that now you want to increase the number of *Firm*'s to 200, but not only in a single copy of *Market*, but in all the copies of *Market* of the model. As indicated in figure 5.9 it is possible to mark the option to extend the chosen number of descendants to the whole model, or to the specific branch of the model structure.
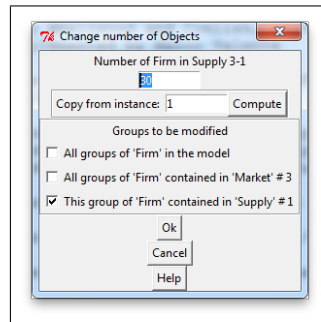
Figure 5.9: When setting the number of a group of object contained in a complex model structure, it is possible to apply the new number of objects to all groups of the concerned objects at different levels in the model.

The interface determining the scope of the changes allows to choose not only among the objects one layer above, as in the example case, but to any possible higher layer. For example, suppose that *Market* is defined as contained in objects of type *Country*. One may decide whether the change should be applied to a single *Market*; to all *Market*'s in a single *Country*; or to all *Market*'s in all *Country*'s.

The system always appends new copies after the set of previously existing copies. The new copies of the objects will replicate the content (e.g. parameters' values, number of descendants, etc.) of an example object. By default the system uses the very first copy of the object as example, but users can choose a different one. This is done inserting the ordinal number of the desired example copy in the field **Copy from instance:** . For example, if the first *Market* contains 10 *Firm*'s and the user wishes to use the first object in the second *Market*, then the field should indicate 11. For complex models the ordinal number of a specific copy may be difficult to calculate. Clicking on **Compute** helps to find out the ordinal number of a copy by specifying the ordinal numbers by different layers of the model.

In case the new number of copies of an object is smaller than the previous value, the system by default would remove the last copies in the set of objects, but the user can indicate specifically which copies must be removed. For example, suppose you have a model containing three copies of an object. Setting one to the copies of this object it is necessary to remove two previously existing copies. If the system decides automatically the copies to remove, it applies the rule that the last copies in the sequence of previously existing objects should be removed. Therefore, the copies removed will be the second and the third, leaving only the first copy. Alternatively, the user can decide to choose the copies to remove. A sequence of windows will ask the ordinal numbers of the copies to remove. For example, the user can indicate the first and third copy to remove, and the new configuration will retain only the second copy. The option to specify each individual copy to delete cannot be used when the new number of objects is applied to several groups, for example, to all groups of *Firm*'s in the previous example. In this cases, the system always remove the last copies in each group.

5.8 Module Initial values

This module allows insert the initial values for all the copies of a specific object. The initial values for a simulations are all the parameters, lagged variables and lagged functions present in the model, that is the values necessary to compute the equations at the initial time step(s).

Parameters obviously need to be assigned an initial value[1]. Lagged variables are variables whose values are used in some equations of the model with a lag, that is, it is the values from previous time steps. In the earliest time step of the simulation, $t = 1$, there are no past values to be used, and therefore the user must explicit provides this value, to be associated to the elements' values at time $t = 0$, $t = -1$, etc.

Lagged functions are functions whose values are required for previous computation of the function. In this case, the very first time function's values are used there are no previous values to be supplied to the "calling" equation, and, again, the past values must be initialized.
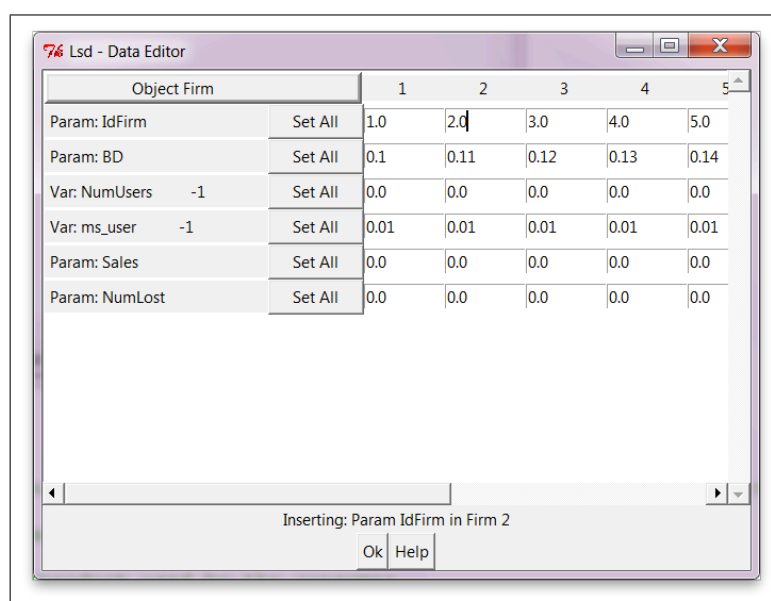
Figure 5.10: Module to set initial values for the elements requiring initializations within one type of object. Elements are presented in rows and each column refers to a copy of the object in the model.

The module to insert initial values concerns all elements to be initialized contained in one single type of object. The module's interface provides one row for each element to initialize (and each lag for past variables' and functions'). The columns refer to the different copies of the object. They are identified with a "tag", a combination of digits referring to the ordinal copy of the objects containing the elements and all objects in higher layers. For example, a model containing objects **Market** in turn containing objects **Firm**, the columns for the initial values of elements in **Firm**'s will contain two digits: the first for the ordinal copy of **Market** and the second for the ordinal copy of **Firm** in that copy of market. A tag may for example be $12 - 25$, indicating the initial value for the element indicated in the row and the copy contained in the 25th copy of **Firm** among the set of this object contained in the 12th **Market**. A similar model composed by objects **Country** containing **Markets**, then the initial values for **Firm** will be composed by three digits, as in $2 - 12 - 25$: 2nd **Country**, 12th **Market**, 25th **Firm**. The tag for the cell with the cursor is reported in the bottom line of the initial values interface

The interface for the initial values allows to insert manually the initial values for an element. Typing one value and pressing the key **Enter** moves to the next cell. This interface can contain a maximum of 100 columns, assuming that larger number of initial values will never be assigned manually, but using an initialization function.

---

[1]Note that modelers may have equations to overwrite the value for parameters, though this may produce uncertain results given that the system is not able to control the timing of parameters' value changes.

5.8.1 Initialization functions

The initialization functions are mathematical expressions assigning the values to the sequence of initial values for an element, that is, all the initial values for, say, a parameter in every copy of the object containing it. For example, an initialization function may consists in assigning identical initial values to all the copies of the element. Initialization functions obviously affect all copies of the element to initialze, even beyond the 100 shown in the module's interface.
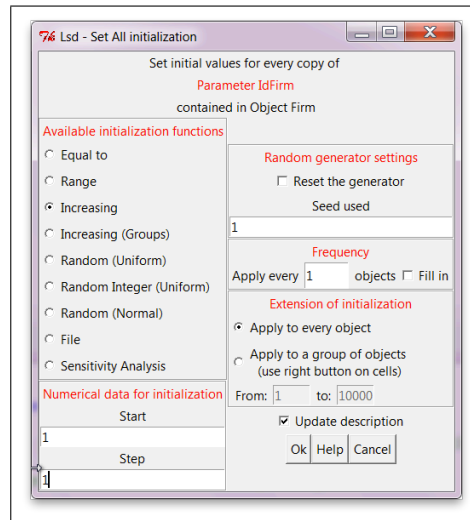
Figure 5.11: The **Set all** buttons permit to apply an initialization function to all the copies of one element.

The initialization function for an element to initialized are determined using the button **Set all** appearing in the beginning of each row, just after the label of the element. This interface is very flexible, providing an extended set of different initialization function, besides the possibility to control which copy of the elements need to be assigned with that function. For example, one may use a given initialization to the first half of the elements and another for the second half.

The **Set all** interface is composed by three sections: the nature of the initialization function; the values of parameters for the initialization function; the choice of pseudo-random values; frequency of application; the extension of its application.

The initialization functions available are the following, each of which uses the two cells containing numerical values in different ways. In the following we will refer to these numerical values as those indicated in the first and second cell.

Equal to. Assign the same value, as indicated in the first cell, to all the copies of the element.

Range. Assign equally spaced values starting from the value in the first cell and finishing to the value of the second cell. For example, suppose that there are 100 copies of the element to initialize, and the initialization function **Range** is used with values from 0 to 100. Then, the first and last copy of the element will be 0 and 100, respectively. The intermediate cells will have equally spaced values: 0, 1.010101, 2.020202, 3.030303, etc.

Increasing. Assign increasing values to the copies of the element to initialize starting from the value indicated in the first cell and increasing of the value in the second cell for each subsequent copy. For example, using **Increasing** and setting the values of the data cells to 10 and 1 respectively, the initial values will be: 10, 11, 12, etc.

Increasing (group). This initialization function is identical to the **Increasing** one, but it is re-set to the starting value for each separated group of objects. For example, consider a model composed by objects **Firm** contained in **Market**. Suppose there are many copies of **Market**, each containing many copies of **Firm**. Initializing an element contained in **Firm**, using this initialization function will assign increasing values to the set of elements contained in the first **Market**. Then, the initial value for the copy of the element in the first **Firm** contained in the second **Market** is reset to the starting value, equal to the value used for the first **Firm** in the first **Market**.

Random (uniform). Assign a real number as a random value for each element to initialize, using a uniform random function whose lower and upper limits are determined by the values in the first and second cells respectively.

Random integer (uniform). Generate an integer number as a random value for each element to initialize, using a uniform random function whose lower and upper limits (included) are determined by the values in the first and second cells respectively.

Random (normal). Generate a real number as a random value for each element to initialize, using a normal random function whose mean and deviation are those indicated in the first and second cell respectively.

File. Load the initialization values from a file. Files for initial values must be text files containing one single column. The first element of the column is ignored, assuming to be a label.

Sensitivity. The user is requested to indicate in the first cell the number of values to use for the element. On confirmation a new window will ask for the list of the values, separated by newline or spaces (may even be pasted from the clipboard). If the list contains fewer value than those specified in the first cell the remaining ones will be considered to be 0. This initialization does not modify the initial values of the model, but allows to generate automatically configurations for sensitivity analysis. When two or more elements are initialized, the **Data / Sensitivity** (see paragraphs 5.4.3, p. 84) creates configurations including all possible combinations for the values of the elements initialized as to be included in the sensitivity analysis.

The random initialization functions can be assigned using a specific seed generator. Using a specific initialization ensures that the random values they are drawn from the same or a different (pseudo-)random sequence as other initializations. Checking on the option to use the seed, the random sequence will be regenerated using that seed.

The two sections in the bottom of the initialization window allow to specify which sub-set of the copies of the element must be initialized according to the specified function. The first of these section concerns the **Frequency** of the initialization, while the second concerns the **Extension**.

By default, the system applies the function to every copy of the element, that is, it has a frequency of 1. Setting a higher frequency N (say, for example, 3), the initialization function will regularly skip $N - 1$ copies and apply the initialization rule to the Nth copy, in the example the third. If the option **Fill in** is not checked, the copies in between those initialized are not set, leaving the previously existing values. If, instead, this option is marked, the intermediate copies will be set with the values previously generated by the initialization function. For example, suppose to have selected the function **Increasing**, starting from 10 and with step 1. If the frequency is 3 and the option **Fill in** is on, then the sequence of initial values for the copies of the elements will be: 10, 10, 10, 11, 11, 11, 12, 12, 12, etc.

Concerning the extension, the user can choose to apply the function to all the copies of the element in the model, or only to a range of contiguous elements (that is, of elements contained in contiguous objects). The two cells in this section refer to the first and last object containing the element of the model, which must be indicated with the sequential number of the objects. To identify the sequential number it is possible to right-click the cells opening the interface to individuate the sequential number of an object by giving the ordinal number of the higher level objects (see figure 5.9 at page 90 and the related text).

The final option **Update description** does not affect the values inserted in the model but only the documentation. If checked on, the system will automatically update the description of the initialization concerning the element, specifying the function used and its values. If the function used concern all the elements, then the new description will replace the previous one. Otherwise, it will be appended, possibly requiring the editing of the modeller in case of potential confusing text.

5.9 Module Analysis of Results

The analysis of results module is designed to present the data produced in (one or more) simulation run(s) in formats suitable for the purposes of the researcher. Here is summary of the functions of the module:
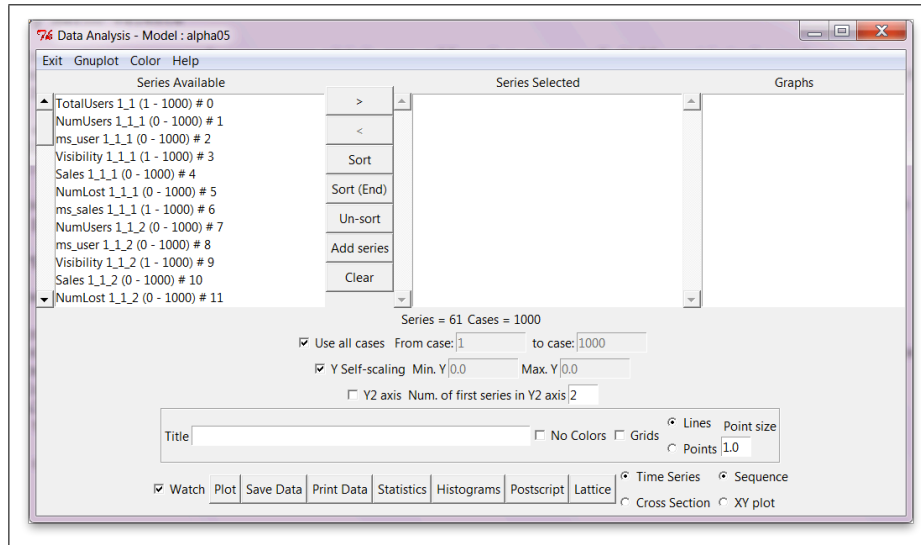
Figure 5.12: Main Analysis of Result window. Series available are listed in the left-hand box, series to be processed in the central box, and graph produced in the right-and box. The user needs to move some of the series in the central box, select the options and the operation required. The analysis of results module is highly efficient in managing large data sets.

- Time series plots. Variables are plotted across time steps.

- Cross-section plots. Values of different variables are plotted at the same time step.

- Scatter plots. Variables' values are represented as function of other variables' values, generating both bi- and tri-dimensional graphical representations.

- Phase diagrams. Values of a variable at time $t + 1$ are represented as function of the same variable's value at time t.

- Frequency histograms. Values across time for a variable, or across variables at a given time, are counted and grouped in frequency classes.

- Lattices. Values stored in matrices are plotted as lattices.

- Temporal or cross-section statistics. Descriptive statistics as average and variances can be produced across times or across variables.

- Options. It is possible to let the system automatically assess the scale of the graphs, or force the extremes of the axis. Other options include the symbols for series (lines or points), grids, colors, labels, etc.

- Exporting graphs. Plots can be exported as encapsulated postscript files.

- Exporting data. Series can be exported as text files in a variety of formats (e.g. choosing a separator, or fixed-column lenght; with or without labels, etc.).

- Gnuplot. The MS Windows distribution includes gnuplot, a specialized graphical package, to produce advanced graphs, for example on data exported from L^{S}D simulations.

The typical use of the Analysis of Results module consists in selecting one or, more in general, a set of series to process; set the options for the type of operation desired; generate the results (graph or statistics); replace the series and continue the process. Eventually, some graphs or data may be exported, or the user returns to the L^{S}D browser to generate a new configuration and running new simulations.

In the following we report the instructions to perform the operations allowed by the module.

5.9.1 Selecting series to process

The first operation in using the Analysis of Result module (AoR) consists in individuating the meaning of the available series and moving the selected ones in the central listbox.

A simulation model may generate large data sets, composed by several thousands of series each containing thousands of data. For this purpose, the module includes also a powerful selection interfaces, allowing to identify a specific set of series by their labels, values, or positions in the model structure. For this purpose, the system identifies each series by a unique combination of labels and digits, which can be exploited by the user to select series to process with many different criteria. The label of a series contains the following elements:

```
Label TAG (TimeStart - TimeEnd) # order
```

The first components in a series identification is, obviously, its label. Secondly, each series is associated to a "tag", a group of digits identifying the "branch" of the model structure in which its object containing the element generating the series was contained. For example, a tag as X_Y_Z indicates that the variable was contained in an object placed at the third layer in the model hierarchy (**Root** is the only object at level 0, level 1 refers to objects contained in **Root**, level 2 their descendants, etc.). That particular series was contained in the Zth copy of object at level 3, which descended from the Yth copy of the object at level 2, which, finally, was contained in the Xth copy of the object at level 1. Note that if a level contains one single instance of an object, then its digit may be skipped, since it is not necessary to identify different series in descending objects.

Finally, the identification includes the time steps at which the copy of the element (that is, the copy of the object containing the element) started to exist in the model and the date at which it was removed.

The last code is unique ordering value, used for technical purposes and of no interest to users.

The first step in using the Analysis of Result module consists in selecting a group of series from the left box **Series Available** and placing them in the central box **Series Selected**. In order to access the series one can scroll the **Series Available** box, and sort its content in three different modes. Firstly, the unsorted mode (used by default, and activated by pressing button **Un-sort**), list the series according to their position in the model: first the series generated by elements in high level objects, and followed by their descendants, then series from elements from the subsequent objects. This sorting mode is useful since arranges the series in a sequence resembling the model hierarchy.

Secondly, it is possible to sort the series in the ascending alphabetical order of their labels. This model is useful to identify a series without knowing their location in the model structure. Press button **Sort** to generate this sorting.

Thirdly, it is possible to sort series in alphabetical order but putting firstly all the series contained objects still present in the model (not removed) at the final time of the simulation, and then the series according to the descending time step of their removal. This sorting mode (activated by pressing **Sort (end)**) is useful when the model stores many variables from objects created and destroyed during a simulation run.

After having sorted the sets of series available as suited, the selection of the series one wants to process can take place by the usual selection standards: click and drag, use of the **Ctrl** key to add a new item to the selection; use of the key **Shift** to add sets of items to the selection and removing previous ones, etc. Double-clicking on a series moves this directly to the central box, while it is necessary to click on the button > to move there the whole selection from the left box.

5.9.2 Advanced selection

The system to select and move the series is practical only insofar users need to select a few series. However, LSD simulations can produce hundreds or thousands of series, and the user may need to select all of them, or a well specified sub-set. In these cases, using the mouse to select the series is impractical, or totally unfeasible. An alternative selection mechanism consists in pressing the right-button of the mouse on one of the series that the user needs to select.

This selection mechanism allows to choose the series according to three possible criteria of filtering the series with the specified label.
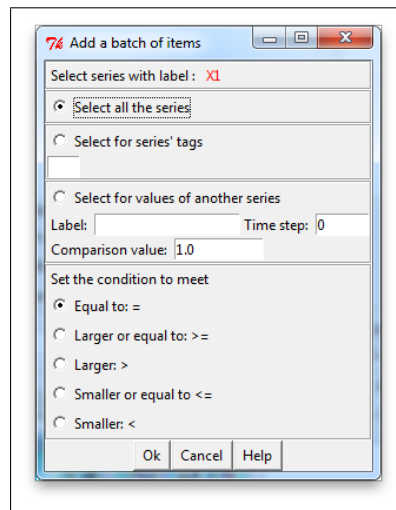
Figure 5.13: Clicking with the right button on a series a powerful selection mechanism allows to choose among the series available with the same label.

Firstly, and most frequently used, the option **Select all the series** allows to select all the series with the specified label in the model. As mentioned, the label concerns the series the mouse has been right-clicked, so that, in effect, the user needs not to type any text at all. This is the default option.

Secondly, it is possible to filter the series with the specified label according to the tags, that is, to the obejcts containing the series. This option is used by marking the window section **Select for series' tags**. This section contains a number of entry cells equal to the hierarchical level of the object containing the element specified (in the example, three cells since the series specified concern an element located in the third level). As mentioned above, this number is equal to the number of digits forming a series' tag. The user can enter integer values in one or more of the cells, or leave them empty. The values entered in these entries will be used as reference values for the condition chosen in the last section of the window **Set condition to meet**. For example, using the default condition **Equal to: =**, the system will select all the series having the elements of the tag equal to the value(s) entered in the entry cells (empty cells are read as accepted condition). Using, instead, condition **Larger: >** the system would select all the series with tag's values higher than those specified in the entry cells.

The third filtering mechanisms (**Select for values of another series**) is based on the conditional filtering based on the values of some element of the model, whose series is among those available. The user must enter: the label of the element by which the to filter the series; the time step of the series to consider; and a reference value. The elements used for the conditional part must be located in the same objects as the elements whose series are selected. The system will compute for each copy of the objects the value for the conditional element (at the specified time step). It will then evaluate whether this element's value satisfies or not the condition indicated in the last section, as compared to the reference value. Only the series whose associated element satisfy the condition will be selected.

For example, consider to use this filtering system and specifying: label for the filtering element **Age**, time step 100, comparison value 10, and condition **Smaller: <**. The system will select only the series with the specified label being contained in the objects that contained the element **Age** at time 100 with values smaller than 10.

5.9.3 Graphs general options

The procedures to create any type of graph follows the same steps: insert one or more series in the **Series selected** box; set the options for the graph; set the options for the type of graph; press button **Plot**. For all graphs the user can set the following options.

Use all cases - **From case ... to case ...**. If the first option is used the system will use all the

data available for the elements plotted. If different series have different starting and final times the time series graphs will ignore data for missing times. For cross-section graphs series missing the relevant data will be ignored. Using the second option (relevant only for time series graphs), the user must specify the time step to be used as origin of the graph and that for the last time step. Notice that when using the automatic option to use all cases, the system inserts in the cells the first and last case (i.e. time step) used.

Y self-scaling - Min. Y ... Max. Y. The first option lets the system compute automatically the vertical minimum and maximum values. Note that the maximum value is not the actual maximum value of the highest series, but extends a bit the range of the vertical axis to allow for the points plotted to appear in the window. The second option allows the user to force specified values for the minimum and maximum vertical axis.

Y2 axis. This option is used only for time series sequential plots, and requires the use of the option **Y self-scaling**. When this option is checked on the system will use two independent vertical axes. The user needs to specify the series from which the second vertical axis is used. All the series before the indicated one will make use of the first vertical axis. This plot allows to plot different series normalized on two independent scales.

Title. This entry is automatically filled with the identification label of the first series entered in the **Series selected** box. The text in this series is used to name the graph generated, and is used only to identify the different graphs. Note that each graph is assigned also a progressive index.

No color. Generates black-only or gray scale graphs, depending on the type of graph.

Grids. Plots a grid in the graph, for readability purposes.

Lines - Points. Use lines (connecting separated points) or points only. The entry cell along the points' option allows to set the points' size.

Menu item **Color**. This menu allows to modify the color associated to the indexes. This color will be used to differentiate the series according to their order.

5.9.4   Graph windows' features

The graphs generated are normally[2] stored in LSD independent windows. These windows have many features useful to interpret the results contained in the graph.

The lower border of the window shows the coordinates of the plane corresponding to the mouse pointer position. Moving the pointer over one series shows the name of the series on the lowest left corner of the window. Cross-section graphs reports also the order position of the series concerned.

Double-clicking anywhere on the graphs brings the main AoR windows on the foreground.

Pressing the **Shift** key and clicking allows to add a text label on the graph, which can be then moved around with the mouse.

In the following we report some examples of graph generated with the AoR window depending on the options chosen.

5.9.5   Graph type Time Series - Sequence

This graph generates one line for each series, placing the time on the horizontal axis.

---

[2]Some 3D graphs can be created in separate windows managed by GNUPLOT.
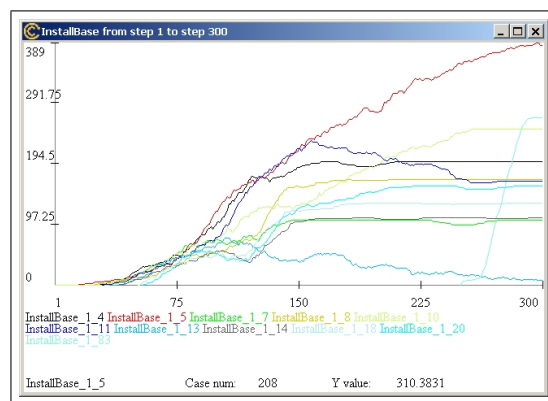
Figure 5.14: The default options produce a standard graph with time on the horizontal axis and each series selected generates one line. Moving the mouse on the area of the graph reports the pointer's coordinates, and, crossing a line, its label.

5.9.6 Graph type Cross section - Sequence

This options generate a graph containing one or more line referring each to a specified time step and reporting all the values for the different series for that time step. This graph requires additional information: the time step(s) to use and the ordering of the variables. When pressing the button **Plot** with these options a new window asks for the necessary information (figure **??**).
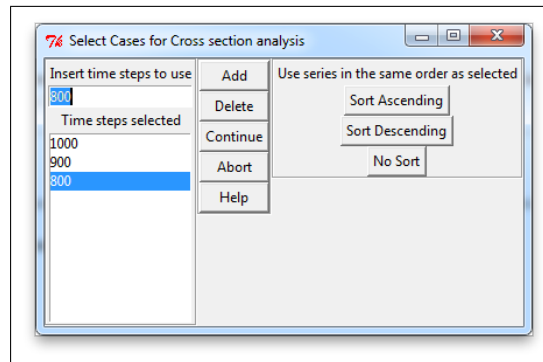
Figure 5.15: Cases to be used as "variables" for cross-section plots are inserted in this window, by typing the times to appear as lines. By default, or pressing the **No Sort** button, the order of the series on the horizontal axis is the same order of the series in the series available box. Alternatively, it is possible to sort the series in ascending or descending order pressing the relative buttons when a time step is highlighted.

The user must type at least one time step (in the top-left entry cell), and pressing the button **Add** for each of them (or just the **Enter** key). The inserted cases will appear in the box below the entry cell. Cases can be removed by selecting them and pressing **Delete**. When all the cases have been inserted the user can highlight one of them and press the one of the buttons to sort the series. The order of the series as appearing on the horizontal graph will be then re-arranged to sort the series depending on their values in respect of the chosen time step.

The window has also a hidden feature. Pressing the following combination of keys, and using the the content of the entry cell, the system automatically add whole batches of series. The combinations are the following:

- **Controf+f** (*from*): use the value as starting time.

- **Controf+t** (*to*): use the value as final time.

- **Control+z**: add to the list of time steps selected all the times from that indicated as starting time to the final time.

- **Control+x**: indicate a step, so that if 100 is the starting time and 10 is the step, the times inserted will be: 100, 110, 120, etc.

Pressing **Continue** will generate the graph, while button **Abort** terminates the window without generating the graph. The graph will show on the horizontal axis the series and will link the points corresponding to the same time step for each series.

5.9.7 Graph type Time Series - XY plot

These options produce a scatter plot where the values of some series are used as independent variables and others as dependent, creating graphs where each point is a single time step. Choosing the option **Lines** the system tries to interpolate the points forming smoth curves. It is anyway better to use the option **Points** for most of cases.

The system behaves differently depending on the number of series in the **Series Selected** listbox.
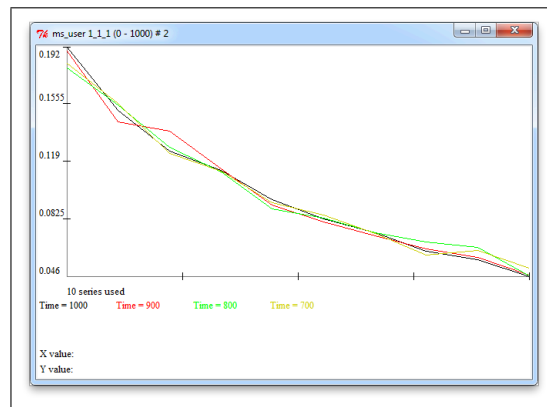
Figure 5.16: Example of a cross section sequential graph. 100 series selected have been evaluated at time 300, and ordered according to their descending values at the chosen time step. The mouse pointer crossing the line indicates the series concerned and its ranking order.

Single series

When the system finds a single series it generates a phase diagram where each point is defined at the coordinate X_t on the horizontal axis and X_{t+k} on the vertical axis. In this case the system asks for number datum from the series and the number of lags k. The system also offers the possibility to generate the graph as image in a standard LSD graph window or in a higher quality gnuplot window (default option). As all gnuplot windows there are special functions embedded in the window, such as the possibility of rotating the graph and direct exporting to file; these options and their control depend on the operative system.

Optionally, the user can plot a line in the graph placed at 45'.

Two series

When two series are present in the **Series Selected** the system directly generates the scatter plot where the values of the first series are reported on the horizontal axis and the values of the second are measured on the vertical axis.

Three or more series

When plotting this graph with more than two series selected, the module generates a new window, asking for additional information concerning the graph.

The user can choose a bi-dimensional scatter plot (2D) or a tri-dimensional (3D) one. In the first case the first series in the set of series selected will be used as independent variable (measured on the horizontal axis), while the second and following series will be plotted generating as many independent lines, one point for each time step.

Choosing 3D graphs there are several options. The default option generates a 3-dimensional space where the first and second series are used as coordinates on the plane. Any subsequent series is considered as independent surface measured on the vertical axis. Figure 5.18 shows an example of a 3D scatter plot with two independent series defined on the same plane.

The second option generates, in effect, bi-dimensional graphs placed, however, in a three dimensional graph where the time dimension is one of the two dimensions on the horizontal plane.

The third option plots each series independently as time series, using the ranked position of the series as one dimension of the plane, and the time as the second dimension.

The last technical option allows to choose between a standard LSD graphical window or a high quality gnuplot one. As all gnuplot windows there are special functions embedded in the window, such as the possibility of rotating the graph and direct exporting to file; these options and their control depend on the operative system.
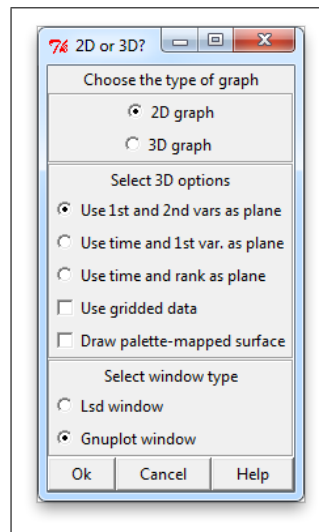
Figure 5.17: To create a scatter plot from time series the system asks whether the graph should be 2D (one independent variable) or 3D (two independent variables). For 3D graphs it is possible to use gridded data (interpolating points) and generating color mapped surfaces. Finally, the user can decide to have the graph as a standard LSD graph window (low quality), or having the graph plotted in an interactive gnuplot window.
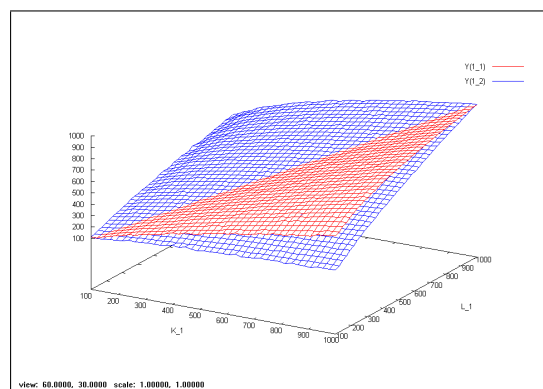
Figure 5.18: Example of a 3D scatter plot computed across time. The model generating the data is $Y_t = K_t^\alpha * L_t^{1-\alpha}$. 10000 random values for K and L are used to compute two different series of Y's using different α's. The plot is generated with lines and gridded data.

5.9.8 Graph type Cross section - XY plot

These options generate a graph at a single time step, indicated in the first cell of the option window, using the data from all the series selected. The system divides the selected series in blocks, with each block providing the sets of values to be used as variables for the axes of the graph.

In case of 2D graphs the system by default divides the available series in two halves, assuming a single dependent variable. In this case the first half provides the horizontal coordinates of the points and the second half provides the complementary vertical coordinates. Users can indicate a larger number of dependent variables. Indicating the series must provide data for k dependent variable the system divides the data in $k+1$ blocks, where the first block provides the common coordinates for the horizontal axis and the system plots k independent series on the vertical axis. The users must ensure that the number of series is a whole multiple of $k+1$; the system reports the number of points implied by the number of series selected and the indicated number of dependent variables. Figure 5.20 shows an example of a 2D graph.

In case of 3D graphs the same system applies. The system divides, by default, the series selected
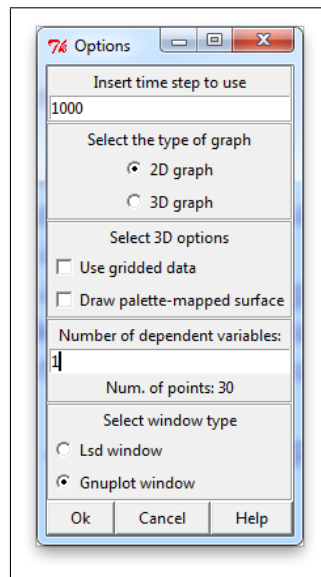
Figure 5.19: Plotting a cross section scatter plot use only one time step value from each series selected. The graph generates the horizontal axis (or axes, for 3D graphs) using the values from the first (or first two) blocks of variables, while the subsequent blocks will provide the values for the dependent variables.

in three blocks, using the first two as coordinates for the plane and the third as height of the plotted surface. Similarly to the 2D case, users can set a different number k of dependent variables. In this case the system divides the series selected in $k + 2$ groups generating the horizontal plane coordinates with the first two blocks and k independent surfaces on the vertical axis.

5.9.9   Graph type Histograms

Clicking on the button **Histograms** generates a graph containing a representation of the frequency of the values indicated in the series selected. The histograms are built dividing the range of the values used in as number of classes, as indicated by the user, with each class having the same width. For example, if the values used to generate a histograms have a minimum of 10 and a maximum of 30 and the user asked for 10 classes, the first class boundaries will be [10-12], [12,14], ..., [28-30]. The system then counts how many values fall in each class and reports on the graphs columns with height proportional to the frequency in each class.

The graphs for histograms show the information on each class when the mouse pointer is moved onto a class, including the class' boundaries and middle value, actual minimum and maximum value in the class, etc. Moreover, the user can ask for the information to be printed in the log window.

Histograms may be computed from a single series using its values across time, or from many series using one single value for each of them. The option **Time series** or **Cross section** determines which type of histograms are computed. When **Time series** is selected, there must be one single series selected, otherwise the system will issue an error message. A maximum of 100 classes can be specified by the user.

5.9.10   Graph type Lattice

Lattices are graphs formed by grids of with each cell colored according to a value specified by a non negative integer. Lattices require to insert as selected series as many series as the number of cells. The system assumes that the series are contained by lines: first all the values for the first line, then the values for the second line, etc.

The user must provide the data for the time step to be used from the series selected, and the number of columns for the graph. This number must be an exact divisor of the number of series selected, since the division of this two numbers provides the number of lines. The user must also
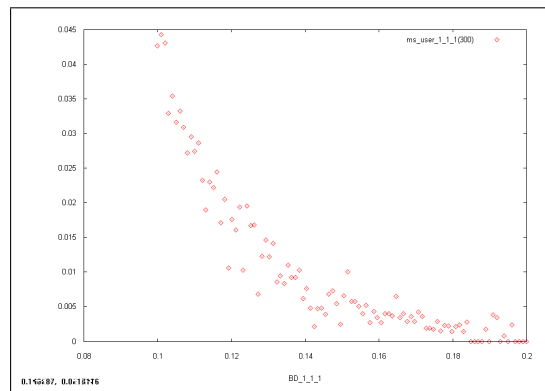
Figure 5.20: Example of a cross section 2D scatter plot using points, represented in a gnuplot window. The series selected contained 200 series. The first 100 provided the values for the horizontal axis. The label reported on this axis is the label for the first series of this block. The second half of the series provided the value (at time step 300) for the dependent variable, again, the label is reported from the very first series in this block.
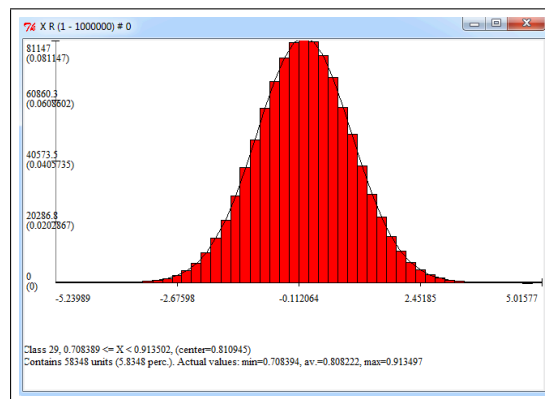
Figure 5.21: Example of an histograms. The data are produced by a generating 1,000,000 random draws from a standardized normally distributed function. The histograms computes the frequency for 50 classes and interpolates a normal function computed with the same mean and variance of the data.

specify the number of pixels to be used for total width and height of the lattice window, which obviously must be larger than the number of lines and columns. See figure 5.22 for an example.

5.9.11 Statistics

The system can compute descriptive statistics from the series selected. The statistics concern average, minimum, maximum, variance and standard deviations of the values indicated. The computation can be performed across time or across series, as indicated by the options **Time series** or **Cross section** respectively. The results will be shown in the log window.

5.9.12 Exporting data

Clicking on the **Save data** button the system exports the values from the series selected in a text file. The data can be exported as LSD result file or as plain text files. In the first case the data can be uploaded by any LSD model program using the appropriate command in the Analysis of Result module (**Add Series**).

Choosing to export data as text file it is possible to choose a variety of options, such as the delimiter symbol and names of the variables.
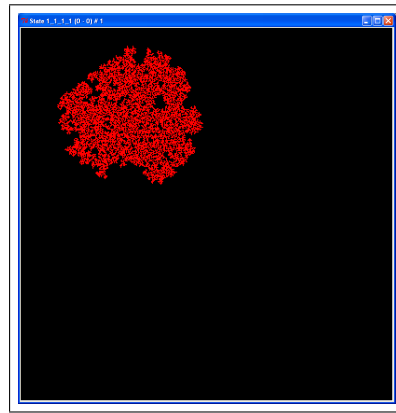
Figure 5.22: Example of a lattice. The series selected are taken from a 400 x 400 matrix, represented in LSD as 400 objects **Row** containing each 400 objects **Col**. The series concern an element located in **Col**, taking values 0 or 1.

5.9.13 Exporting graphs

Clicking on the button **Postscript** the user can save a graph window as postscript file, for inclusion in a document.

5.9.14 Adding further series

The values that the user can process in the AoR module are those produced during the last simulation run by elements marked as being saved for this purpose (see paragraph 5.1.2, pag. 78). However, the module allows also to add more series in the set of the available series. Additional series can be obtained pressing button **Add series** in between the two listboxes.
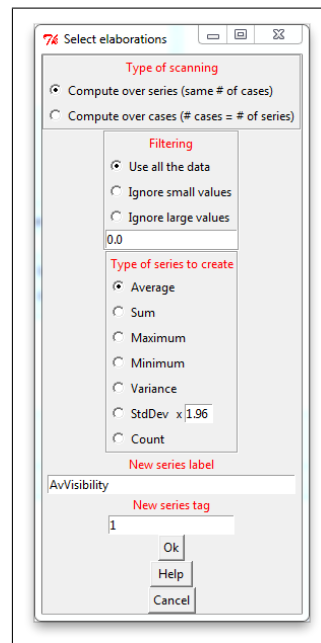
Figure 5.23: A single new series can be created as a statistics computed on the data contained in the series selected. The new series can be computed by computing the statistics for each time step across all the series, or for each series across all its time steps.

The new series can be obtained from several sources.

Current state of the model. It is possible to insert the data from an element of the model not saved during a simulation run. In this case, the number of new series will equal to the copies of the objects containing the element. Each of the new series will contain a single value, which will be assigned the conventional time step of $t = 0$. This series can be used as independent variable in cross-section scatter plots even in case the user specifies different time steps for the dependent variables.

Previously saved result files. User can add new series stored in files previously generated during past simulations. Users can select single files or whole batches of them, as those, for example, generated during batteries of simulations. In this case, the system will add all the new series as contained in the result files, assigning them the original names extended with the letter **F** (so that **Label** will appear as **LabelF**). Moreover, these series will have a further digit attached to the tag, indicating a progressive index for any file. The log window will report the code for each file so as to associate series to specific simulation runs.

Moving averages from selected series. If one or more series are reported in the **Series Selected** listbox it is possible to create as many new series computed as moving average of the selected series smoothed over the indicated number of time steps.

New series from elaboration of selected series. The user can elaborate the values from the selected series generating a single new series. The options available are listed as in figure 5.23.

The options available are the following. By default the system considers the creation of a new series with the same number of time steps as the selected series, and considering the elaboration across the selected series. Alternatively, the user can choose to elaborate the data across time steps to generate a new series having as many virtual time steps as the number of series.

The second option allows to remove from the elaboration outliers, i.e. data too small or too large, where the user imposes the treshold.

The type of elaboration includes: average; sum; maximum; minimum; variance; standard deviation (times a constant); counting.

Finally, the user can specify the new variable's label and a tag number.

5.10   Module L^{S}D Debugger and Data Browser

The L^{S}D browser shows the content of the model by means of their structure, but does not allow to inspect the copy of each element. The L^{S}D debugger and the data browser use similar windows to provide access to each individual element of the model, object by object. The data browser is used when a simulation is not running, activated by menu **Data/Data Browse** of the L^{S}D browser. The L^{S}D debugger is instead used during a simulation run, when it is interrupted. The L^{S}D debugger includes, besides the data browser, also the commands to control the simulation run. In the following we will refer for rbrevity only to the L^{S}D debugger, though all the commands work also for the data browser.

The L^{S}D debugger shows the state of one copy of an object. Starting from the button, the debugger window includes:

- The list of elements contained in the object, including their current value.

- The position of the copy shown in the debugger within the object structure of the model.

- A set of buttons to move the browser through the object structure.

- A set of buttons to control the simulation run (not present in the data browser).

- The label of the equation just executed, which caused the debugger to be activated (not present in the data browser).

The L^{S}D debugger is activated during a simulation run under a variety of conditions:

- The equation for an element marked to be debugged is just computed and the simulation is running in debug mode.

- The value for an element just computed meets a condition previiuously specified by the user.
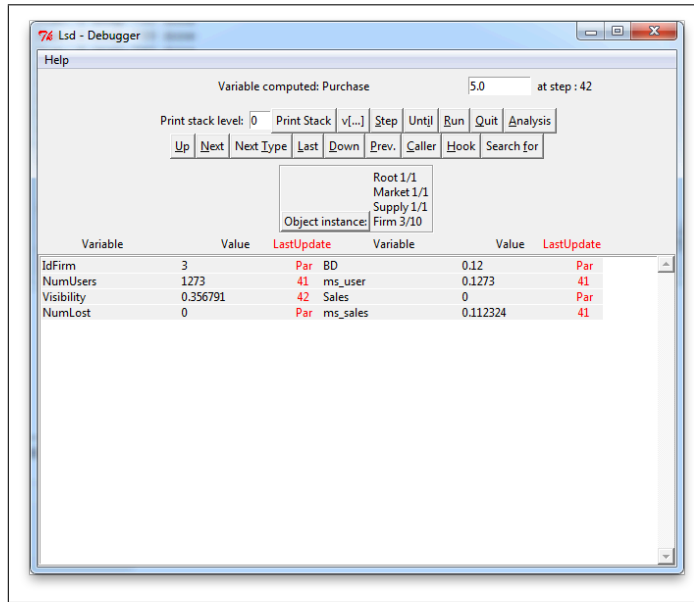
Figure 5.24: Debugger set on the equation for **Purchase** at the 42nd time step. The debugger window, as the data browses, show the state of a model for each individual copy of an object. The user can observe the state of the elements within the objects, moving through the object structure, modify the values, activate equations, analyse the results, and continue the simulation run.

- The equation's code executed encountered the command `INTERACT(...)`.

- The execution of an equation caused an unrecoverable error, aborting the simulation.

In all the cases the debugger contains the state of the model during a simulation run at a specific point of the simulation. This point is indicated by the computation for a variable (or a function) and the copy of the object containing the element. Users can perform several actions with the debugger window.

5.10.1 Inspecting and changing elements' states

The list of the elements in the debugger shows all the elements contained, their values and their nature. For variables, the window shows the time step at which the variable was lastly updated, that is, executed its equation. Note that in the top right corner of the window it is reported the current time step, so that the user can assess whether a variable has been already updated or still waits for its equation to be executed within the current time.

Every element can be modified, that is, its values changed. Double-clicking on its label, a new window appears, showing its value(s) used in the model, and offering a variety of options.

This option window shows the value of the element, which can be modified by the user, including, if existing, the lagged values. Along the cells to enter the new value, the button **Set all** activates the initialization function for the same type of elements of the model, so that the user can change, at a single stroke, a whole set of elements.

Below the cell(s) for the value(s) of the elements, it is possible to decide the debug option for the element (obviously, not present for parameters). Marking this option will cause the simulation to be interrupted as soon as this specific element is computed. The option to debug or not the element can be applied to every copy of the same element in the model marking also the second option.

The rest of the buttons perform the following operations:

- **Done**. Concludes the option setting for the element and return to the L^SD debugger;

- **Equation**. Shows in a new text window the equations' code for the element (not present for parameters).
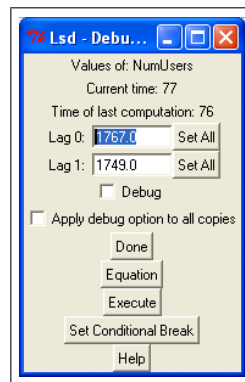
Figure 5.25: Double-clicking on one element in the list shows the value contained in the element and various options to edit its state and properties, both for that single copy or for all the copies of the same element in the model.

- **Execute**. Computes the code for the equation of the element, if the computation is compatible with the state of the element. That is, always for functions, and only if the variable's last update time is more recent than the current time.

- **Set conditional break**. It is possible to define a comparison value and one of three conditions to determine whether the debugger should be activated again. The condition will be computed every time the element is updated, and, if met, will activate the debugger in any case.

5.10.2 Inspecting and changing objects

The central square in the window shows the position of the object in the model structure, and the ordinal number of its copy within the group of the same type of objects descending from the same copy. Similarly, the same information is reported for all the objects at higher hierarchical level, up to the **Root** of the model.

It is possible to click on the **Object instance:** label of the window to access the automatic setting of the number of objects. This window (shown in figure 5.9 pag. 90) allows to increase or decrease the number of objects within the model.

5.10.3 Moving the browser through the objec structure

The lower row of buttons allows to move the debugger window through the objects.

- **Up**, or the up arrow. Move to show the object containing the current copy.

- **Next** or the right arrow. Move to show the following copy (on the right) of the current object.

- **Next type**, or the key **t**. Move to show the first copy of a different type pf object following the current object.

- **Last**, or the key **l**. Show the last copy in the group of the object of the same type as the current object.

- **Down**, or down key. Show the first object contained in the current object.

- **Prev.**, or left arrow. Show the previous object contained in the same parent as the current object.

- **Caller**, or key **c**. If pressed as soon as the debugger was activated, show the object containing the element whose equation caused the present equation to be computed. Otherwise, and in case the equation was computed because of the L^{S}D simulation manager normal activity, does nothing.

- **Hook**, or key **h**. Move to the object linked through the *hook* (a link set by the modeller) to the current object.

- **Search for**, or key **f**. Show the object containing a specific element with a specific value.

5.10.4   Simulation run' controls

The first row of buttons is available only for the debugger window, not for the data browser. These commands provide information about the state of the simulation and allow to set the options to continue the simulation run.

- **Print stack level** X. When the simulation re-start, the log window will show the time spent and various information about the execution of equations computed below the stack level specified. This is the same option available from menu **Run/Sim. settings** in the L^SD browser.

- **Print stack**. Prints in the log window the current content of the stack. That is, which equations initiated their computation and were interrupted because other equations needed to be computed firstly.

- **v[...]**. Shows the list of the temporary C++ variable used to store intermediate results during the execution of the equation that caused the debugger to be activated.

- **Step**, or key **s**. Continue the simulation in debug model, in order to interrupt the simulation as soon as another element marked as to be debugged has its equation computed.

- **Until**, or key **i**. Set the next time step at which the simulation must enter the debug mode.

- **Run**, or key **r**. Exit the debug mode and continue the simulation.

- **Analysis**, or key **A**. Exit the debugger and activate the analysis of results module. The module will be given all the data produced by the simulation until the previous time step. However, if new data, besides the series explicitly saved, are requested in the module, they will concern those contained in the current state of model, without controls on which time step they refer to.

5.10.5   Debugger header

The top row of the debugger shows the element just computed (which caused the debugger to be activated), its most recent value, which can be modified, and the current time step of the simulation.

If the debugger is activated by the modeller's command `INTERACT(...)` inserted within an equation, the header will contain the message and the value specified by the modeller. The return value of the command will be those inserted by the user.

5.11   Log window features

This window shows the messages generated by the system, as warnings, errors, or statistics. Moreover, this window prints the step completed for a simulation when the users do not use the run time plotting.

The window is formed by a text editor, and a row of buttons. The editor can be used to select and copy the text contained. The buttons serve exclusively at simulation time, while a simulation is executed.

5.11.1   Button **Stop**

Quit the simulation, keeping the data up to the last time step available.

Figure 5.26: Log window, used to communicate messages from the system to the user and to issue commands while a simulation is running.

5.11.2   Button Fast

Iconify the run time plot window and interrupts the printing of the steps completed. While running in this state the simulation is sensible faster.

5.11.3   Button Observe

Return the LSD model program to execute the simulation as before pressing button **Fast**. The simulation is slower, but the run time plot is visible, or the time steps plotted in the log window.

5.11.4   Button Debug

Activates the debugging mode. In this mode the simulation is interrupted at the first occasion an element marked as being debugged completes the equation computing its new value. See instructions on the module **LSD debugger**.

5.11.5   Button Help

Show the manual page for the log window.

5.11.6   Button Copy

Copy in the clipboard the text selected in the **Log** window, useful when, e.g., statistics are written from the Analysis of Results module.

5.12   Model structure window features

The model structure window contains a graphical representation of the tree of objects for the model. The object ***Root*** on the top is not shown. Any other object contains the number of instances for that object in the configuration, possibly divided in groups of digits in case the parent of the object is present with many copies. The window shows only the number of the first 5 groups.

Moving the mouse over the symbol for one type of objects makes appear a window detailing the elements contained in the objects. The window disappear when the mouse leaves the symbol for the object.

Double-clicking on the symbol of an object moves the LSD browser to show the object indicated.

Right-clicking on the symbol of an object, or keeping the key **Ctrl** pressed and left-clicking, the LSD browser activates the window to initialize the content of that type of object.
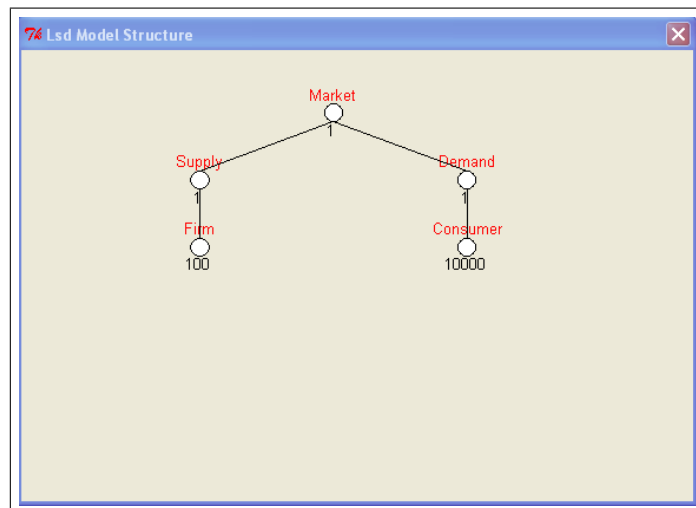
Figure 5.27: Graphical representation of the model structure. This window provides an overview of the object structure of the model and gives access to the most frequent commands, like moving the browser to show an object, list its content, or opening the module to initialize its elements.

Chapter 6

LSD modelling language

6.1 Introduction

Simulation models can be thought of as computer programs sharing some common features, such as defining a simulation clock, or saving and plotting simulation results. LSD provides the opportunity to exploit the computational power of C++, a basic programming language, to represent the computational content of a simulation model, but generating automatically all the features common among all simulation models. Therefore, any type of model can be implemented in LSD, and the implementation is extremely fast and simplified, since it requires no further programming knowledge than that required to express the model's own computational content.

A LSD model is composed by four types of elements: objects, variables, parameters and functions. Objects must be thought as the simulated counterparts of real-world entities, and in the model act as containers of the any types of element.

A simulation run consists in executing a sequence of time steps, during which the system "scans" all the objects present in the model at the time and "updates" the variables there contained. The updating consists in executing the "equation" for the variable, that is, a chunk of programming language code that returns a numerical value.

The modeller writes the model by defining the model structure and the equations. The structure of a model consists of the lists of objects, variables, parameters and functions, that are inserted in the model by means of simple and intuitive graphical interfaces, requiring basically only to type the element's label. The equations are written as text in a file that is then compiled along with the rest of the LSD source code. The code of an equation consists of a header (declaring which variable or function it refers to), a body, in which any computational command can be inserted, and a numerical results, to be associated to the variable or function as equation's result.

The results of the simulation depend on the interaction between the equations of the model and the model structure. As any program, simulation models also have generally many different possible implementations generating the same results. Therefore, one may choose among at least a few different approaches to implement the model. In the following we present the description of the rules governing the definition of a model structure, and of the language available to write the equations of the model. It is worth to notice that building a model entails working in parallel on the model structure and on the equations, writing, testing, and editing them by small incremental steps. Therefore, there is no rigid priority on whether to start defining the configuration or from the equations.

Ideally, a modeller should start by having on paper a set of equations expressed as discrete difference equations, and trying to implement them while generating the structure within which the elements required by the equations are inserted. Any new equation should be tested by testing its behaviour, and then adding further elements. Modifications of either the structure or the equations can be easily introduced. Therefore, for example, a model containing several variables, each computed as elaboration of others, can still be tested by initially defining one single equation for a variable, and defining as parameters the other elements, so that the model can run and undergo testing. Once the first equation is reliably tested, one of the model parameters can be turned into a variable, its equation added, and a new round of testing can be performed. Following

this cautious approach, one is guaranteed to quickly develop a model without wasting time in hunting tens of errors, or missing crucial implicit aspects of the model.

6.1.1 Model structure

The structure of a model is composed by a hierarchy of objects, where each object is contained within another one, and, in turns, contains variables, parameters and functions, besides, possibly, other objects. The object **Root** is necessarily the top-most object of any model, is the only object that cannot be multiplied in many copies, and its label cannot be changed. Therefore, the first steps in building a model structure consists in defining one or more objects as contained into **Root**.

Given this constraint, any user-defined object has necessarily one of three possible relations with the other objects of the model. Considering, for example, a model composed by only two objects labelled **A** and **B**, we may have that **A** contains **B**; **B** contains **A**; or they have not relation. The issue is to determine therefore whether two objects should be placed in the structure in one of the above relations.

However uneasy one may feel at the beginning when designing the model structure, a little time working on it will quickly remove obvious mistakes, and will speedily produce a sensible structure. In the following we provide a few rules of thumbs to apply when in doubt about the relation between any two objects.

The relation of containment among objects represents, literally, that the contained object is part of the "parent", or container, object. Let's use the convention that **Root** is the top-most object, and that contained objects are at a "lower" a lower level than containers. High level objects should be thought as aggregate entities, containing variables that have a relevance over the whole set of elements contained in the lower level objects. For example, an aggregate object may contain as a variable the average of variables stored into lower level objects. Or the high level object may contain a parameter used by all the variables in the lower level objects.

The major role of the structure is evident when a model is configured with many copies of object types. In fact, the structure of the model, how the objects are organized in the hierarchy, determines how the equations in the model access the elements stored in other objects. Given the hierarchical structure, all the copies in lower level objects are contained within a single copy of a high level object type. Suppose, for example, that a model is defined as having two objects, **Market**'s and **Firm**. Defining **Firm** as contained in **Market** implies to assume that any given firm will be assumed to stay within the same copy of **Market** during a simulation run. Alternatively, placing **Market** as contained in **Firm** implies to assume that every firm will act upon several, independent markets. The independence is due to the fact that each firm will have its own set of markets, which are not related to other firms. A third option is to consider the position of **Market** and **Firm** as "parallel" in the hierarchy, but contained within a third object, as **Root** or another, high level object. In this third case, firms and markets will not have a structurally determine relations, and the model will be able to relate any firm to any market, and vicevera, though the modeller will have to write equations determining the relations any time they may occur.

The relation among objects are relevant because of the way L^{S}D computes the equations, in particular, how the system provides, at run time, the data from the model necessary for an equation to be computed. L^{S}D offers a simple default system, and many possible ways to work around the default choices. The test of a structure, whether it is a "correct" one or not, consists in checking whether the equations computing the elements in the objects can be easily written using the default L^{S}D system, or require an extensive use of manually instructing the code. In this latter case, one may be certain that a different structure is more likely to allow for an easier implementation of the code.

6.1.2 L^{S}D equations

Modellers express their equations as chunks of C++ and L^{S}D code producing a numerical value[1] which is attached, during a simulation run, to one copy of the variable which the equation refers to. Modellers can control the steps executed during a simulation run exclusively by writing code for one element of the model.

---

[1]L^{S}D limits to consider only real valued numbers (double precision floating point).

L^{S}D is inspired to the representation of a model as normally used in discrete difference equation models. Therefore, equations must be conceived as pieces of code to be executed at the generic time step t for the generic copy of an element (say, a variable) that can be present in the model with several copies. Therefore, the theoretical representation of a model's code can be directly transcribed in L^{S}D from a set of equations as:

$X = f(X_{-l}, Y, Z, ...)$

Notice that L^{S}D equations do not need redundant information, which, being redundant, risks generating inconsistencies. In particular, the L^{S}D equations do not need to use a time index t, since it is implicitly assumed that the value produced will refer to the present time step of the simulation. If the arguments of the equation need to be evaluated at a past time, then the modeller will indicate only the number of lags.

Also, the equations' code does not distinguish between types of elements for the arguments. They may be parameters, variables or functions, but, in the equations' code they are always referred to by their label. These feature is very handy since one can turn an element from, say, a parameter to a variable, or viceversa, without affecting the code for the equations using the element.

More relevantly, the modeller needs not to provide (necessarily) an identification, by means of an index or other data structures. For example, suppose that the model includes several copies of entities representing markets, and that each market is formed by a group of firms. Standard representation of the equations for variables contained in markets and firms would use a combination of indexes to refer to the elements contained in the different entities. For example, elements referring to the markets will be indexed with i, and elements in firms with a double index (i, j), for the market and firm.

L^{S}D equations ignore these indexes, referring only to the labels of the elements without making explicit reference to the position of these elements in the model. For example, consider that the quantity of a firm is a function of the price. The L^{S}D equation will be the equivalent of the following expression: $Q = f(p)$, independently from the price being an element part of firms or of markets. L^{S}D system will use in the equation the most "sensible" price: if there is such an element associated to the firm, then it will be used. Otherwise, the system will search for the copy of the market containing the firm, and, if found, will use that copy.

These properties of L^{S}D rely on the system to retrieve the appropriate elements for the computations of an equation. Such run-time solution of potential indeterminacies is extremely useful to easily edit a model and re-using parts of it. In fact, we may change the behaviour of the model by maintaining intact the equations' computational content, but modifying only the positions of the elements in the model structure. Also, we may extend the model without requiring the modification of the existing code. Consider, for example, the case of a model that, after having defined markets and firms, decide to introduce countries as new entities, each containing many markets. The indexing system referring to the elements of markets and firms would need to be changed, in order to include also the information on the country each market and firm is contained into. L^{S}D would operate this modification automatically without requiring any modification to the existing code.

The automatic system used to provide each equation with the "correct" elements necessary for the computations will be described in the following of this section. It needs to be noted, however, that this system can be overruled by the modellers to express different choices, for example, when an equation needs to change, according to some rule, some specific elements to use. For example, suppose that a firm has a rule to shop around different markets to spot the currently highest price, and that it will then compute accordingly the quantity produced. Such equation cannot rely on the automatic L^{S}D system to find the "correct" copy of the price, since the rule needs to be explicitly defined. The language available for express L^{S}D equations allows to easily cope with such cases.

6.2 Computable elements: variables and functions

L^{S}D admits two types of elements that can execute computational operations, and consequently (potentially) change values. It is only the execution of the equations associated to these two classes of elements that the modeller can control. The two classes are variables and functions. Variables are elements of the model that need to be computed once and only once at each time step. That is,

at each time step a variable will assume a new value which is the result produced by the equation associated to the label of the variable. The equation associated to a variable will then be computed once and only once at each time step.

Functions in L^{S}D models are elements similar to variables, in that their value is computed as the result of an equation, but the timing of updating for functions is different. Functions are not updated automatically at each time step, but only if other elements of the model (that is, other equations in the model) require their values. If, in the same time step, several equations require the value of a function, then the equation for that function is re-computed at every requests. Conversely, if the value of a function is never requested during a time step, then its equation is not computed at all.

As example of a variable, consider a model where variable **Price** is determined at market level, and it is meant to be the same variable (and the same value) at each time step for all the firms in the model. Each of many firms uses **Price** (say to determine the profits), so that each time step the value of **Price** is requested many times by all the equations for **Profits** in the model. 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 desire to create one single variable computing the probability to innovate for any firm (which may depend on both the firms own internal characteristics, plus other elements, like the overall level of scientific development) 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.

Note that for both variable and functions it is possible (and, generally, it is the case) that the model includes several copies of the same type of element. For example, we may have a variable **Price** contained in object **Firm**, and the model includes several copies of this type of object. Each copy of the variable **Price** will be computed once and only once at every time step, and therefore, the code for its equation will be actually computed, in a time step, as many times as many copies of **Firm** are defined in the model. The differences in the results of each execution of the equation for **Price** will be due to the *environment* in which the equations code is executed, that is, the copy of the object in which **Price** is contained.

6.3   L^{S}D Simulation Manager

At the start of a simulation run the model is composed by the configuration and the list of equations, compiled as a list of pieces of code associated to every variable and functions of the model. The L^{S}D module dealing with generating an actual simulation run is called L^{S}D Simulation Manager, LSM. The task of the LSM is to ensure that the sparse information entered by the modeller is arranged in such a way to produce the intended results, that is, generating a stream of values for each of the elements contained in the model at each time step. The values of the elements in the model are generated executing the equations associated to them, and using their results as the up-to-date values. For a simulation to work properly, it is necessary to guarantee two properties: firstly, all the equations for the variables of the model must be executed once, and only once; secondly, if two variables have equations that must be executed in a specific order, this order must be respected. The LSM ensures that both properties are respected. In this paragraph we provide in some detail how the LSM guarantees the proper execution of a simulation step, updating all the variables with the correct order of execution of their equations. However, this knowledge is not necessary for L^{S}D modellers, therefore uninterested readers may skip the rest of this paragraph.

It is possible to consider the task of the LSM as divided in two sub-tasks. The first sub-task consists in scanning all the objects of the model, with an efficient and exhaustive strategy. By efficiency we mean that the strategy needs to pass through each object once and only once, and exhaustive imposes that every object is scanned. There are many possible such strategies. The one used by LSM consists in starting from the top-most object (**Root**) then moving to its descendants and, for each of them, replicating the same routine, scanning the descendants. When an object

without descendant is encountered, then the routine returns "up" to the object containing the object just scanned, and moves to scan its next descendant.

The second sub-tasks consists in updating the variables of an object. When an object is reached by the scanning routine, the LSM operates the updating of the variables stored there. The updating routine for a variable terminates when the equation for the variable is completed, and the result is stored as the value for the variable at the current time step. However, in between the beginning of the updating of a variable, and the assignment of the result, it is possible that other operations are performed. The LSM is also used, during the execution of the equation, to provide the computation with the necessary values, which may require the execution of other equations, even in different objects. For example, suppose that the scanning routine has reached an object and a variable X undergoes its updating. Suppose also that the equation for X requires the value of another variable Y at the current time step (i.e. no lags). The LSM searches the model for the copy of Y required for the computation, and controls whether the variable has been already updated or not. In this second case, the LSM interrupts the computation of X, updates Y, and uses its value to complete the equation for X. When the LSM will scan the object containing Y, the updating routine will recognise that the variable has been already updated at the current time step, and will not execute its equation again. Therefore, the updating actually triggers the computation of an equation only when it encounters a variable that has not been already updated because of other reasons.

For example, consider a model where a high level object, for example **Market**, contains a variable **TotQ**, computed as the sum of all variables Q present in objects **Firm**, contained in **Market**. The LSM will encounter firstly **TotQ**, because it is stored in a higher object than those containing the Q's. Therefore, its updating will begin before the "natural" updating of any Q. However, the equation for **TotQ** cannot be completed until all the Q execute their equations, providing their up-to-date values. Therefore, the execution of the equation for Q will be triggered and completed during the updating of **TotQ**. When the scanning routine will pass in the objects **Firm**, it will recognise that the variables Q are already updated, and will skip the (re-)execution of their equation.

Notice that the strategy used by the LSM to scan the object is irrelevant, as far as every object in the model is scanned and their variables updated. In fact, the variables of the model may be either independent from one another, or they may need to follow a local precedence order, that is, a few ones need to be updated before others. LSM ensures anyway that the compulsory precedences orders are respected, while independent variables may be computed in any order, without affecting the model results. The strategy for scanning objects described above ensures that when an object is "under computation", that is, its variables updated, all the higher order objects have been already scanned and their variables updated.

6.4   Environment for LSD equations

When writing the equation for a variable (or a function), the modeller must assume that the code will be executed by a generic copy of the variable at a generic time step. One of the major features of LSD is that a modeller can express the computational content of the equation in a generic way, and then leaving to LSD the job of automatically "customize" the actual computation to each specific copy of the variable.

The commands contained in an equation are expressed by the modeller assuming a default system: if not other specified, the most "obvious" result is produced. Such system allows to express the code for the equations in a very compact and intuitive format, though leaving the possibility to express any type of computation.

6.4.1   Managing time lags

Any operation concerning the values of the model, as, most typically, the requests to use the value of an element, is assumed by default to concern the value at the most recent time step available. Using the conventional mathematical notation, if the equation for variable X requires the value of variable Y, then, by default, the computation executed will be $X_t = f(Y_t)$, if the modeller does

not specify otherwise. Obviously, the same notation applies to parameters, which do not have time tags attached. When the equation is instead required to use past (or lagged) values, then the modeller needs to specify the number of lags.

Notice that in both cases, the modeller does not need to worry for the availability of the required elements at the required updating stage. The system automatically ensures that the correct values will be used as indicated in the equations. For example, if a required value is not available as yet (say, that variable Y had not been updated at the current time step when requested for the computation of X), then the system automatically updates the necessary values before completing the computations requiring those values (that is, the equation for Y is executed before the equation for X). Variables that are updated because of the need of their values by other equations are not computed again in the same time step. Instead, functions have their equation re-computed at every (and only) request.

6.4.2 Managing multiple copies

When an equation requires the values of other elements in the model, and these other elements are present in many copies, by default the system provides the "closest" element, where the "distance" is computed according to the hierarchical links among objects (obviously, unless otherwise specified). Suppose, for example, that the equation for variable X requires the value of an element Y. Suppose also that there are many copies of X, that is, many copies of the objects containing variable X, so that the equation for X will need to be executed, in the same time step, as many time as many copies of X needs to be updated. For each execution, the system is requested to retrive the value of an element labelled Y. Depending on the structure of the objects (that is, the relation between the object containing Y and X), there will be different results, that is, the system will provide the equations for each X with values from different copies of Y.

The default system, retrieving automatically the data required within the equations, works as follows. If Y is an element stored into the same object containing X, then the copy used by every X will be the one stored within the same object. Therefore, each execution of the equation for the different copies of X will each use a different copy of Y, which has "distance" 0 from the object containing X.

If the system cannot find Y within the same type of object containing X, then it starts looking at the descending objects, that is, those contained with the object storing X and, if necessary, those at still lower levels. The first copy of Y found in this search is returned, and its value used in the equation. If even this search does not find Y, then the search continues on the higher level objects, those in the hierarchy above that storing X.

At any stage, the system replicates the same search strategy: search among an object, then its descending objects, and eventually among the higher levels ones, obviously without returning in those already explored. This strategy ensures an exhaustive search over the whole structure, so that if the searched element exists in an object of the model, the strategy is able to find it. Moreover, if there are many copies of the searched element, the copy "closest" to the object storing the variable under computation is provided.

6.5 C++ basic grammar for L^{S}D coding

The code for the equations must be legal C++ code, extended with the L^{S}D functions described below. Before moving to describe the L^{S}D specific commands to be used in L^{S}D equations it is necessary to have a minimum knowledge of the most frequently used C++ commands.

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 (unless otherwise instructed by the code itself).

Any line of code must respect the C++ rule of terminating with a semi-colon ";", unless the line is a multi-column L^{S}D command like `EQUATION`, or C++ command, like `if(condition)`.

6.5.1 Comments

The code can include comments, that is text that is ignored by the compiler and serves only to the readers of the code to facilitate the interpretation. 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
```

6.5.2 Assignments, arithmetic operations and increments

If `a` is a C++ 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 contains 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 also 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;
```

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 be, and then increments `a`), and `c` equals 5, because `a` increased from 4 to 5 before being assigned to `c`.

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.

6.5.3 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 lines 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 "not", 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 at least one cond1 or cond2 is true, otherwise is false.

For example, the following code distinguish 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. Therefore, the condition `if(1)` is always true, while `if(!1)` is always false.

6.5.4   Use of cycle `for`

A very frequently used command in all programming languages allows to repeat a block of lines again and again until a specific condition is satisfied. Note that in LSD equations you will not need to use explicitly this command, since you have available the `CYCLE(...)` command, executing a repetition for each descending object.

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 variable i 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
```

6.6   System variables available for equations' writing

Writing the equations of a model it is frequently necessary to make use C++ variables to store intermediate results or, in general, data required to perform specific computations. The environment where the equations are executed provides a set of system variables[2] that facilitates the most frequent operations. Modellers can also use a set of system variables containing data from the current state of the model, which may be useful to customize the computation of the equation.

---

[2]In order to avoid confusion, we will refer to a *system variable* meaning a C++ data structure, to be used within the code for an equation. When, instead, we refer simply to a *variable* we mean a variable of the model.

Finally, since the equations are basically C++ code, new system variables can be generated and used by the modeller, though this is rarely necessary. For instructions and example on user defined system variables see below.

We can distinguish the system variables in three different groups: system variables locally available for use within the equation's code; system variables specific to the equation computed; global system variables concerning states of the whole model.

6.6.1 System variables locally available within an equation

The most frequently used system variables are those used to store intermediate values generated within the equation. These variables are *local* in that their existence (and therefore the values contained) is strictly limited to a single equation. Therefore, it is impossible to pass data from one equation to another by means of a local variable, even though the name of the variable is identical.

The numerical values generated within an equation are stored into the local system variables `v[0]`, `v[1]`, `v[2]`, etc. These system variables are generally lost when an equation is completed. However, during the LSD debugging their values are available, so that the user can inspect also the intermediate steps within an equation.

The objects dealt with within an equation are stored in the local system variables `cur`, `cur1`, `cur2`, `cur3`, etc. These system variables can, for example, be assigned a newly created object, in order to perform specific operations like initializations.

In some cases it is necessary to integer system variables, differing from the values used in the LSD models that are all real-valued numbers. The equations' environment provides the following integer system variables (always local): `i`, `j`, `h`, `k`. Note that assigning values between integer and real-valued system variables requires a casting. For example, a line containing the command `i=v[2];` may generate an error. The correct form is `i=(int)v[2];`. Viceversa, the casting from integer to real-valued system variable is `v[2]=(double)i;`.

6.6.2 System variables specific to the equation under computation

The equation executed is meant to provide a result to a specific copy of a variable, stored into a specific copy of an object. The object containing the element computed is referred to as `p`. Therefore, any operation requested to the `p` object will concern the very copy of the object whose element is updated by the equation.

As we have seen, an equation may be computed either because the LSM updates the variable associated to the equation or because the updated value for the element (variable or function) is requested by another equation. In this second case, the equation being executed may require information concerning the entity whose equation triggered its computation. The environment for the equations provides the object `c` (for *caller*) whose element caused the equation to be computed, that is, containing the variable or function whose equation is currently interrupted waiting for the computing equation to be completed.

Notice that when the LSM requests a variable to be updated, and therefore there is no object to be used, the system variable `c` still exist, taking the conventional value of `NULL`.

6.6.3 Global system variables

A third class of system variables contains information concerning the overall state of the model.

- System variable `root` contains the **Root** object of the model.

- System variable `t` contains the time step of the simulation. This is an integer variable;

- System variable `max_step` contains the total number of steps defined for the simulation run. This is an integer variable;

- System variable `sim_num` contains the current simulation run executed, starting from 0. This is an integer variable;

- System variable `seed` contains seed used to initialize the random number generator at the start of the simulation run. When more than one run is executed, the `seed` is increased at every new run. This is an integer variable;

- System variable `quit` is set to 0 while the simulation is running. In order to stop the simulation, before the completion of all the steps initially defined, it is possible to set `quit=1`. In this case, the simulation will complete the updating of all the variables at the current time step and then will terminate normally. This system variable is set to 2 when an error occurs, and the simulation tries to stop without computing the equations. This is an integer variable;

- System variable `msg[]` is a character string, used to generate text messages within the equations.

6.7   LSD objects' links: `->up`, `->next` and `->son`

This paragraph describes some of techniques used to optimize the speed of execution of models. Non expert users would better skip this paragraph paragraph.

Any object in the model is linked to other objects in order to generate the model structure. Therefore, it is possible to use an object available in the equations' environment and access its "neighbours" (and potentially any object of the model). This operation is never necessary, since there are LSD commands for the equations that provide the same result, that is, return object with given characteristics. However, in some cases, it is more efficient to use directly the intended object rather than relying on the LSD commands, which, being expressed in general forms, cannot exploit specific properties of a model.

Given an object, like `p`, `c` or `cur`, the immediate neighbours of the objects are the following (call `obj` the starting object).

- `obj->up`: is 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, that is, having the same "parent" object `obj->up`. Notice that `obj->next` may be `NULL` is `obj` is the last object of the group; `obj->next` may be a copy of the same type of object as `obj`, or they may be different. The command `go_brother(obj)` returns `NULL` in either case when `obj->next` is NULL or is a different type of object.

- `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.

6.8   LSD commands for equations

This section describes all the LSD commands available for coding the equations, presenting also examples for the most frequently used expressions. The same information contained in this section is reported in the help menu in LMM. Any changes to the LSD functions or command is most readily updated in the help pages of the distributed LSD version.

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, function or object, to be inserted in a LSD function;

- **Label**: refer to the model name of a variable, function, 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;

The equations must be placed in a text file to be compiled with the rest of the LSD source code in order to generate a LSD model program. The name of the file must be consistent with the compilation option. Using **LMM** the user can simply ask the editor to show the equation file for the model.

The equations can be placed in the equation file in any order, provided they are located between the lines `MODELBEGIN` and `MODELEND`.

6.8.1 `EQUATION("Label")` ... `RESULT(value)`

The equation for a variable or a function is written in the equation file and is indicated by the keywords:

```
EQUATION("Label")
/*
Place here a comment to be
used for the automatic documentation
of the model
*/

//any legal code here
RESULT(value)
```

The code above is the framework for the variable **Label**. Any code after the header `EQUATION(...)` will be executed every time the variable or the function is computed. The final result must a real value to be inserted in the last line `RESULT(value)`.

6.8.2 `V("X")`

Return the value of X at the present time of the simulation, using the default LSD system to retrieve elements of the model (see the example below). There are three other members of the family:

- `VL("X",lag)` or `p->cal("X",lag)`: return the value of X starting the search from the currently computed objects at time `t-lag`;

- `VS(obj,"X")` or `obj->cal("X",0)`: return the value of X starting the search from the object `obj` at time `t`;

- `VLS(obj,"X", lag)` or `obj->cal("X",lag)`: return the value of X starting the search from to the object `obj` at time `t-lag`;

Example This is obviously the LSD command most frequently used in the equations' code. This command is implemented in such a way to always return the intuitively obvious value, that is, to select the correct copy of the element 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[3].

The command V(...) applied within an equation is actually a command executed on behalf of a specific copy of a variable, which contained within 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(...) 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 elements contained in obj.

2. **Search down**: search among the elements contained in the objects descending from (i.e. contained in) obj.

3. **Search up**: search among the elements of the object containing obj.

The search avoids to return in already explored objects, and therefore terminates either with one copy of the searched element is found (and in this case the required value is returned), or, otherwise, an error message is produced and the simulation aborts.

The error may also be due to the different spelling is between the label of an element in the equation file and in 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 object.

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

6.8.3   V_CHEAT("X", fake_caller)

This is a rarely used LSD command, and only for reasons of efficiency. Return the value of **X** at time t starting the search from the object under computation. The equation for **X**, if executed because of this "call", will use fake_caller as if it were the object that requested its computation. For example, the standard version of V("X") is equivalent to V_CHEAT("X", p)

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:

- VL_CHEAT("X",lag, fake_caller) or p->cal(fake_caller,"X",lag): Return the value of **X** at time t-lag starting to search from 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 starting the search from the object obj1. The equation for **X** will use fake_caller as if it were the object that requested its computation.

---

[3]That is, the object whose copy of **Revenues** is computed.

- `VLS_CHEAT(obj1, "X",lag, fake_caller)` or `obj1->cal(fake_caller,"X",lag)`: Return the value of X at time `t-lag` starting the search from the object `obj1`. The equation for X will use `fake_caller` as if it were the object that requested its computation.

6.8.4 `SUM("X")`

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"))
```

6.8.5 `STAT(X)`

Compute descriptive statistics for X at time `t`, supposedly an element contained in a set of objects descending from the one under computation.

The command 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]=mininimu value

The function can also be used as:

- `STATS(obj, "X")` or `obj->stat("X", v)`: Compute descriptive statistics for X at time `t`, assumed as elements 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])
```

6.8.6 `WHTAVE("X","W")`

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 stored 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 **Firm** containing two elements, **Productivity** and **MarketShare**. A variable contained in **Market**, containing a group objects **Firm** can contain the following equation:

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

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

6.8.7 `MAX("X")`

Return the maximum value of X at time `t` across 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` across 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` across a group of objects descending from `obj`.

Example: suppose to have a group of objects **Firm** 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") )
```

6.8.8 Mathematical and probabilistic commands

LSD provides a list of mathematical and probabilitic functions to be used in the code for equations.

- `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`, that is, e^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`, that is, a^b;
- `RND`: return a value drawn from uniform random function between 0 and 1;
- `UNIFORM(min, max)`: return 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 random value drawn from a poisson random function with mean `mean`;
- `gamma(mean)`: return a random value drawn from a gamma random function with mean `mean`;

Other variables are provided by the C++ mathematical library, added by default to the LSD model programs (e.g. all the trigonometric functions).

6.8.9 `WRITE("X",value)`

Replace the value contained in the element X stored in the same object of the variable under computation, with the value `value`. This command requires X to be located in the object indicated and cannot be used for variables, since this would generate conflicts with the automatic recording of the variables times of updating.

Other members of the same family of commands are:

- `WRITEL("X",value, time)` or `p->write("X",value,time)` : Replace with `value` the value for variable X contained in the same object containing the equation's variable, setting the time of last update for X to `time`, which must be an integer. This format can be used for variables, though, obviously not for the variable under computation.
- `WRITES(obj, "X",value)` or `obj->write("X",value,0)` : Replace with `value` the value for variable X contained in object `obj`. This format cannot be used for variables.
- `WRITELS(obj, "X",value, time)` or `obj->write("X",value,time)` : Replace with `value` the value for variable X contained in object `obj`, setting the time of last update for X to `time`, which must be an integer. This format can be used for variables, though, obviously not for the variable under computation.

For an example, see for function `STAT` in paragraph 6.8.5 pg. 124). As another example, consider the case of creation of a new object **Firm**.

```
EQUATION("Entry")
/*
Generate a new firm
initializing its elements
*/
cur=ADDOBJ("Firm");
v[0]=V("IdGenerator");
WRITES(cur,"IdFirm",v[0]);
WRITELS(cur,"Capital",0, t);
RESULT(1 )
```

Parameters and variables use the different formats to set only the values or also of the time step of (apparent) last computation.

6.8.10 `INCR("X",value)`

Replace the value of X with the previous value of the element increased by `value`, using the copy of X found with a search started from the currently computed object. This command does not modify the indicator of time of for the last computation of X, if this is a variable. The command returns the new value of X after the increment.

Another form of this function is:

- `INCRS(obj,"X",value)` or `obj->increment("X",value)`: Replace the value of X with the previous increased by `value`, using the X found with a search started from the object `obj`. 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] )
```

6.8.11 `MULT("X",value)`

Similar to the command `INCR(...)`, replaces the value of X with the previous value multiplied by `value`. The command returns the new value of X after the multiplication.

Another form of this function is:

- `MULTS(obj,"X",value)` or `obj->multiply("X",value)`: replaces the value of X with the previous value multiplied by `value`, using the X contained in the object found with a search starting from the object `obj`. The command returns the new value of X after the multiplication.

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 depreciation of capital and variable I for investment. Then, the equation for Q can 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] )
```

6.8.12 `SEARCH("X")`

Return the first object with label 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`.

Notice that this command does not search the whole model structure, but only within the descendants of the object indicated.

```
EQUATION("Entry")
/*
Generate a new firm identical to the
first firm existing in the model
*/
cur=SEARCH("X");
ADDOBJ_EX("Firm",cur);//add an object identical to cur
RESULT(1 )
```

6.8.13 SEARCH_CND("X",value)

Return the object containing the element **X** having the value at the current time equal to `value`. The command returns the first copy of the object satisfying the condition found starting a search from the object under computation (with the strategy described in LSD function `V("...")`, paragraph 6.8.2, pg. 122).

Other members of the same family are:

- `SEARCH_CNDL("X",value, lag)` or `p->search_var_cond("X",value,lag)`: return the object containing the element **X** with value `value` at time `t-lag`.
- `SEARCH_CNDS(obj,"X",value)` or `obj->search_var_cond("X",value,0)`: return the object containing the element **X** with value `value` at time `t` starting the search from object `obj`.
- `SEARCH_CNDLS(obj,"X",value, lag)` or `obj->search_var_cond("X",value,lag)`: return the object containing the element **X** with value `value` at time `t-lag` and starting the search from object `obj`.

Example: Suppose to have a model where several firms in different industries offer products at different prices, and a variable needs to provide the price of a randomly chosen firm in a randomly chosen industry.

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] )
```

6.8.14   `RNDDRAW("X","Y")`

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 probabilities to draw each copy of X will be given by by the ratio of the value of its Y divided by the sum over all the Y's. Therefore, the Y cannot be negative.

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, particularly when there are many copies of X. 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 $Y/\texttt{tot}$, 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 $Y/\texttt{tot}$, 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 $Y/\texttt{tot}$, 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 $Y/\texttt{tot}$, with the value of variable or parameter Y computed at time `t-lag`.

A last member of the family choose randomly an object with identical probabilities:

- `RNDDRAWFAIR("X",)` or `p->draw_rnd("X")` : Return a randomly chosen object with label X among a group of them contained in the object under computation, where each object has the same probability.
- `RNDDRAWFAIRS(obj,"X",)` or `obj->draw_rnd("X")` : Return a randomly chosen object with label X among a group of them contained in the object ***obj***, where each object has the same probability.

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:

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

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

6.8.15 CYCLE(obj,"ObjLabel")

This L**S**D expression is a very frequently used form of the `for` cycle (see 6.5.4, pg. 119). 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`, referred to as 'pointers'. The system provides several local pointers: `cur`, `cur1`, `cur2`, etc.

The use of the command is the following:

```
CYCLE(cur, "Obj")
 {/*********
  place here any code. It will be repeated as many times
  as many copies of Obj are found.
  Each iteration will have a new copy of Obj
  assigned to pointer cur
  *********/
 }
```

When the keyword `CYCLE(cur, "X")` the system searches for the first copy of **X** and assigns it to `cur`. Any L**S**D function operating on `cur` is now actually operated on this first

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[4]. The index is computed as 1 divided by the sum of the squate of market shares:

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

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

```
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 of squares of ms
 }
RESULT( 1/v[0] )
```

The command `CYCLE(...)` cannot be used when the objects scanned may be deleted. In this case it can be used the form: `CYCLE_SAFE(obj, "ObjLabel")`, for object contained in the currently computed object, or `CYCLE_SAFES(objFrom, obj, "ObjLabel")`, where the cycling objects `obj` are contained in the `objFrom` object.

---

[4]The index is a value from 1 to n, the number of firms. The lower the value the higher the concentration.

6.8.16 `SORT("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 for decreasing market shares
cur=SEARCH("Firm"); //take the first in the group
v[0]=VS(cur, "Price"); //this is the price of the highest market share firm
...
```

6.8.17 `ADDOBJ("X")`

There are several commands that create objects in a model. The commands differ depending on the values stored in the newly created objects, and on the number of objects to create. All the formats of these commands require that the equation specifies the location for the newly created objects, which must be an object already containing at least one copy of the object type added to the model.

  The simplest form is `ADDOBJ("X")`. In this case, the command adds a single copy of object ***X*** to the object containing the variable computed. The new object will be initialized with the elements' values set by the modeller for the very first object of type ***X*** at the start of the simulation run. If object ***X*** contains descendants, they will be contained in the newly created object with only one copy. All the variables in the newly created object will be set as if they were already computed at the time of creation. Also, the flags for saving the data series will be those set by the modeller. The series for the elements saved from newly created objects will appear as missing values for the time steps before their creation.

  The function returns the copy of the object, so that the modeller can modify some of its elements, if necessary.

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

- `ADDOBJS(obj,"X")` or `obj->add_an_object("X")`: add new copy of the object ***X*** to object `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`, with the same values for all the elements and for the number of descendants.
- `ADDOBJS_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`.

The above commands are pretty slow when creating many copies of objects. Other members of the same family of commands to add objects permit to specify the number of new objects to create. All the following commands return the first of the set of objects created.

- `ADDONBJ("X", num)`: as `ADDOBJ` but generates `num` new copies.
- `ADDONBJS(obj,"X", num)`: as `ADDOBJS` but generates `num` new copies.
- `ADDONBJ_EX("X", num, exObj )`: as `ADDOBJ_EX` but generates `num` new copies.
- `ADDONBJS_EX(obj,"X", num, exObj )`: as `ADDOBJS_EX` but generates `num` new copies.

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=RNDDRAW("Firm", "MarketShare"); //Choose randomly an existing firm with prob=ms
cur1=ADDOBJ_EX("Firm",cur); //add a new firm identical to the chosen one
v[0]=V("MaxProductivity"); //highest productivity, computed in the market
v[1]=V("DevImitation");//Market parameter
v[2]=norm(v[0],v[1]); //draw randomly around best productivity
WRITEL(cur1, "Productivity",v[2]); //assign the new productivity (parameter)
WRITELS(cur1, "MarketShare",0,t); //assign 0 to variable market share at present time
v[0]=V("InitCapital"); //compute elsewhere the initial capital
WRITES(cur1, "Capital",v[0]); //assign the capital (parameter) of the new firm
WRITELS(cur1, "Production",0, t-1); //useless value, just mark the variable as computed at t-1
VS(cur1,"Production");//force the computation of the var Production with the new capital.
RESULT( 1 )
```

As another example, consider an equation creating a new market and inserting a set of firms. The newly created market contains only one firm, which are therefore incremented to reach the desired number.

```
EQUATION("NewMarket")
/*
Insert a new market with a number of new firms
Initialize the new firms with increasing IdFirm
*/

cur=ADDOBJ("Market"); //generate a new market
v[0]=V("NumNewFirm");
cur1=ADDNOBJS(cur,"Firm",v[0]-1); //creates remaining firms (one existed)
v[1]=1;
CYCLES(cur1->up, cur2, "Firm")
 {//scan all newly created firms
  WRITES(cur2,"IdFirm",v[1]++);
 }
RESULT( 1 )
```

6.8.18 `DELETE(obj)`

Remove object `obj` from the model. The object will be removed from the model, but its variables and parameters saved will show their value up the time step they have been computed, filling with missing values the remaining time steps.

The object removed is immediately removed from the model, and therefore this function must always be used for objects different from any object in use by the model, like `p` or `c`. For example, using the command `DELETE(p)` would imply to remove the object whose equation is being computed, which is not possible.

Notice that removing all objects of a certain type will make impossible to re-create new ones.

```
EQUATION("Exit")
/*
Remove firms with zero market shares
*/
v[1]=0;
CYCLE_SAFE(cur, "Firm")
 {
  v[0]=VS(cur,"ms");
  if(v[0]==0)
   {
    DELETE(cur);
    v[1]++;
   }
 }
RESULT(v[1] )
```

6.8.19 `INTERACT("message", value)`

This command interrupts the simulation presenting the user with the window of the LSD debugger showing the object whose element is computed by the equation. The simulation will continue after the user gives the command to continue the simulation.

The field `message` must be a brief text message, while value can be any real value, typically is used to show the content of a local system variable. The command returns a value, which will be the one inserted by the user in the cell of the LSD debugger window.

The command admits also the format `INTERACTS(obj, "message", value)`, which makes the LSD debugger show the object `obj`.

```
EQUATION("Exit")
/*
Remove firms with zero market shares
Deletion depending on the user
*/
v[1]=0;
CYCLE_SAFE(cur, "Firm")
 {
  v[0]=VS(cur,"ms");
  if(v[0]==0)
   {
    v[2]=VS(cur,"Age");
    v[1]=INTERACTS(cur, "Delete?", v[2]);
    if(v[1]==1)
     {
      DELETE(cur);
      v[1]++;
     }
   }
 }
RESULT(v[1] )
```

6.8.20 `PARAMETER`

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 at the beginning of the simulation and never again:

```
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 )
```

6.8.21 Lattices: creation and updating

LSD uses a rather limited set of graphical tools to present the results, relying on the export of data for any further elaboration besides those used in the Analysis of Results. However, there is an experimental implementation of lattices available.

Lattices are bi-dimensional grids composed by cells that can be set to different colors. Modellers can use two commands to create a lattice containing a window, and to change the color of a specific cell of the lattice. The grammar of these commands is described below.

The command to create a lattice is:

`init_lattice(pixW, pixH, nrow, ncol,"LRow" , "LCol", "LVar", obj, color)`

where the arguments have the following meaning:

- `pixW`: number of pixels of the window to be used for the width of the lattice
- `pixH`: number of pixels of the window to be used for the height of the lattice
- `nrow`: number of rows
- `ncol`: number of columns
- `color`: code for the initial color of the lattice.
- `"LRow"`: not used. Can be any character string.
- `"LCol"`: not used. Can be any character string.
- `"LVar"`: not used. Can be any character string.
- `obj`: not used. Can be any object, for example `NULL`

Obviously, the command to create a lattice must be executed only once for a simulation run. The command to change the color for a cell is:

`update_lattice(nline, ncol, color)`

where the arguments have the following meaning:

- `nline`: line number of the cell
- `ncol`: column number of the cell
- `color`: code for the color of the cell

The color codes are integer values from 0 to 20.

The lattices generated have several features and can be managed in several ways.

Before launching the simulation the user can set two different systems to update the lattice, using menu **Run/Lattice updating** from the LSD browser. The when the option is set on the lattices will be updated in a more efficient way for models where few cell switch frequently colors. Conversely, when the option is de-selected the refreshing is faster for large lattices where cells change colors only rarely.

In both cases, the lattice window at run time can have two modes of refreshing. In one case, the lattice is refreshed every time a cell changes color. On the other mode the refreshing takes

place only at the end of the step, for all the cells set during the step. To change refreshing mode simply click on the lattice window, that will switch between modes at every click.

A further option of the lattice window consists in clicking the window with the right button of the mouse. It is possible to save a dump of the window, as a graphical file formatted as an encapsulated postscript document.

To increase the speed of the simulation it is possible iconify the lattice window. The refreshing will continue to be performed in the background, and the speed of the simulation will increase substantially.

6.8.22 `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.

As an example, consider that your model requires a C++ vector, to be allocated to a given dimension, to be defined as a global variable, so that every equation may use its values. The modeller must declare the C++ global variable, assign memory to it, and remove the memory when the simulation is over. The following code performs these steps.

```
#include "fun_head.h"
double *myvector; //global C++ variable
MODELBEGIN

EQUATION("Init")
/*
Generate a vector. Equation computed only once
*/
v[0]=V("N");
mymatrix=new *double[(int)v[0]]; //Assign memory to the vector
PARAMETER
RESULT( 1)

MODELEND
void close_sim(void)
{
delete myvector;//remove the allocated memory
}
```

Such structures are rarely used, since they cannot be managed as standard LSD elements. For example, they cannot produced data to be used in Analysis of Results and the memory allocated to them must be explicitly dealt with. Moreover, LSD model structure is always able to reproduce whatever data structure may be required.

The only reason for using such instruments is efficiency, in terms of higher speed of execution. In some cases, typically dealing with matrices, it may be worth to use specifically defined data structures to store data that LSD standard variables can operate upon. For am example of this use, see the matrix-based implementation of the percolation model. In this model you can find two system variables declared on the top of the equation file:

```
#include "fun_head.h"

int **dat;
int **sta;

MODELBEGIN
...
```

Then, a L^SD variable, executed only once at the beginning of the simulation (stored in **Root**) allocate memory for these variables:

```
...

v[2]=V("NCol");
v[3]=V("NRow");
dat=new int *[(int)v[3]];
sta=new int *[(int)v[3]];
for(i=0; i<(int)v[3]; i++)
 {
  dat[i]=new int [(int)v[2]];
  sta[i]=new int [(int)v[2]];
  j=(int)v[2];
  for(h=0; h<j; h++)
   {sta[i][h]=0;
    if(RND<v[8])
     dat[i][h]=1;
    else
     dat[i][h]=0;
   }
 }
...
```

Other L^SD variables use these data, and, at the end of the simulation, the C++ function `close_sim()` releases the memory:

```
void close_sim(void)
{
double v[10];
v[0]=root->cal("NRow",0);
v[1]=root->cal("NCol",0);
for(v[2]=0; v[2]<v[1]; v[2]++)
 {
  delete sta[(int)v[2]];
  delete dat[(int)v[2]];
 }
 delete sta;
 delete dat;
}
```

6.8.23   Free pointer `hook`

Any object in a Lsd model contains a special field called hook. This field is a pointer to another object, like the other pointers − >up that allow to move from one object to its "father" object. The difference is that − >hook is never used by the system, and is left to the modellers for special purposes. Typically, it serves to speed up the access of a frequently requested object.

For example, suppose you have a model where the object consumer contains parameter **IdUsed** storing the code of product currently used. Moreover, suppose that the products can break at each time step with a probability contained in the object Firm. The equation of the consumer to check if the product breaks should be the following:

```
EQUATION("IdUsed")
/*
Id of the product used by the consumer.
Look whether the product breaks down or not. In teh first case choose a new product.
*/
v[0]=VL("IdUsed",1); //product used by the consumer

cur=SEARCH_CND("IdFirm",v[0]); //find the object with IdFirm equal to my IdUsed
```

```
v[2]=VS(cur,"BD"); //read the probability of breaking the product
if(RND<v[2])
 { //product broken
  cur1=RNDDRAWFAIRS(cur->up,"Firm"); //choose a new firm from the father of the former firm
  v[1]=VS(cur1, "IdFirm");
 }
else
 v[1]=v[0]; //product not broken, used the same product as before
RESULT(v[1] )
```

This equation can slow down the simulation in case it contains thousands of producers and of consumers, since the system has to scan many Firm's to find the one required by the consumer. The same code can be written as follows:

```
EQUATION("IdUsed")
/*
Id of the product used by the consumer.
Look whether the product breaks down or not. In teh first case choose a new product.
p->hook is the consumers' free pointer containing the firm used by the consumer.
*/
v[2]=VS(p->hook,"BD"); //read the probability of breaking the product
if(RND<v[2])
 { //product broken
  cur1=RNDDRAWFAIRS(cur->up,"Firm"); //choose a new firm from the father of the former firm
  p->hook=cur1;
  v[1]=VS(cur1, "IdFirm");
 }
else
 v[1]=v[0]; //product not broken, used the same product as before
RESULT(v[1] )
```

The code is faster because the equation do not need to scan all firms to find the desired one.

Beware that the modeller must ensure that the hook is always correctly assigned. In the example, you need to consider that at the very first time step the hook is not assigned. At the start of a simulation run all the hook's are set to NULL. Therefore, you may change the code as follows:

```
EQUATION("IdUsed")
/*
Id of the product used by the consumer.
Look whether the product breaks down or not. In teh first case choose a new product.
p->hook is the consumers' free pointer containing the firm used by the consumer.
*/
if(p->hook==NULL)
  {//executed the very first time step, when hook is not assigned
   v[0]=VL("IdUsed",1); //product used by the consumer
   cur=SEARCH_CND("IdFirm",v[0]); //find the object with IdFirm equal to my IdUsed
   v[2]=VS(cur,"BD"); //read the probability of breaking the product
   p->hook=cur;
   }
else
  v[2]=VS(p->hook,"BD"); //read the probability of breaking the product

if(RND<v[2])
 { //product broken
  cur1=RNDDRAWFAIRS(cur->up,"Firm"); //choose a new firm from the father of the former firm
  p->hook=cur1;
  v[1]=VS(cur1, "IdFirm");
 }
else
 v[1]=v[0]; //product not broken, used the same product as before
RESULT(v[1] )
```

6.8.24   `turbosearch, initturbo`

This is an experimental command, whose grammar is likely to be modified in future. See online documentation in case of error messages.

The `SEARCH_CND(...)` command is computationally impractical if the number of potential objects to scan is too large because its implementation needs to access all the objects. The command `turbosearch` allows an almost instantaneous access to an object in a large group[5].

The command has limitations in its current implementation.

- The group of objects on which `turbosearch` will be applied must be treated preventively with the command `initturbo`, which has some computational costs but must be executed only once.
- The group of objects allowing the `turbosearch` cannot be modified removing or adding new objects after the application of `initturbo`.
- The search can be made exclusively on their position, not on their content.

The grammar for `initturbo` requires the name of the objects to be searched `obj` and their total numbers `n`. Assuming `p` is the object parent of the group of `obj`, the initialization requires the following command:
`p->initturbo("obj", n);`

To obtain the ith object from the group and assign it to `cur` the command is:
`cur=p->turbosearch("obj",n,i);`

Let's see an example. The following code generates a group of objects called **Node** each containing in turn a descendant called **Link**. It then establishes a random number of links in between any **Node** and a fixed number of randomly chosen ones.

The code also avoids duplications and write the id of linked nodes.

```
EQUATION("InitNetUniform")
/*
Create a network with a fixed number of uni-directional links per node.
*/
v[0]=V("NumNodes"); //desired number of nodes
ADDNOBJ("Node",v[0]-1);
v[1]=1;
CYCLE(cur, "Node")
  WRITES(cur,"idNode",v[1]++); //assign a unique id to nodes

p->initturbo("Node", v[0]); //initialize the turbo structure
v[1]=V("NumLinks");

CYCLE(cur, "Node") //for each node
 {WRITES(cur,"nLinks",v[1]);
  ADDNOBJS(cur,"Link",v[1]-1); //add objects Link not created before
  v[3]=VS(cur,"idNode");  //register the id of the current node
  CYCLES(cur, cur1, "Link")
   { //for each link
    v[2]=v[3]; //force the initial computation
    while( v[2]==v[3] )
     {//repeat to avoid duplication or self-links
      v[2]=rnd_integer(1,v[0]);
      if(v[2]!=v[3])
      {
       CYCLES(cur, cur2, "Link")
        {
         if(VS(cur2, "LinkTo")==v[2])
          {//if a link already exist fake the drawing of itself
           v[2]=v[3];
           break;
          }
        }
      }
     };
   WRITES(cur1,"LinkTo",v[2]);
   cur3=p->turbosearch("Node", v[0],v[2]);//search the actual node corresponding to v[2]
   cur1->hook=cur3;
   }
```

---

[5]The number of steps to search in a group of N objects is linearly increasing with `SEARCH_CND` and increasing with LOG_{10} with `turbosearch`. For example, the average 50,000 steps necessary to search in a group of 100,000 objects become just 4.

```
  SORTS(cur,"Link","LinkTo", "UP");
 }
PARAMETER
RESULT( 1)
```

On an iMac (late 2013) with an Intel i5 processor the equation generating 10 links per node takes less then 2 seconds for 10^5 nodes, about 17 seconds for 10^6 and about 32 second for 20^6. The limitation for the application concerns the necessary memory, which is 0.58GB, 4.95GB and 9.85GB respectively.

6.9   User defined external functions

[to be completed]

6.10   External functions from C++ libraries

[to be completed]

Chapter 7

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.

7.1 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.

7.1.1 /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**.

7.1.2 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 subsequent distrubutions). 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++".

7.1.3 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.

7.2 Equations' programming errors

These errors appear when you try to compile a L^{S}D 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.*

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

7.2.1 fun_sd.cpp:17: error: XXX was not declared in this scope

This error occurs because at the line indicated (17 in the example), the command XXX is not understood by the compiler, probably because of mis-typing a correct command. Remember that in C++ and L^{S}D the case matters, so that, for example, v("X") is illegal, while V("X") is correct.

7.2.2 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 **Ctrl+m**. The LMM editor will show the matching opening curly bracket missing the closing matching 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.

7.2.3 lsd_gnu.exe: Permission denied

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

Close the running L^{S}D model program and re-compile.

7.2.4 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.

7.3 Simulation run errors

This section lists the errors occurring during a simulation run.

7.3.1   Simulation just run

If you complete a simulation run and try to start immediately a new one the system will not allow to proceed. The reason is that before starting a simulation run the system saves in a file the present content of the model. At the end of a simulation run the content is the final step of a simulation run, not the initial state as stored in the configuration file that produced the previous simulation.

If the user wants to replicate the just executed exercise must first re-load the configuration, also using the **Ctrl+w** shortcut.

If the user wants to continue the just terminated simulation beyond the number of steps, it is necessary first to save the current state of the model in a configuration file (menu **File/Save**) and the load the new configuration file.

7.3.2   The simulation cannot start because 'XXX' has not been initialized

Adding new parameters or lagged values require initialization. If the user only added a parameter, but did not provide an initialization, the system refuses to start under the assumption that the default value may be incorrect.

Move the browser to show the object containing the indicated element and press **Ctrl+i** to open the initialization window. This is sufficient to start a simulation run.

7.3.3   Error in equation for 'XXX'. The model does not contain any element 'YYY'

Why executing the equation for variable **XXX** the system found a call to the element labelled **YYY**. However, the system does not contain any element with such a label.

Likely reason is mispelling, differentiating, for example, the label **YYY** in the model configuration and `Yyy` as spelled in the equation's code. Fix the spelling error in either configuration or equations.

7.3.4   Error trying to compute variable XXX. Equation not found

The system found in the configuration the element **XXX** as a variable or function. But the equation file does not contain any equation with that name. Possibly the modeler forgot about it, but there can be other reasons.

One possible reason may be the different spelling between the configuration and the equation. To fix this problem use the same spelling.

If the equation is there, it may be possible that the currently used LSD model program has not been compiled with the new equations as added to the equation file. To check for this close the LSD model program and re-run it, and then try to run again the simulation.

A similar problem may be that the equation file used for the compilation by the system is different from the equation file edited by the user. Thus, the equations are written there, but the system does not use them. To fix this error check the file name where the equations are written. For precaution select all the text of the equations. Then, in LMM uses menu **Model / Show equations**. The file shown now is the one used for the equations. If this is empty, or not updated, paste there the text previously copied. Otherwise search for the previously saved file and copy the text.

7.3.5   At time 'T' the equation for 'XXX' produced the non-valid value...

The equation for element **XXX** contains a computation producing a non valid number, such as a division by zero (code **inf**), the logarithm of a negative number, etc.

Using the LSD debugger try to reconstruct the motivation for the un-expected error (negative market shares? Null total production?) and fix the code as necessary.

7.3.6   Search for 'XXX' failed

The system search, possibly conditionally, for elements not found.

To fix the error ensures that the necessary conditions for a successful search are met. Typically due to mispelling or for search of non-existing values.

7.3.7   Operation 'zzz' requested to a NULL pointer in equation for 'XXX'

If you wrote in your code an operation using a non-initialized pointer, such as `VS(cur, ''X'')` without having firstly assigned `cur` this error may occur. It may also result in a crash. To fix it ensure that all pointers are correctly assigned controlling the code for 'XXX'.

Note that the object `c` exists with certainty only for functions, since their code is executed exclusively when another equation asks for its value, and therefore there is a *calling* object. For variables, on the contrary, it may be possible that the system asks to a variable to be updated without being triggered by others. In these cases the object `c` is a pointer, and its use leads to this error.

7.3.8   LSD crashes - `DEBUG_AT(T)`

The major problem when a crash occurs is to identify the equation causing the crash, since once identified it is possible to restrict the code to control. For that purpose you can use the command `DEBUG_AT(t)`. This causes the program to write into a file the label of every variable *before* computing its equation, so that reading the last label in the file it is possible to identify the equation with the problem.

The command must be located immediately after equation file header:

```
MODELBEGIN
DEBUG_AT(42)
```

The command will be ignored until the simulation reaches time step indicated, in the example 42. From that point onwards the system will write in file **log.log** the sequence of variables starting the computation of their equations. After the crash open the file indicated, in the model directory (you can use also the LMM editor for this purpose) and read the very last line, which will provide the name of the variable causing the crash.

7.3.9   Use of *gdb* debugger

Use the **gdb** debugger to follow the simulation instruction-by-instruction. A debugger is a program that allows other programs to run in a controlled environment. When a program is run in a debugger, it is possible to observe in detail what it does, for example interrupting its calculations and checking the values of its internal variables. The debugger works as if the single lines of code fire their commands only when explicitly directed.

Note that the **gdb** is a C++ debugger, and, therefore, it is not able to explore LSD specific code, Objects etc., but their technical representation. **Gdb** permits, for example, to investigate line by line the execution of an equation. However, this means that one has also to follow line-by-line a lot of non-interesting technical code. LSD programmers may find much more useful the LSD Debugger: it permits to follow a simulation by individual equations. When one variable, marked to be debugged, is computed, the LSD debugger interrupts the simulation, shows the value produced in the equation, the values of v[n] temporary values, the whole models' values, etc., permitting to understand, most of times, the cause of an error. See the LSD Debugger in section **??**. However, the gdb debugger is still useful in case of very hard problems where the modeler needs to observe line by line the execution of the code within one equation.

The LSD model program in the **gdb** debugger runs as usual (slightly slower, if anything) but the user can interrupt the program at any point, check the value of any variable, make the program proceed by single steps, and a whole series of other functionality meant to help finding out misbehaviours or errors in the program.

In case of a LSD model program, it is usually useful to focus on one equation and see the behaviour of the program while executing that code. In the following we provide instructions on a few **gdb** commands. They all take the form of commands to be typed at the line prompt of the **gdb** shell. More information can be found in the **gdb** manual, available on the net, or simply typing help in the **gdb** shell, possibly followed by a precise command.

To use the debugger you need LMM to compile the LSD model program using the option `SWITCH_CC=-g`, to be inserted in menu **Model**

Model compilation options. After confirming this option recompile and shut down the resulting LSD model program.

You can run the **gdb** in two ways. Using the menu **Model / gdb Debug** you will have a command shell already loaded with **gdb**. There you can set a breakpoint and run the model (see below).

Alternatively, right-click on the code of the equation exactly on the line you would like to start the debugging. In the resulting menu choose the option **Place a break and run gdb**. This will open the shell with the breakpoint already set at that line.

In the **gdb** shell you can use the following commands:

- `break fun_XXX.cpp:132`: places a breakpoint at line 132 of the file `fun_XXX.cpp`. When the execution of the program reaches that line, gdb interrupts the program and offers the users with the gdb prompt.

- `break 232`: places a break in the currently considered source file at line 232;

- `run`: causes the program to run. The execution will be interrupted at one of the following events: the normal end of the program is reached; the flow of the program finds a breakpoint; the program crashes.

- `break myfun`: places a breakpoint at the first line of the function myfun;

- `break variable::fun` places a breakpoint at the first line of the method fun of the object variable.

- `print X`: shows the content of the variable X. For example `print v[3]` will show the content of the local variable indicated.

- `display X`: shows the content of the variable X, which remains visible through other commands (e.g. next, step etc.) until undisplay is called.

- `step`: makes just one step in the execution of the program

- `next`: like step, but, if a subfunction is found, next "step over" it without entering within the code of the function

- `continue`: resort the program continuing after an interruption;

- `delete [n]`: remove the breakpoints. If no number is specified all the breakpoints are removed. Otherwise, only the one marked with the specified number is eliminated.

- `list`: shows the source code lines above and after the one that is going to be executed.

- `list -10 [10]`: shows 10 previous [subsequent] lines in respect of the one shown before.

- `cond 2 v[0] == 3`: assigns a condition to a breakpoint, so that the execution is interrupted only if the condition is true. In the example, the second breakpoint specified is active only if the value of v[0] is identical to 3, otherwise the execution will not be interrupted.

- `bt` (backtrace): shows the stack of the program. It is the list of functions that are called at the moment in the execution of the program.

For more information on this and other commands, use the help in gdb (just type help and the name of the command). Moreover, in gdb works the command completion, so that typing, for example, "disp" and then pressing the tab key, the command display is automatically completed. In case of possible ambiguity (e.g. typing only "dis" and tab) the prompt shows the list of alternative among which to choose (this is also a weird, but rather useful, way to discover new functionalities).