

Aim of the course

The goal of this course is to teach students how to use computers to make (some) sense of economic phenomena.

In practice, the course will teach how to **design, build, use** and **distribute** computer simulation models. This course is based on the idea that computer simulations in social sciences require a different approach from simulations for models of realities where plenty of reliable data are available.

In social sciences a simulation model must not only replicate numerical sequences from the real world, but, to provide an increment of knowledge, need also to show the *causes* linking the model specification to the results obtained.

Course content

The course concern both the philosophy of simulation and the technical steps required. You may consider these lessons also as an introductory course to the use of Laboratory for Simulation Development (Lsd), a language for simulation written by Marco Valente. Lsd is freely available for download from:

<http://www.business.auc.dk/lsd>

Lsd is available for Windows and Linux platforms, requiring only software already present in these platforms. Lsd and the other distributed packages are free (GPL). Lsd runs also under MacOS X; users should read the Lsd installation instruction.

Introduction to Computer Simulations

This lesson will discuss the basic elements of a simulation model. We will use an extremely simple example both for the presentation and for the first exercises, meant to precisely define the terms used in the course and get started with the Lsd main commands.

The lessons of the course are designed to be accompanied by practical exercises.

Simulation models

Simulation models “simulate” a given reality by producing a series of values for one or more variables. For each time step each variable takes a value produced by an equation. We can express an equation in general terms as:

$$X_t = f(X_{t-1}, Y_t, Z_{t-2}, \alpha, \dots)$$

meaning that the value of the variable X at time t is computed as the elaboration of:

- Past values of the same variables (X_{t-1}), and/or
- present and past values of other variables (Y_t, Z_{t-2}), and/or
- parameters (α)

Simulation models

The elaboration used can be algebraic, logical, or any other conceivable operation. Let's start with a very simple example. Let's consider the equation:

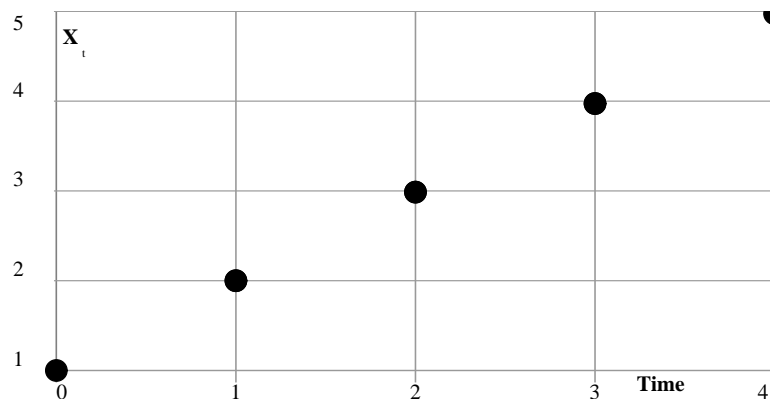
$$X_t = X_{t-1} + 1$$

We will obtain the following values:

X_t	Values
X_0	1
X_1	2
X_2	3
X_3	4
...	...

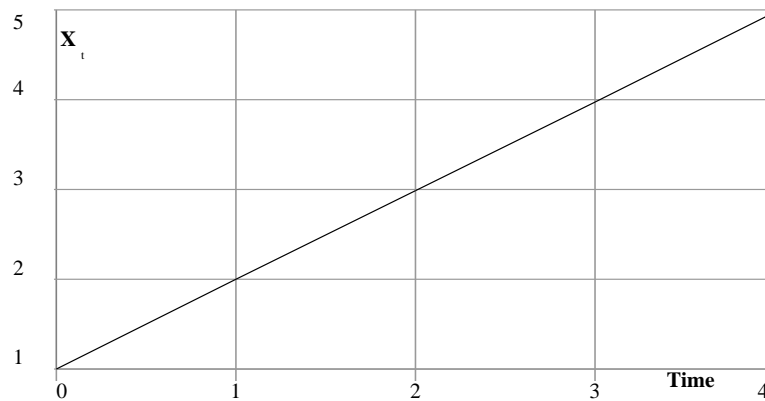
Simulation models

We can express this function using a Cartesian plane, having the time on the X axis and the value of the variable on the Y axis.



Simulation models

We can simplify the reading of the data interpolating lines in between two time points.



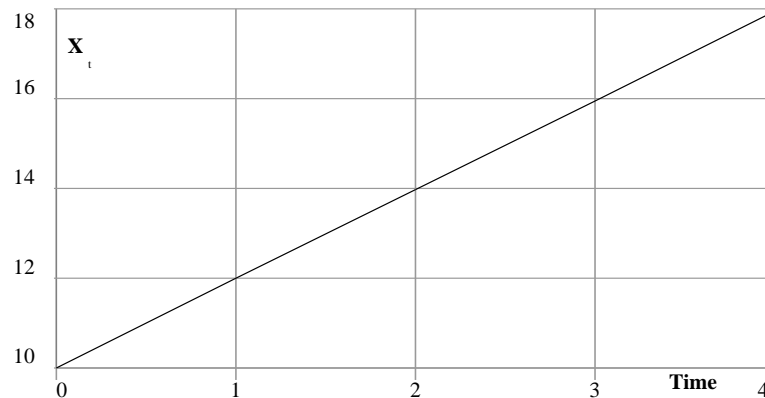
Initial Values

The data produced by a simulation run depend on the equations of the model (in our case $X_t = X_{t-1} + 1$) and by the initial values with which the simulation begins. In our example, at the very first time step we decided to use $X_0 = 1$. Moreover, instead of the 1 we may use any real value. We may refer to this value as the parameter σ . Instead of using $\sigma = 1$ and $X_0 = 1$ we can use any value.

In the next example we set $X_0 = 10$ and $\sigma = 2$.

Simulation models

Simulation using $X_0 = 10$ and $\sigma = 2$



Number of Objects

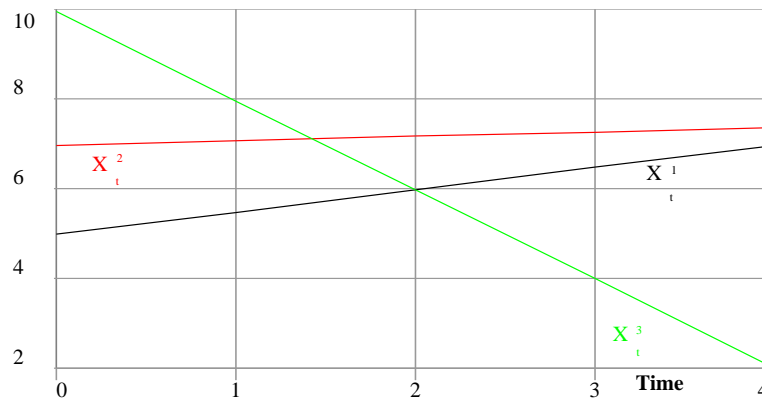
Generally a simulation is used to compare several series computed in parallel. In mathematics it is used the concept of **vectors**. We should have a vector $\vec{X}_t = \{X_t^1, X_t^2, \dots, X_t^m\}$ where each element is computed with the same function, but different parameters $X_t^i = X_{t-1}^i + \sigma^i$.

In programming languages it is generally used the concept of **objects**. An object is name (or label) indicating a whole set of variable, parameters, or even other objects. In our example we may define the object $Obj = \{X_t, \sigma\}$. We can then define in our model several objects Obj , and assign to each of the variables and parameters in the different copies different initial values.

In the next example we use three Obj with $X_0 = 5, 7, 10$ and $\sigma = 0.5, 0.1, -2$.

Simulation models

Simulation with three objects *Obj* with initial values: $X_0 = 5, 7, 10$ and $\sigma = 0.5, 0.1, -2$.



Summary: elements of a simulation model

Let's summarize the elements composing a model. First, we have variables:

- **Variables**, composed by:
 - *Label*: just a name (in the example X);
 - *Equation*: the formula to use to compute its values through time (in the example $X_{t-1} + \sigma$)
 - *Init. values*: (only if required at time $t = 0$) the value to kick-off the simulation (in the example $X_0 = 0$).

Summary: elements of a simulation model

Then we have parameters:

- **Variables**, composed by:
 - *Label*: just a name (in the example X);
 - *Equation*: the formula to use to compute its values through time (in the example $X_{t-1} + \sigma$)
 - *Init. values*: (only if required at time $t = 0$) the value to kick-off the simulation (in the example $X_0 = 0$).
- **Parameters**, composed by:
 - *Label*: just a name (in the example σ);
 - *Init. values*: the value of the parameter (in the example $\sigma = 1$).

Summary: elements of a simulation model

Then the objects, to store the elements of the model:

- **Variables**, composed by:
 - *Label*: just a name (in the example X);
 - *Equation*: the formula to use to compute its values through time (in the example $X_{t-1} + \sigma$)
 - *Init. values*: (only if required at time $t = 0$) the value to kick-off the simulation (in the example $X_0 = 0$).
- **Parameters**, composed by:
 - *Label*: just a name (in the example σ);
 - *Init. values*: the value of the parameter (in the example $\sigma = 1$).
- **Objects**: the containers of any other element of the model
 - *Label*: just a name (in the example Obj);
 - *Number of objects*: number of copies (in the second example 3).

Summary: elements of a simulation model

Finally, the settings used to run the simulation

- **Variables**, composed by:
 - *Label*: just a name (in the example X);
 - *Equation*: the formula to use to compute its values through time (in the example $X_{t-1} + \sigma$)
 - *Init. values*: (only if used at time $t = 0$) the value to kick-off the simulation (in the example $X_0 = 0$).
- **Parameters**, composed by:
 - *Label*: just a name (in the example σ);
 - *Init. values*: the value of the parameter (in the example $\sigma = 1$).
- **Objects**: the containers of any other element of the model
 - *Label*: just a name (in the example Obj);
 - *Number of objects*: number of copies (in the second example 3).
- **Sim. settings**: num. of steps, values to save (4 and save the X 's), others.

Languages to implement a simulation model

Simulation models can be implemented not only with computers but with any type of language, **if the elements of the model are rigorously defined in all the details**. For example, one can use paper and pencil, a pocket calculator, a spreadsheet, or simply mental reasoning.

A language allows the model to interact with the user, and to users to explain the model to others.

Any language can be used, as long as one is able to define unambiguously the model's element. Computers are particularly suited because:

- Computers need to be instructed very precisely: no possibility for ambiguous statements.
- Computers can manage reliably large amounts of data very quickly.

Languages to implement a simulation model

However, a model producing tons of data in a nanosec is useless, unless a user can make sense of the model's behaviour. Even simple models produce easily very messy results, because of the interaction among many variables, though simple they can be. Experience shows that it is generally at least as difficult to make sense of the results than writing the model producing them.

The course will pose particular importance in teaching the tools, logical and practical, required to understand how the results of a simulation have been produced. This aspect is of minor importance to physical sciences, where a model can be judged correct or false comparing the results with real data. In this case, understanding what happens within the box of the model is not much relevant. However, they are of extreme importance in social sciences, where simulation models represent realities where reliable quantitative measurements are rare, if available at all.

Computer Languages for Simulation Models

A language serves not only to represent the model (i.e. define variables, parameters, equations etc.) and to execute a run. There are other elements that are necessary for a model to be of any use:

- Grammar for the equations (to express the elaborations);
- Interfaces to implement the model's elements and initializations;
- Interfaces to read the data produced;
- Interfaces to interpret the model's behaviour;
- Interfaces to modify initial data;
- Interfaces to *debug* the model;
- Documentation of the model.

Laboratory for Simulation Development - Lsd

We are going to use a simulation language developed on purpose for building and using simulation models, with the aim of gaining knowledge by reproducing artificial dynamics similar to real ones.

Lsd is a pure language, in that it is not a general model: a Lsd user needs to specify all the elements of a model outlined above. The advantage of Lsd is that when users generate a model, the system automatically provides a rich set of sophisticated interfaces for a professional management of the simulation runs.

In the following slides we will present a general overview of the Lsd system, and then we will start a guided exercise on its use.

Laboratory for Simulation Development - Lsd

The phases of a using Lsd model are the following:

1. **Equations:** write the code for the model as the elaborations required to produce variables' values, exactly like difference equation models.
2. **Model Structure:** define objects, variables and parameters, specifying the labels and nature of the entities represented in the model.
3. **Model initialization and settings:** define the numerical values to be used to start a simulation run.
4. **Simulate the model:** run simulations under a different modalities for, e.g., catching errors, observing results, robustness tests, etc.
5. **Analysis of results:** use the simulation data to generate plots, computing statistics, or exporting data to external statistical packages.

Laboratory for Simulation Development - Lsd

All the operations mentioned above are managed with efficient graphical interfaces endowed with on-line help and error controls. For example, the system signals if a user define an element with a label already used; divisions by zero cause the system to interrupt the simulation signalling the equation concerned and showing all intermediate values used until the error.

The programming principle of Lsd is to allow users to do quickly and easily the most frequent operations. However, complex elaborations are always admitted, either using advanced Lsd functions, or, in the limit, resorting to the underlining C++ layer.

In the following we start to provide some snapshot of the system we will use to generate an early, very simple model.

Laboratory for Simulation Development - Lsd

The **equations** of a model in Lsd are written as a much simplified C++ code extended with Lsd-specific commands. The text is inserted in a file using **Lsd Model Manager - LMM**. LMM is a sort of integrated environment to manage Lsd models. As any compiled language, Lsd requires a rather complicated process to generate the executable program from the source code. Moreover, generally modellers work on more than one model at any one time, risking to make confusion among different models and different versions of the same model.

LMM hides all the technical operations involved in the making of the simulation program, providing a simple management of many projects.

Lsd Model Manager - LMM

LMM performs three classes of operations:

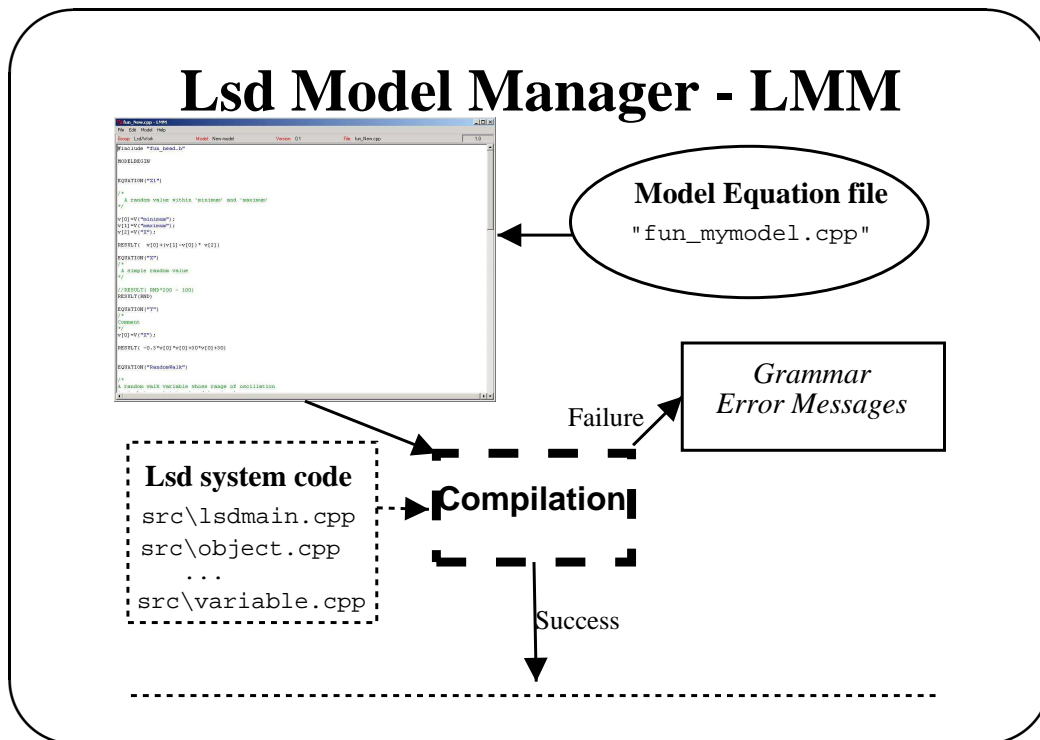
1. **Model Browser:** Each model is composed by several files located in a directory. LMM allow users to create, copy, edit or delete models in an ordered directory structure. This function is of vital importance, as everybody who worked with more than one model can testify.
2. **Editor:** allows users to easily write the equations' code. The LMM equation editor offers colored text for the different types of code, guided "wizards" to write the most frequent commands, on-line help on available Lsd commands, and many other handy functions.
3. **Compilation:** run for the user all the system commands required to generate the Lsd model program from the equations' code. Non-expert users need only to press a button using the default options. Otherwise, users can specify separately system-level compilation options (typically dependent on the OS used) and model specific options (e.g. optimization options).

Lsd Model Manager - LMM

Modellers use LMM to write in a file blocks of code for each equation in the model. This file is called **equation's file** and is just a text file to be compiled and linked with the rest of the Lsd source code.

As any source file, the equation file may (and usually does) contain errors, that is, lines of code that do not respect the C++/Lsd grammar. LMM tries to compile the equation file and, in case of errors, provides indications on the likely location of the error(s). In these cases, the modeller needs to correct the error (typically a misspelled command), and re-compile.

In case of success, the compiled equation file is linked to the rest of Lsd code to generate a new program, called **Lsd model program**. The following schema summarize the uses of LMM.



Marco Valente

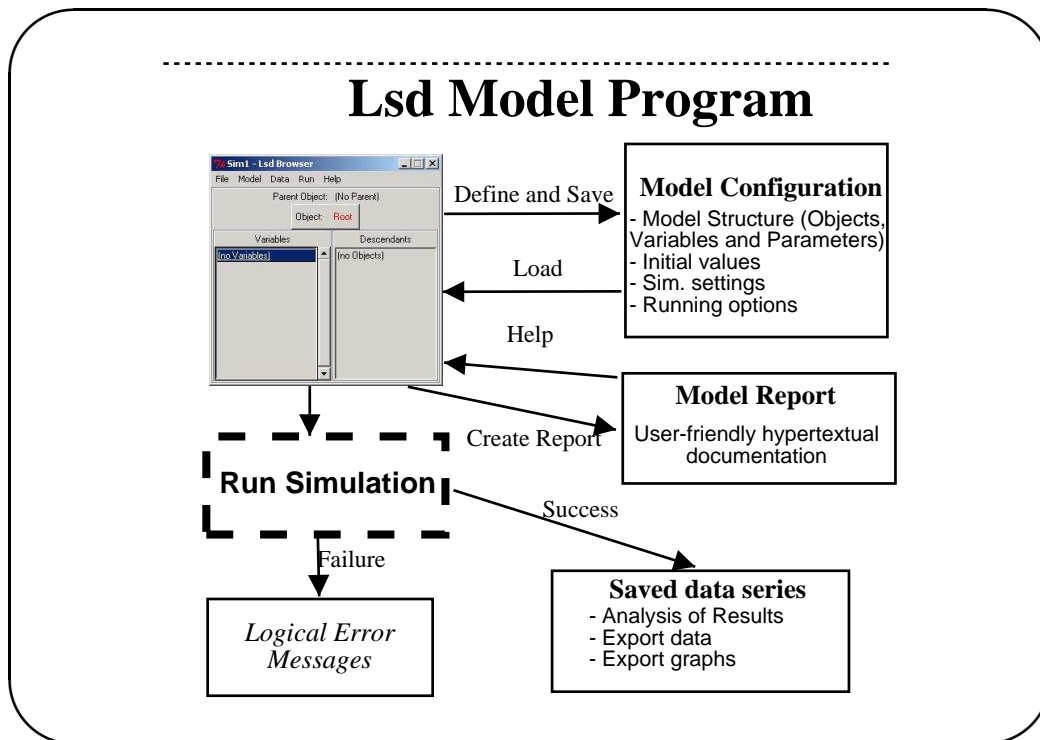
Università dell'Aquila

Lsd model programs

A successful compilation generates a **Lsd model program**. All Lsd model programs have identical interfaces, but each contains only the equations of the model it was compiled for.

Modellers use the Lsd model programs to make all the operations concerning simulations, apart the modification of the equations (the compiled code for the equations is embodied in the Lsd model program, and cannot be changed with the program itself. This limitation is compensated by the high computational speed provided by compiled C++ code).

In particular, Lsd model programs generate, save and load model configurations; run simulations; catch and analyse run-time errors; analyse simulation data; prepare and generate automatically a detailed documentation of the model.



Simulation 0

Let's implement in Lsd the model $X_t = X_{t-1} + m$. We will implement the following steps:

1. Use LMM to create a new model, call it "Simulation 0".
2. Generate an "empty" Lsd model program (without equations' code).
3. Use the Lsd model program to define the model structure Obj , X , and m .
4. Save the configuration and kill the Lsd model program.
5. Write the C++/Lsd code for the equation of X and generate the new Lsd model program.
6. Load the saved configuration, set the initial values (i.e. the number of copies of Obj , the values for X_0 's, and m 's).
7. Run the simulation and analyse the result.

Start LMM and create a new (empty) model

Locate on your disk the directory containing the Lsd system. In the top-most directory run the **run.bat** if you are a MS Windows user, or **./lmm** for Unix users (the system has minimal requirements to run under Unix and no requirements at all for Windows systems; see installation instructions.)

LMM starts with the following window

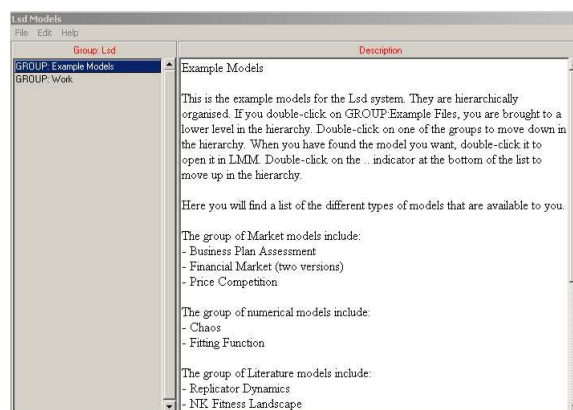


Marco Valente

Università dell'Aquila

Start LMM and create a new (empty) model

The introductory window offers the options to use LMM for working with Lsd models, or simply to edit a file. Choose **Browse Models** and press **Ok**. The following window appears.



Marco Valente

Università dell'Aquila

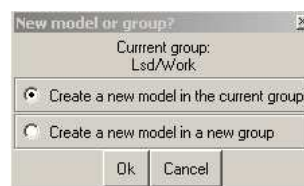
Start LMM and create a new (empty) model

The **LMM model browser** shows all the models available in your Lsd installation. Models are stored in directories which can be grouped in **Groups**. The model browser allows to visit the groups' and the model contained there. Use the arrows or the mouse to observe the group of example models (choose the final “..” in the lists to go up from one group).

Selecting a model LMM prepares to work on that model: the model browser disappears and the editing window shows a description text for the model (which is the same showed in the model browser when a model is highlighted).

Start LMM and create a new (empty) model

To open the model browser from LMM select menu **Model/Browse Models**. Use the browser to enter in the group **Work** (where one should place models under construction). Here select the menu in the browser **Edit-New model/group**. You are offered the possibility to create a new model or a new group in the current group.



Use the default selection (create a new model) and press **Ok**.

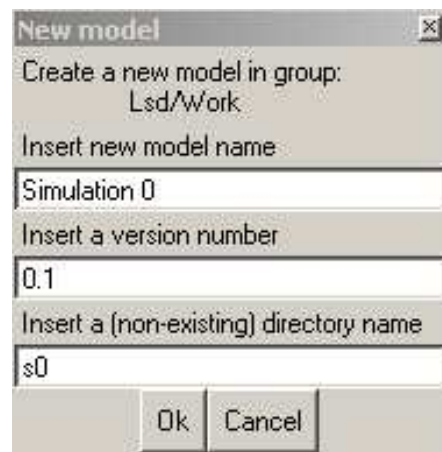
Start LMM and create a new (empty) model

Now LMM asks for the information concerning the new model. You are requested to provide:

1. A name for the new model. This text will be used by LMM to indicate the model project. It can be any text, though it would be better to constrain the name to a few short words.
2. A version number. This value serves to distinguish among different versions of the same model. It is treated as an additional part of the model name.
3. A directory name. This name will be used to generate a directory, and therefore should not contain spaces or other special characters, and should not be the name of an already existing directory (the system will prevent such events).

Start LMM and create a new (empty) model

Insert the text as indicated below and press **Ok**.



New model

Create a new model in group:
Lsd/Work

Insert new model name
Simulation 0


Insert a version number
0.1

Insert a (non-existing) directory name
s0

Ok Cancel

Start LMM and create a new (empty) model

Now a new model has been initiated. LMM shows in its editor the equation file for the model, at this very moment empty, while the header shows model's group, name, version and the file open.



The screenshot shows the LMM editor window titled 'fun_s0.cpp - LMM'. The menu bar includes 'File', 'Edit', 'Model', and 'Help'. Below the menu bar, a status bar displays 'Group: Lsd/Work', 'Model: Simulation 0', 'Version: 0.1', and 'File: fun_s0.cpp'. The main editor area contains the following code:

```
#include "fun_head.h"

MODELBEGIN

MODELEND

void close_sim(void)
{
}
```

Start LMM and create a new (empty) model

Before describing the model, a warning is due. The name of the equation is crucially important for LMM, since it needs this file to create the Lsd model program. Modellers need to use only one equation file, and be sure that the edited file is the one LMM considers as the model's equation file. When a model is selected, menu **Model/Show equation** loads in the editor the file currently considered as the equation file. Re-naming the equation file means that all the modifications to the file will not be used to create a Lsd model program.

It is always possible to modify the LMM choice of file (see menu **Model/Model Compilation Options**), but it is rarely a good idea.

Start LMM and create a new (empty) model

Before continuing, let's give a look at the equation file:

- `#include "fun_head.h"`: C++ command to include the indicated file. The file contains declarations and macros to express Lsd functions.
- `MODELBEGIN`: marks the line after which modellers can insert equations. This line declares a function, so that one may declare global C++ variables before this line.
- `MODELEND`: marks the end of the equations.
- `void close_sim() { }`: C++ function used only for very special cases. It is a C++ function executed at the end of each simulation run. Some models, during a simulation run, perform operations that require a subsequent reversing operation, like the allocation of memory to be freed.

Running a (useless) Lsd model program

To insert equations we need to write code, and writing code involves almost always also writing errors. For the time being let's ignore the equations' code and explore some of the interfaces of a Lsd model program. After that we will come back to the equation file, to learn how to write equations, ... and how to fix errors.

Note that we will create a variable in the model without providing an equation for it. In Lsd the two aspects of writing equations and building the model structure are completely independent. Of course, in order to run a simulation we need to have both the variables declared and their equations compiled, but the order in which we do the two operations is not relevant.

In this exercise we will first declare the variable, and save the model structure. Then we will create a new Lsd model program with the equation's code, reload the structure and eventually run a simulation.

Running a (useless) Lsd model program

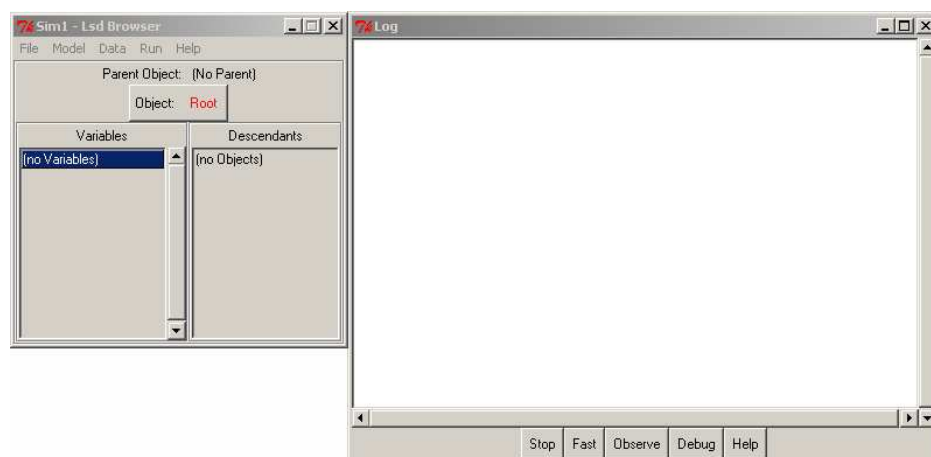
We can safely assume that an empty equation file contains no error, so we can compile it^a. Use menu **Model/Run Model**. This commands will tell LMM to compile the equation file and all the other Lsd source files to generated a Lsd model program.

If it is the very first time you create model the compilation process will require some time, because the system needs to compile a lot of code. Any subsequent compilation, instead, requires at most to compile the equation file only, spending a much shorter time.

At the end of the compilation LMM starts the Lsd model program, appearing as in the next window.

^aIf the compilation fails it is probably due to installation errors. See the manual on System Compilation Options.

Lsd model programs



Lsd model programs

There are two default windows for a Lsd model program: the **Lsd Browser** and the **Log** window. The Lsd Browser shows the content of the model and allows operations before and after a simulation run. The Log window contains messages from the system and provides the control during a simulation run.

Initially, a Lsd model program is empty; that is, contains no model elements (but for one special object, see below). We are going to create a model configuration and save it in a file, so that next time we will be able to re-use it.

Lsd model programs

The Lsd Browser shows the structure of one object. Any model must compulsorily have a **Root** object, which is the one shown by the otherwise empty Lsd Browser. Any object (but **Root**) is contained in a “parent” object and can contain variables, parameters or other (descending) objects. For the moment **Root** is empty, as shown by the two empty lists **Variables** and **Descendants**.

Lsd Browser

The Lsd Browser is controlled using the available menus:

- **File:** Load and save configuration files.
- **Model:** Create a model structure (define variables, parameters, objects); generate documentation.
- **Data:** Define and observe initial values; analyse or save results.
- **Run:** Define simulation settings and run simulations.
- **Help:** Open help pages on Lsd interfaces and read the model documentation.

Generate Model Structure

Let's use the Lsd Browser to generate the model structure. We need to define one object containing a variable and a parameter.

IMPORTANT: to label model entities use only characters (no spaces or other symbols). Capital letters differ from normal text.

1. **Model/Add Descending Obj:** Create an object `Obj`.
2. Click on `Obj` in the list of **Descendants**. The Browser shows the new object.
3. **Model/Add a Variable:** Create a variable `X` with 1 lag.
4. **Model/Add a Parameter:** Create a parameter `m`.
5. **File/Save as:** Save the current configuration in a new file `Sim1`.

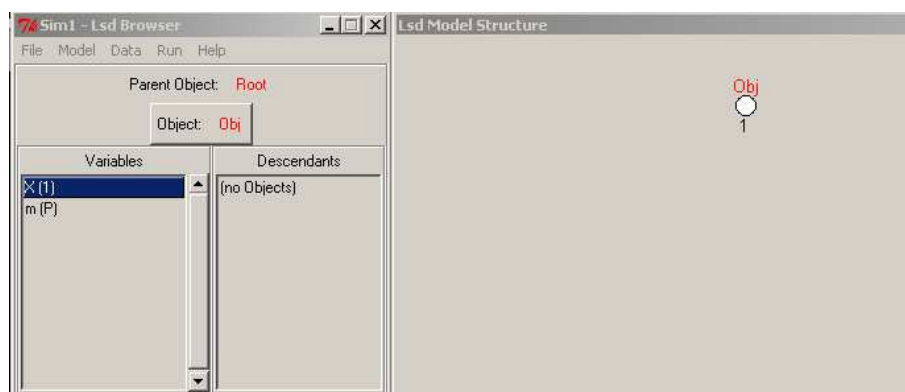
Generate Model Structure

Note that when we have defined the variable we have specified that it is used with one lag. This is necessary because during a simulation Lsd stores, by default, only the values for the present time step. If we did not tell the system that X needed to keep its value for one period, at any time t Lsd would have already discarded the value for time $t - 1$.

If the modeller forgot to declare the lag the system would have issued a message and interrupted the simulation. Then the modeller will have to edit the variable declaration adding the lag.

Generate Model Structure

The Lsd model program shows the model structure both in the Lsd Browser and with a graphical representation, as depicted below:



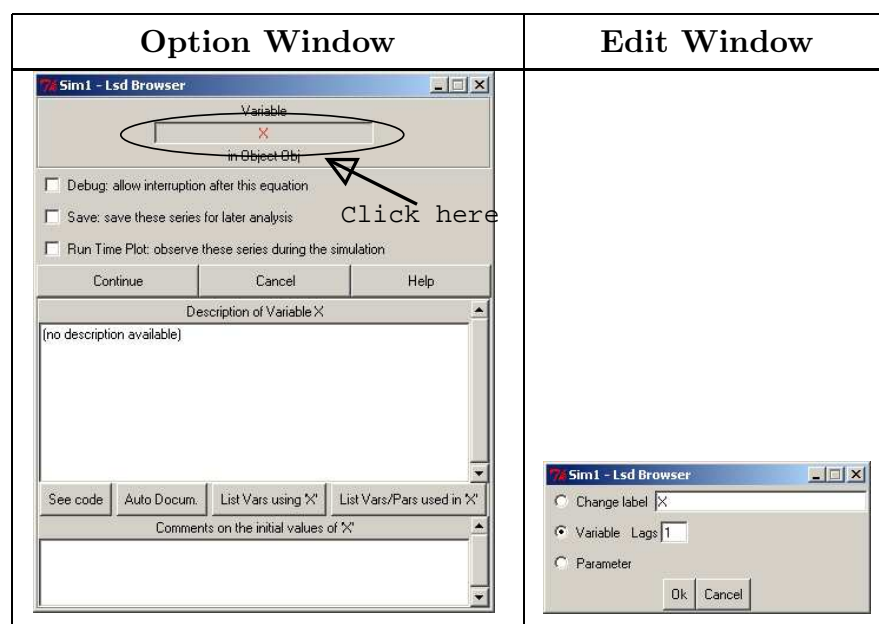
Edit Model Structures

It is frequent the case that model structures need to be corrected, either because of errors or because of modifications of the model (e.g. transforming a parameter in a variable). In order to remove an element from the model structure you need to edit its label and assign an empty label. It is not possible to move an element from one object to another, nor to create an element with the same label of an existing element. To move an element remove it from its object and create a new one in the correct object.

To modify a variable or parameter, for example to change its label, double-click on its label shown in the Lsd Browser. The resulting window allows to set a series of options for the element. In this window double-click on the label of the element (written in red) to open a new window allowing the modification of the element.

Marco Valente

Università dell'Aquila



Marco Valente

Università dell'Aquila

Write model's equations

We have now generated a file containing the structure of the model (but not the initialization; we will do that later). It is not time to write the equation for the variable X . To do this we need to close the Lsd model program because we will build a new one with the new equation file. In the Lsd model program choose menu **File/Quit** and confirm.

Let's go back to the LMM window. If don't have it on your screen, locate its icon on the menu bar and click on it.

Write model's equations

Equations in Lsd must be conceived as separated chunks of code that are repeated again and again at any time step.

In practice, for each equation we need to insert in the equation file two lines marking the beginning and the end of the code concerning the equation. Within these two lines we can place any C++/Lsd compliant code.

The order in which the equations appear in the equation file is not relevant.

LMM - The equation file

The equation $X_t = X_{t-1} + m$ is expressed as:

```
#include "fun_head.h"
MODELBEGIN

EQUATION("X") //begin the equation for X
/*
The new X is equal to its own past value plus 'm'
*/
v[0]=VL("X",1); //past value of X, lagged of 1 period
v[1]=V("m"); //current value of m

RESULT(v[0]+v[1] ) //final result of X

MODELEND
```

LMM - The equation file

Let's see what this code means. First of all, there are some comments, written in plain English, that the compiler ignores. There are two types of comments:

```
//this is a single line comment

/*
This is a multi-line comment
*/
```

The first type of comment spans only until the end of the line. Instead, the multi-line comment begins the symbol `/*` and terminates at `*/`.

LMM - The equation file

In Lsd the equations are written as separate blocks of code beginning with `EQUATION('VarName')` and ending with `RESULT(anyValue)`, where `VarName` is the label of variable, and `anyValue` is a real-valued number.

In between these lines the modeller can place any code. Generally, modellers use the Lsd functions that provide information concerning the state of the model. The most used Lsd function by far is the function that returns the value of a variable or a parameter in the model. The function `V('VarOrParName')` search in the model the name of the item specified and returns its current value. The variant `VL('VarOrParName',lag)` returns the value of the item `lag` periods ago. Note that `V('VarOrParName')=VL('VarOrParName',0)`

There are many Lsd functions to be used for any necessary computation. A complete list of Lsd functions is provided in the LMM help pages at the link [Macros for Lsd Equations](#).

LMM - The equation file

The `v[0]`, `v[1]`, `v[2]`, etc. are temporary C++ variables whose life spans only the time of the computation of one equation. They are normally used as storing places for intermediate calculations. In fact, the whole equations could be written as:

```
EQUATION("X") //begin the equation for X
RESULT(VL("X",1)+V("m")) //final result of X
```

However, this way of expressing equations is, for even relatively simple equations, much more difficult to read, and therefore it is generally better to assign to one of the `v[...]` variable all the required values and then writing the result as their elaboration.

LMM - The equation file

To write the code for the equations in LMM you can just type the symbols as reported. However, LMM is an editor specifically designed to write code for Lsd models. Many **scripts** make LMM insert the most frequently used code with the user just typing the necessary information. For example, you can have a script inserting for you the text for `EQUATION('VarLabel') ... RESULT()` pressing **control-i** to show the available scripts, **e** for inserting an equation and typing the label to write in place of `VarLabel`.

The scripts place the requested text in the position of the cursor. The scripts can also be invoked by using menu **Edit/Insert Lsd script** or by pressing the right button of the mouse and selecting the script item.

LMM - The equation file

All the scripts request the user to insert the relevant information, proposing default values that the user needs to adapt. For example, to insert the command:

```
v[0]=VL("X",1);
```

you have to modify the default proposed fields as indicated below:

Default values	User chosen values
<p>Insert a 'V(...)' command</p> <p>Type below the number v[x] to which assign the result 0</p> <p>Type below the label of the variable to compute Label</p> <p>Type below the lag to be used 0</p> <p>Type below the object to which request the computation p</p> <p>Ok Help Esc</p>	<p>Insert a 'V(...)' command</p> <p>Type below the number v[x] to which assign the result 0</p> <p>Type below the label of the variable to compute X</p> <p>Type below the lag to be used 1</p> <p>Type below the object to which request the computation p</p> <p>Ok Help Esc</p>

LMM - Compiling

IMPORTANT: when you finish to write code you need to save the file. In fact, the text you see in the LMM editor is kept in memory only, while the compiler reads the text contained in the equation file. Saving the text in the equation file you **need to not change the file name**, otherwise the compiler will keep on using the old equation file (if you don't save the file LMM reminds you to do so). Saving the file means always to replace the former equation file, though the last version is renamed with the extension **.bak**.

LMM - Compiling

Now we need to compile the equation file together with the result of the Lsd source code. This process entails many complicated commands. LMM takes care of this by simply using the menu **Model/Run**.

Writing code (and therefore also simulation models) means to spend 90% of time to fix errors. Here is the first possibility of errors: a wrong command, for example forgetting the semicolon at the end of the lines when requested, prevents the compiler to understand the code, and the compilation process fails. In this case, LMM generates a new window with approximate indications on the type and location of the error.

LMM - Compilation Errors

As example, suppose that the equation contains this mistaken line (missing semicolon in the end):

```
v[1]=V("m")
```

Suppose that this is the line number 11 (you can see the line and column number of the cursor in the text reading the top left part of LMM). The compilation error message will read:

```
fun_s0.cpp: In method 'double variable::fun(object *)':  
fun_s0.cpp:13: parse error before '='
```

LMM - Compilation Errors

Errors create always confusion for the compiler, and so the messages concerning them are best guesses. In our example the message says that the compiler realised that there was an error when reading line 13 of the file `fun_s0.cpp`. In general, messages indicate the exact faulty line or one immediately after the error.

It is also frequent the case of false errors signaled only because of an original (actual) error. For this reason, even when there are many errors reported, you should fix only the first one, and then trying to recompile. At least some of the other (apparent) errors will not be signaled.

LMM - Compiling

IMPORTANT: Compiling means creating a new file, in our case `lsd_gnu.exe`. MS Windows does not permit to overwrite the file of a program already running. If you are running a Lsd model program and try to compile a new version with a different equation file the compilation fails even if there are no errors in the code, simply because the operative system cannot create the new file with the program.

In these cases (*very* frequent) you need to close the Lsd model program and recompile the model from LMM with **Model/Run**.

Lsd Simulation Runs

We have now our new Lsd model program embedding the equation of the model, and we have the structure of the model (i.e. the object, variable and parameter labels) prepared in the previous session. In order to run a simulation we need to:

- Assign initial values to, both, the number of copies for the object, the parameter(s) and the lagged variables;
- Assign the simulation settings, like the number of time step to execute and marking the simulated series we are interested in observing.

After these operations we will be able to run a simulation and to analyse its results.

Initialize a model - Number of Objects

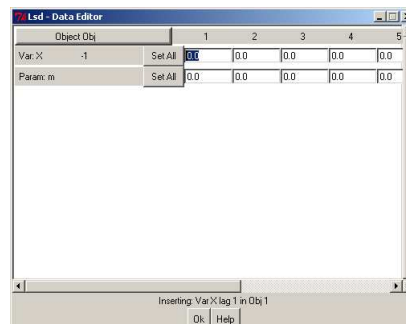
To set the number of objects in a model choose menu **Data/Set num. objects/All objects**. The new window will allow to set all the number of copies for each object.

The present model is extremely simple, having only one single type of object, and therefore we need to determine only the number of `Obj`'s. Working on more complicated models this interface permits an extremely flexible interaction.

Click on the button `Obj` and insert 10 in the resulting window. Confirm and exit the number of objects window.

Initialize a model - Initial values

We have now 10 copies of `Obj`, each containing a copy of `X` and of `m`. We need to define the values of these values in order to start the simulation. In the Lsd Browser choose menu **Data/Init.Values**. The resulting window presents an entry cell for each element in each copy of the object.

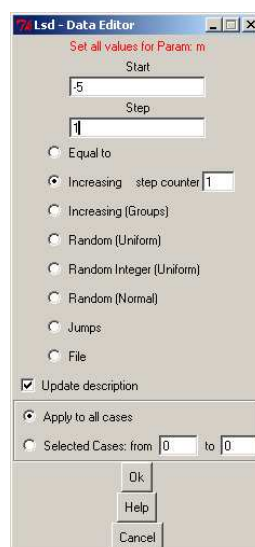


Initialize a model - Initial values

There are 10 columns, one for each copy of **Obj**, and 2 rows, one for the lagged value of variable **X** and one for the parameter **m**. Users can enter values directly in the cells. However, each row contains a button **Set All** allowing for the initialization of the whole sequence of values. This window allows many forms of initializations, based on the on the most frequently used initialization functions, like equal values, increasing values, random values etc.

Use **Set All** to set all values of **X** equal to 10, and to set the ten value of **m** to increasing values starting from 5 and adding 1 to each copy (see the **Set All** window below).

Initialize a model - Initial values



Simulation Settings

There are few settings we still need to set in order to control a simulation. Choose menu **Run/Sim.Settings** and set the values as in the following.



These settings tell Lsd to run a single simulation; use a given set of pseudo-random number sequence (if necessary); to span the simulation for 10 time steps; and, finally, that no interruption of simulation for debugging purposes is requested.

Saving simulation data

In order to maximize the computational efficiency a simulation run in Lsd discard immediately any non necessary data from memory. In our case, we are interested in observing the data produced for the variables **X**, so we need to tell explicitly the system to store such values for post-simulation analysis.

Double-click on the label for the variable **X** and mark the option **Save**.

Running a simulation

To run a simulation choose menu **Run/Run**. The system will ask for a confirmation recalling that no data will be permanently saved, but only store in memory, and that the model configuration (i.e. initial values, simulation settings etc.) will be written on the configuration file before running the simulation.

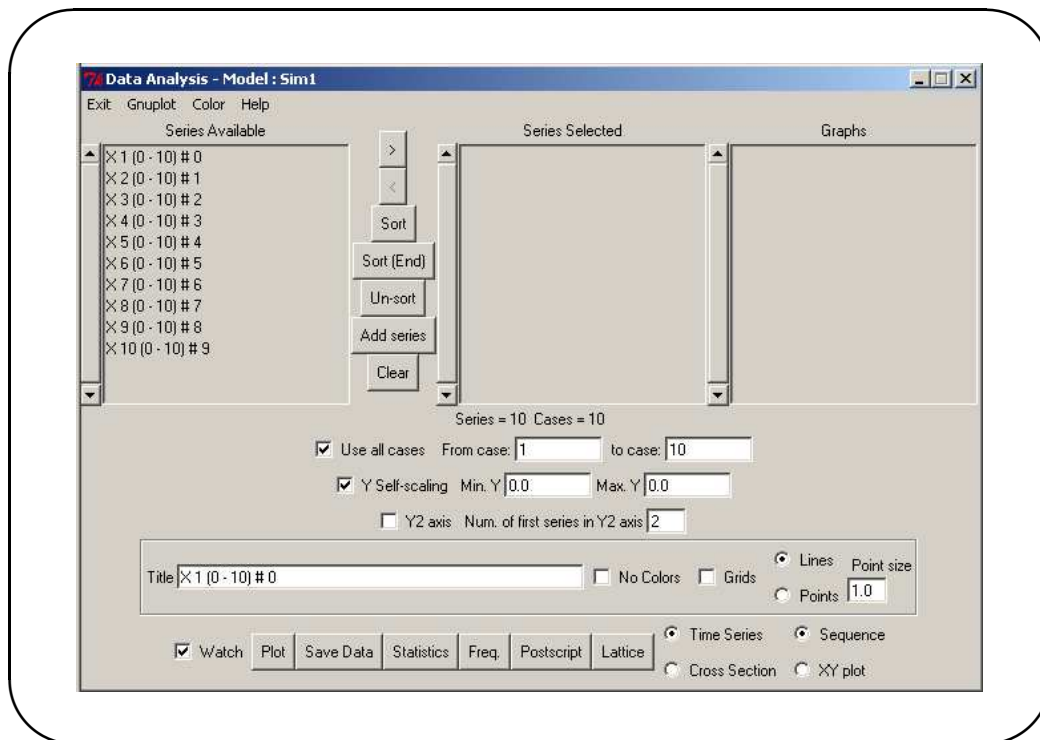
Any simulation run can always be replicated by re-loading the model configuration used to produce it. Files for model configurations that are not renamed, as in our case, are overwritten with the new configuration.

During a simulation run the system writes in the **Log** window the step executed. Optionally, it is possible to produce an approximated graph of the values for some variables, or having the system running without any output to increase the speed.

Analysis of Results

After a simulation run the system re-generate the Lsd Browser windows. However, the model loaded in the Lsd model program does not correspond to a model configuration, but to the last time step of the simulation just run. Moreover, the system has stored in memory the sequences for the values saved, in our example variables **X**.

To analyse the results choose menu **Data/Analysis Results**. The system will ask whether you want to analyse data from the simulation just run or from data previously saved. We have no simulation results saved, and are interested in analysing the simulation data, therefore choose the **Simulation** option. The Lsd Browser is replaced by the Lsd interface for data analysis.



Marco Valente

Università dell'Aquila

Analysis of Results

This window shows the series available, in our case the 10 series for variables **X**. Each series is indicated with a label including: the element label (**X**); an index of the object containing it; the life time of the data collected (**(0-10)**, since we provided the time 0 value); a (generally useless) indicator of the series.

The analysis of the results consists in selecting one or more of the series available and moving them in the **Series Selected** listbox. Pressing one of the buttons you can produce several types of graphs, statistics, or export them in a common formats. The options determine the results of the commands.

Any graph produced will be listed in the **Graph** listbox, where the user can click on it to bring it in the foreground.

Marco Valente

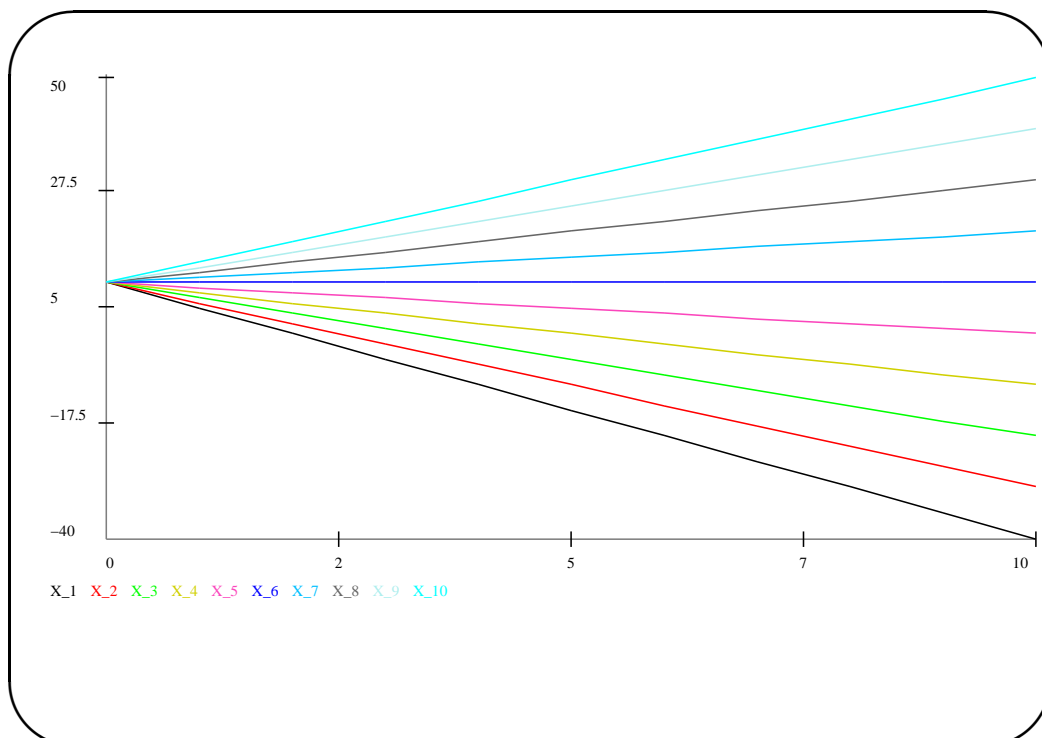
Università dell'Aquila

Analysis of Results

The Lsd Analysis of Result window is endowed with several functions helping in the most frequently needed actions. For example, click with the right button of the mouse over one of the labels in the **Series Available** listbox and press **Ok**. All the **X** labels will be copied in the **Series Selected** list. Press the button **Plot**. The default options are set for producing time series graphs with automatically computed axis and using colored lines. The following graph will be produced.

Marco Valente

Università dell'Aquila



Marco Valente

Università dell'Aquila

Analysis of Results

The graphs produced by Lsd can be edited in several ways, adding new labels, editing existing ones or changing colors. Modifications of the axis scales or of the series used require the production of a new graph. Graphs can be exported as PostScript files to be included in a text document.

The Analysis of Result module allows several operations on the simulation data. For advanced statistical analysis it is possible to export selected series with several text formats (e.g. tab-delimited or fixed-length columns). Moreover, the MS Windows versions include the Gnuplot plotting package (the Lsd Unix version expects the package to be already installed; Gnuplot is required for the scatter plot graphs).

Help yourself

All the Lsd commands are issued with pretty intuitive graphical interfaces whose use should be immediately clear. Moreover, each window has a button (or menu item) for help. Use these help pages for instructions on how to operate the windows. Help pages are linked to other related help pages, forming a hyper-textual manual. A possible initial point, for readers interested in a general overview of the help manual, is the table-of-content page, or the list of the main Lsd windows.

LMM is also endowed with its own Help page, including an exhaustive list of the Lsd functions available for writing equations.

Summary

What we have done in this introductory lecture is:

1. Design a model on paper;
2. Written the code implementing the equation (and fixed bugs in the code?);
3. Defined the model structure and initialization;
4. Run the simulation;
5. Analysed the results.

Conclusions

Any simulation model requires the same steps we have seen above. To avoid “complexity crises” it is good practice to develop a model in a modular way. Firstly, implement only one or few equations’ code, and insert the required model configuration. Clean-up the code from errors, and then move on increasing the complexity of the equations, turning a parameter in a variable (and adding the corresponding equation) and add new variables.

Exercises

- Test the *Simulation 0* model with different initializations, number of `Obj` and number of steps. Check that it does what it is supposed to do.
- Modify the model changing the equation for X as:
$$X_t = m * X_{t-1} * (1 - X_{t-1}).$$
 Test the model for initial values of X_0 between 0 and 1, and for values of m between 1 and 4. Pay particular attention to the values of m around 3, 3.45, 3.547, 3.84.