Treinamento LSD

Aulas 2-10 – 2ª Parte

Baseado nos slides de Marco Valente

Agenda

- Aula 2: Linear model and LSD basics
- Aula 3: Random walk model and LSD model structure
- Aula 4: Logistic model and chaotic behavior
- Aula 5: Replicator dynamics model
- Aula 6: Extended replicator dynamics model
- Aula 7: Network externalities model – Part 1
- Aula 8: Network externalities model – Part 2
- Aula 9: Consumers' model – Part 1
- Aula 10: Consumers' model – Part 2

Lecture objective

In this lecture we will implement a model with an economic interpretation.

As we will see, though the model is extremely simple it can generate results not immediately intuitive.

We will use LSD to **implement** the model, **analyse** the results, **debug** the model, and address a basic **methodological** issue with simulation models.

Replicator Dynamics Model

Let's consider a model made of a market and several firms. Firms are assigned a constant level of quality. Firms' sales are computed according to the equation:

$$Sales_t = Sales_{t-1}\left[1 + a\left(\frac{Quality - AvQuality_t}{AvQuality_t}\right)\right]$$

That is, sales change of $\frac{a * Sales_{t-1}}{100}$ for each percentage of difference between the firms' own quality and the market average quality.

Create a new model in LMM, call it **RD**. Insert the code for this equation.

Replicator Dynamics Equation

```
EQUATION("Sales")
/*
Level of sales, computed as the past sales plus a share of the
difference of quality in respect of the average quality
*/
v[0]=V("Quality");
v[1]=V("a");
v[2]=VL("Sales",1);
v[3]=V("AvQuality");
v[4]=v[2]*(1+v[1]*(v[0]-v[3])/v[3]);
RESULT(v[4] )
```

Replicator Dynamics Model

The average quality is the *weighted* average of firms' qualities, where sales are used as weights.

This makes sure that the contribution of each firm to the average is proportional to its relevance in the market: firms with higher level of sales are more relevant in defining the quality of goods in a market.

Replicator Dynamics Model

If we have N firms on which to compute the average of the quality weighted with the sales, the equation is:

$$AvQuality_t = \frac{\sum_{i=1}^{N} Sales_t^i \times Quality^i}{\sum_{i=1}^{N} Sales_t^i}$$

Average quality: the use of cycles

In computational terms the average is implemented with the following steps:

1. Set to 0 two temporary variables where to cumulate values for the numerator and denominator, call them *Num* and *Denum*;
2. go in the first object **Firm**;
3. compute the values of **Quality** and and **Sales** for the object;
4. increase the numerator by **Quality** x **Sales**;
5. increase the denominator by **Sales**;
6. move to the next object **Firm**. If there is one goto 3.
7. Otherwise (no more **Firm**), return the ratio *Num*:*Denum*.

Use of cycles

The Lsd code equivalent is the following:

```
EQUATION("AvQuality")
/*
Average quality, weighted with the value of sales
*/
v[0]=0;
v[1]=0;
CYCLE(cur, "Firm")
 {
  v[2]=VS(cur,"Quality");
  v[3]=VS(cur,"Sales");
  v[0]=v[0]+v[2]*v[3];
  v[1]=v[1]+v[3];
 }
RESULT(v[0]/v[1] )
```

Use of cycles

Let's see in detail the code for the equation.

```
...
v[0]=0;
v[1]=0;
...
```

These two lines set to 0 two local variables. These variables are used as containers for the cumulated values of **Sales** x **Quality** (numerator) and **Sales** (denominator).
They need to be explicitly set to 0 because C++ does not initialize variables.

Use of cycles

```
CYCLE(cur, "Firm")
 {
  v[2]=VS(cur,"Quality");
  v[3]=VS(cur,"Sales");
  ...
 }
```

This code implements a cycle. That is, the lines within the cycle are repeated again and again as many times as many copies of object **Firm** are contained into the object containing the computing variable, in this case **AvQuality**.

At each repetition, the local *pointer* **cur** refers to a different copy of a **Firm**; **cur** is the similar to local variables `v[.]`, but, instead of containing numerical values, pointers deal with object.

The LSD command `VS(cur, "VarLabel")` is like `V("VarLabel")` but instead of searching the element **VarLabel** in the object where the equation is executed (**AvQuality** is contained in **Market**) it searches in the element pointed to by `cur`, which is an object of type **Firm**.

Replicator Dynamics Model: cycle

In LSD it is possible to generate many different computational structure acting on the position of the variables within the structure of objects.

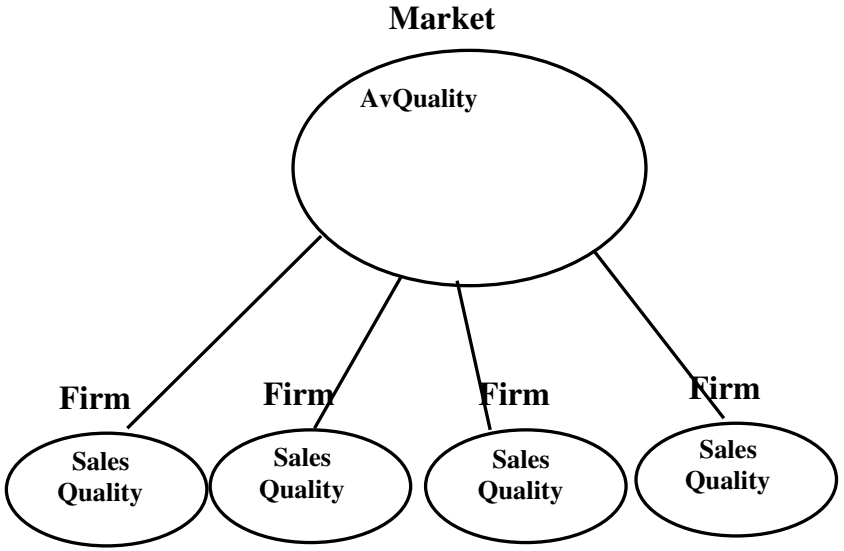
In general, objects “up” in the hierarchy contains elements affecting, or affected by, many copies of other elements contained in the entities “down” in the hierarchy.

LSD the code automatically executes the most sensible operation depending on the position of the object containing the variable whose equation is executed.

Replicator Dynamics Model: cycle

In our case the model is composed by an object, **Market**, containing several copies of object **Firm**. The equation for **AvQuality** is to be located in **Market**, since it is collective property of all firms, not a specific one.

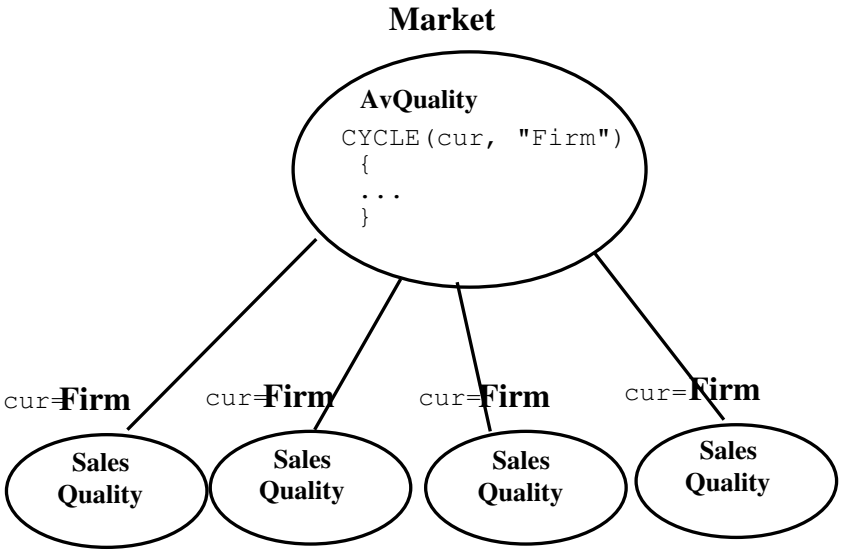
It its code requested generically **Quality**, as in `V("Quality")`, without pointing to a specific copy of an (using object `cur`), the system would be erroneously return always the value read from the very first copy in the set of objects **Firm**.



Replicator Dynamics Model: cycle

The solution is to use the expression `VS(cur,"Quality")`, which forces the code to search **Quality** not in **Market** but in the object pointed to by `cur`.

Pointers are temporary variables where modellers store objects instead of numbers. When the command line `CYCLE(cur,"Firm")` is used, the system repeats the lines of code that follow as many times as many objects **Firm** are found in **Market**. During each repetition `cur` will contain a different copy of **Firm**.



Replicator Dynamics Model: cycle

```
CYCLE(cur, "Firm")
 {
  v[2]=VS(cur,"Quality");
  v[3]=VS(cur,"Sales");
  v[0]=v[0]+v[2]*v[3];
  v[1]=v[1]+v[3];
 }
RESULT(v[0]/v[1] )
```

The lines for `v[0]` and `v[1]` in the cycles computes the values for the weighted average, using the values of each firm `v[2]` and `v[3]`.

Use of cycles

Note that the LSD command `CYCLE(cur, "ObjLabel")` works correctly only if object executing the equation is *above* (i.e., contains) the group of objects to scan with the cycle.

The command works as follows. Firstly, it searches the group of objects **ObjLabel** among the objects contained within the object whose variable is updated. When it is found, the first copy of the group is assigned to the pointer `cur`, and the body of the cycle is executed. When finished, the cycle controls whether there is another copy of object next to the one just used. If there is another copy, this is assigned to `cur` and the body is repeated again.

IMPORTANT: control always that the variable whose equation contains a cycle is located in the correct object.

Model Compilation and Configuration

We can now compile the model: menu **Model/Run** will create the Lsd model program for this model, or give messages on the grammar errors preventing the compilation.

If this is case, check the number on the error message next to the file name. It indicates the line number where the error has been detected, so you can easily fix equation checking the code immediately before the location.

In case of success, create the model structure using the commands in menu **Model** to create the variables, parameters and objects necessary for the model. Beware to have the LSD Browser showing the correct object before including new items.

Replicator Dynamics Model

The model must be composed by:

- Object **Market** containing variable **AvQuality** (0 lags), parameter **a** and object **Firm**.
- Object **Firm** containing variable **Sales** (1 lags) and parameter **Quality**.

Finally, generate 10 copies of object **Firm** initializing their content as follows:

- assign to each **Quality** an increasing value from 10 to 19;
- assign to each **Sales[-1]** (lagged value of **Sales**) the same value 1000.
- assign value 0.2 to parameter **a**;

We can now run the model, but...

Dead lock errors

If no other error appeared, the model program shows a small window signaling the occurrence of a fatal error preventing the continuation of the simulation. As in any instance of error the **Log** window contains extensive information on the reason for the problem. In this case it will read:

Dead lock errors

```
Dead lock! Variable:
AvQuality
requested by Object:
Firm

Fatal error detected at time 1.
Offending code contained in the equation for Variab
... (deleted text) ...
Level Variable Label
2 Sales
1 AvQuality
0 Lsd Simulation Manager
```

Dead-lock errors

The dead-lock error is an example of a logical error. The two equations have a perfectly legal implementation, *considered independently*. But they cannot work in the same model.

To see why let's consider again the equations.

$$AvQuality_t = \frac{\sum_{i=1}^{N} Sales_t^i \times Quality^i}{\sum_{i=1}^{N} Sales_t^i} \tag{1}$$

$$Sales_t = Sales_{t-1} \left[1 + a \left(\frac{Quality - AvQuality_t}{AvQuality_t} \right) \right] \tag{2}$$

Dead-lock errors

Suppose that at the beginning of the simulation 1 the model starts to compute equation (1). In order to execute the code for the equation it needs the values of **Sales** and **Quality** for each firm.

The system can access the value of **Quality** immediately, since it is a parameter. But **Sales** is not computed yet, so the system cannot access its value.

In this case the system interrupts the computation of **AvQuality** and starts the execution of the equation (2) in order to get the value of **Sales** from the first **Firm**.

This is the way LSD determines automatically the priority of execution for the equations within a time step.

Dead-lock errors

The problem is that, in order to compute equation (2), the system needs, among other values, also the value of **AvQuality** at time $t = 1$. But this is not available, because its execution has been interrupted.

Even if we start from equation (2) we have the same problem: neither of the two equations can be computed before the other!

This is essentially a chicken-egg problem rooted not in the code of a single equation, but in the logic of the model itself, which is a limitation, and a strength, of computational models.

Dead-lock errors

Dead-locks can appear only in computational models, not in mathematical ones. In mathematics it is perfectly normal to have a system of equations like:

$$X_t = 2 + 3Y_t \tag{1}$$

$$Y_t = 4 - 2X_t \tag{2}$$

The mathematical meaning of the system is: *find the values of X_t and Y_t such that the conditions (1) and (2) are satisfied*.

In computational terms, however, such system is “illegal”, in the sense that generates an impossible computational structure. The computer reads the equations as: *to compute X first you need to compute Y. To compute Y first you need to compute X*. Locked.

Mathematics vs. Computers

Mathematical models and simulation models are different because the second unfold irreversibly through time, while mathematical models express a-temporal relations.

Which modeling style is better in Economics? Depends on the phenomenon you want to represent.

- Mathematical models represent laws deemed universal. For example, a demand curve $P = \frac{1}{Q}$ (or $Q = \frac{1}{P}$?)
- Computational models represent dynamical phenomena depending on many local conditions. For example, the development of a market.

Learning by coding

The logic of time consistency, necessary for computation models, requires the modeler to consider the **timing** of events, besides their computational structure.

In our example the two equations are computationally correct to implement the variables' content, but the model cannot work because timing we involuntarily imposed is inconsistent.

Many economic phenomena depend crucially on time, and the need to implement a model in computational format forces modelers to consider this aspect.

Solving dead-lock errors

How do we solve a dead-lock error? We need to allow the system to find a viable priority, completing first one equation and then the other.

We have two options: first equation (1) and then equation (2), or the other way round. Let's have the model to compute first **AvQuality**. That is, our model becomes:

$$AvQuality_t = \frac{\sum_{i=1}^{N} Sales^i_{\mathbf{t\text{-}1}} \times Quality^i}{\sum_{i=1}^{N} Sales^i_{\mathbf{t\text{-}1}}} \tag{1}$$

$$Sales_t = Sales_{t-1} \left[1 + a \left(\frac{Quality - AvQuality_t}{AvQuality_t} \right) \right] \tag{2}$$

In this way, equation (1) can be computed because it does not need to compute $\mathbf{Sales}_t$. When (1) computes $\mathbf{AvQuality}_t$ then the system can compute (2).

Solving dead-lock errors

Notice that in LSD the modeler does not need to produce the whole list of variables to be updated, which can be annoying in case of large models.

Rather, modelers simply provide local indications, telling the system that one value may be computed before of another.

The system automatically generates one of the many possible sequences of variables' computation guaranteeing that the constraints indicated are respected.

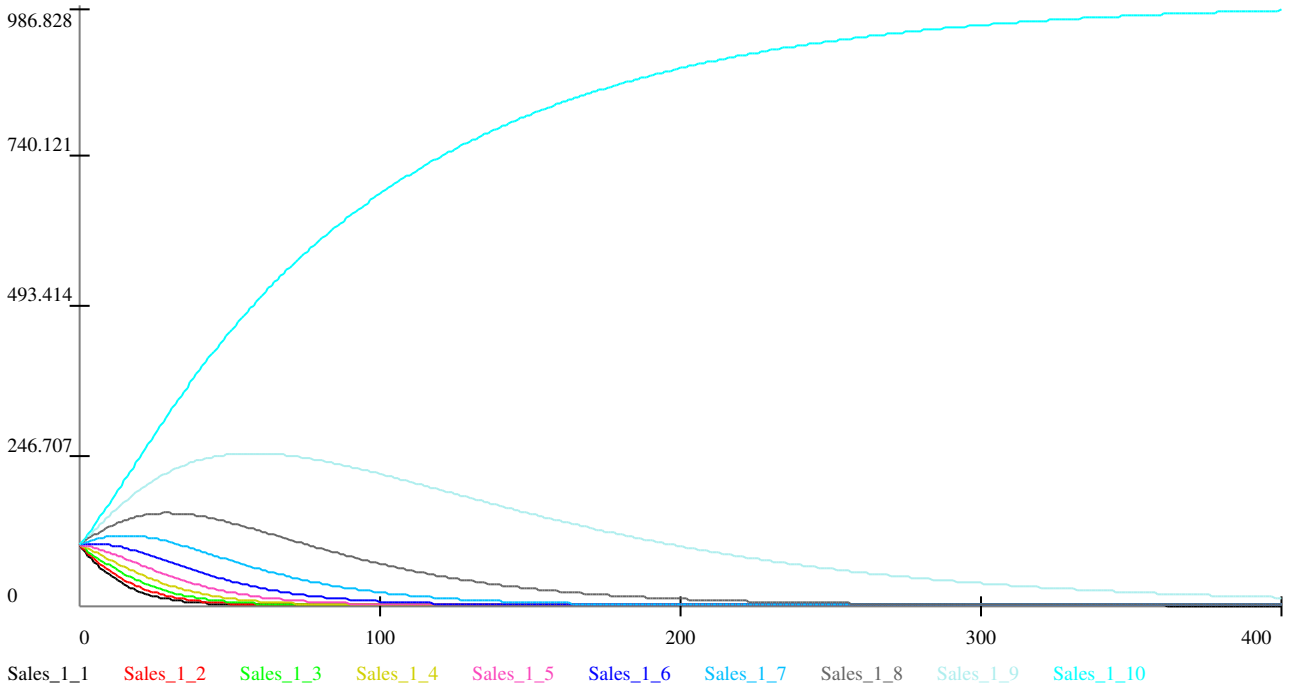
Fixing the dead-lock error

The equation for average quality becomes:

```
EQUATION("AvQuality")
/*
Average quality, weighted with the value of sales
*/
v[0]=0;
v[1]=0;
CYCLE(cur, "Firm")
 {
  v[2]=VS(cur,"Quality");
  v[3]=VLS(cur,"Sales",1); //use lagged values
  v[0]=v[0]+v[2]*v[3];
  v[1]=v[1]+v[3];
 }
RESULT(v[0]/v[1] )
```

Running the model

Now the model works. The 10 firms with qualities ranging from 10 to 19 and initial sales at 100 produce the following result.



Interpretation

Simulation results gives the possibility to reach conclusions difficult to appreciate without executing simulations.

The model can be interpreted as expressing the fact that high-quality firms eventually gain shares at the expenses of low quality ones.

Note that this is *not* a model of competition, at least in the long term. It is a model representing the path leading to a monopoly, since it is impossible to prevent the highest quality firm to dominate the market.

Question

Moreover, even extremely simple models, such as this one, have the potential to generate puzzles, which the very nature of the model as computer program can help to solve.

QUESTION: The model states, essentially, that sales' dynamics depend quality. However some series, such as the 9^{th} show initially an increase in sales followed by a decrement. How is this possible, given that quality is a constant parameter?

Debugging Lsd models

LSD allows to inspect each and every aspect of a model during a simulation run. This is performed using a module called **Lsd Debugger**. This module starts automatically in case of errors. Moreover, users can activate it under two, joint, conditions:

1. One variable marked with the option **Debug** concluded its computation.
2. The simulation is executed in *debug* mode.

Debugging Lsd models

The debug mode is activated during a simulation run by one of several possibilities:

1. Clicking on the button **Debug** in the **Log** window.
2. Pre-setting a specific time step before a simulation run (among **Simulation Settings**).
3. Pre-setting a future time step during a debugging session.
4. Specify a condition on the value of an element, e.g. when a variable is negative.
5. Writing in the code of one equation a command triggering the debugging session (command `INTERACT(...)`).

Debugging Lsd models

During a debugging session the modeler can:

- Analyse the results available.
- Inspect the content of every copy of every object, reading the current values of the elements.
- Proceed the simulation run step-by-step, with each step being the completed computation of a variable with the flag **Debug** set on.
- Set/remove settings for debugging (flag, conditions, etc.).
- Modify the current values of the elements.
- Add/remove objects (DANGEROUS).
- Force the computation of equations.
- ...

Interpretation and Question

As an exercise tries to use the LSD debugger to track the instant in which the sales of the 9^{th} firms changes slope, so as to clarify the motivations for the puzzle.

After doing so, use the *Analysis of Results* module to present the solution in a fool-proof way