A Gentle Introduction to TerraME

TIAGO CARNEIRO[1], GILBERTO CÂMARA[2]

[1]Computer Science Department, Federal University of Ouro Preto,

Campus da UFOP, Ouro Preto, Brazil

[2]Image Processing Division, National Institute for Space Research

Av dos Astronautas, 1758, 12227-001 São José dos Campos, Brazil

{tiago, gilberto}@dpi.inpe.br

1 Introduction

TerraME is a development environment for spatial dynamical modelling that supports the concepts of nested cellular automata (nested-CA) . TerraME uses a spatial database for data storage and retrieval. A *spatial dynamic model* is a model whose locations are independent variables. The outcomes of these models are maps that depict the spatial distribution of a pattern or of a continuous variable. TerraME enables simulation in two-dimensional cellular spaces. Among the typical applications of TerraME are land change and hydrological models.

This tutorial provides an introduction to the basic features of TerraME. For a full description, see . The tutorial has four parts. In section 2, we present the TerraME architecture. In section 3, we present the basic commands of the TerraME programming language. In section 4, we show an example of using TerraME for hydrological modelling. In section 5, we show an example of land change modelling. Before using this tutorial, the reader should first install TerraME. Instructions for installation are in the Appendix to this report. Readers interested in an introduction to the principles of modelling should refer to or . Useful discussions on spatial modelling include .

2 The TerraME Environment

The key part of the TerraME development environment is the *TerraME interpreter,* as shown in Figure 1. It reads a program written in the TerraME modelling language (a LUA language extension), interprets the source code, and calls functions in the TerraME framework. This framework is a set of modules written in C++ that provides functions and classes for spatial dynamical modelling. It also links to a *TerraLib* spatial database. The *TerraView* application displays the results of the simulation.

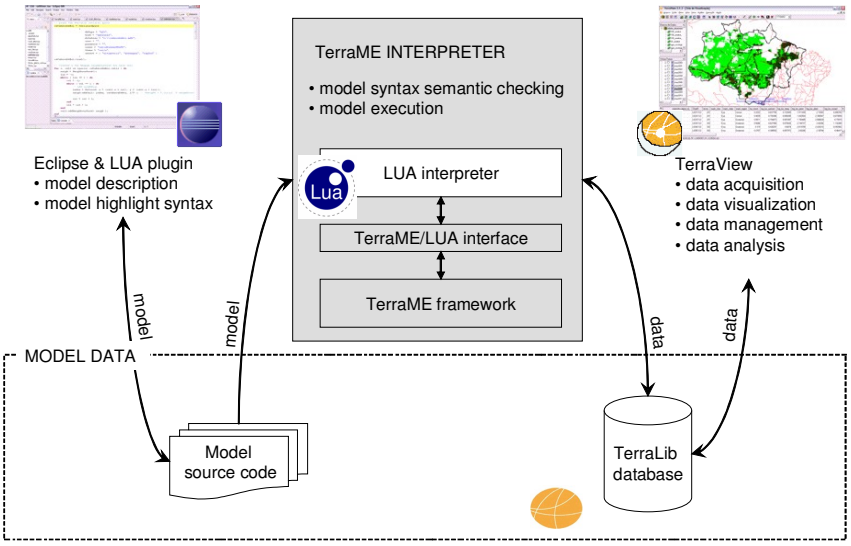


Figure 1 – The TerraME development environment

The TerraME environment consists of the following parts:

- The TerraME interpreter, which executes the model code.
- *TerraLib,* a GIS library for spatial database management .
- *TerraView* is a *TerraLib* application used for vector and raster data acquisition, visualisation, and analysis.
- The *LUA* programming language serves as base for the TerraME modelling language.

The *Eclipse* software development platform is the model development environment, and uses the LUA plugin. Figure 2 shows the architecture of *TerraME* architecture. Lower layers provide basic services over which upper layer services are implemented.

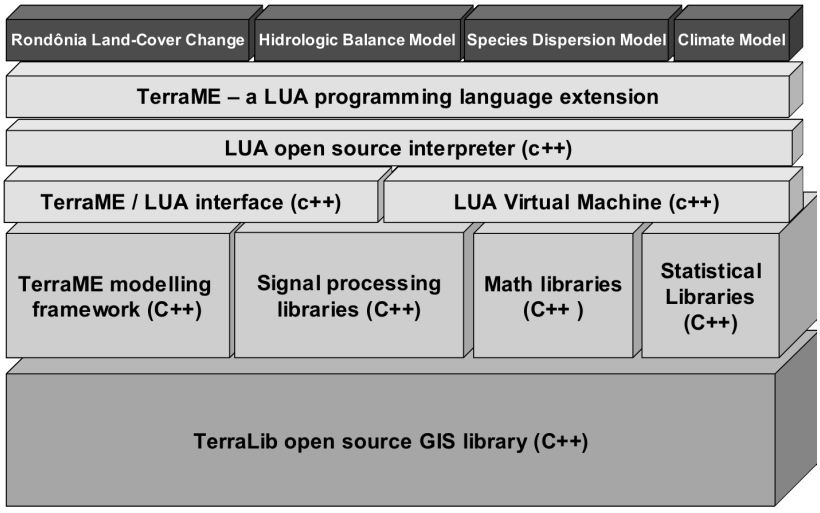


Figura 2 – TerraME architecture

In the first layer, *TerraLib* offers typical GIS spatial data management and analysis services, and extra functions for temporal data handling. The *TerraME framework* provides the simulation engine and the calibration and validation services. It is an open source ANSI C++ implementation of the nested-CA model, portable for Windows and Unix-like operating systems. This framework can be used directly for model development. Since developing models in C++ can be a challenge for non-programmers, *TerraME* provides a high-level modelling language. The third layer of the architecture implements the *TerraME modelling language* interpreter and runtime environment. The TerraME/*LUA* interface extends *LUA* with new data types for spatial dynamic modelling and services for model simulation and evaluation. Using the *LUA* library API, it exports the *TerraME framework* API to the *LUA* interpreter, so it recognizes the *TerraME* types. If needed, other C or C++ applications (such as statistical libraries) can have their APIs exported to the LUA interpreter and integrated in the architecture. The last layer, called application layer, includes the end-user models.

3 The TerraME Modelling Language: Basic Commands

This section presents the basic *TerraME Modelling Language* mechanisms for multiple scale spatial dynamic model representation and simulation.

3.1 TerraME as a LUA Extension

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Being an extension language, Lua has no notion of a "main" program: it only works embedded in a host client. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. By using C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework . The *TerraME Modelling Language* is a *LUA Programming Language* extension. It uses the LUA extensibility mechanisms to include new data types and functions.

LUA is a dynamically typed language: variables do not have types; only values do. There are no type definitions. The basic value types are *number* (double) and *string*. The value *nil* is different from any other value in the language and has the type *nil*. Functions in *LUA* are first-class values. That is, a function definition creates a value of type *function* that can be stored in variables, passed as arguments to other *functions* and returned as results. The only structured data type is *table*. It implements associative arrays, that is, arrays that can be indexed not only with integers, but with *string*, *double*, *table*, or *function* values. For *table* indexing, both *table.name* and *table["name"]* are acceptable. *Tables* can be used to implement records, arrays, and recursive data types.

They also provide some object oriented facilities, such as methods with dynamic dispatching .

```
loc = { cover = "forest", distRoad = 0.3, distUrban = 2 };
loc.desfPot = loc.distRoad + loc.distUrban;
...
loc.reset = function( self )
              self.cover = "";
          self.distRoad = 0.0;
          self.distUrban = 0.0;
            end
```

Figure 3 – The use of associative table and function values in *LUA*.

Figure 3 shows the use of *table* and *function* values. The code creates a *table* with three attributes (land cover, road distance, and urban centre distance) and stores it the variable *loc*. It calculates a new attribute and adds it to *loc* (deforestation potential is the sum of the road and urban center distances). Finally, it creates a second attribute called *reset* and adds it to table *loc*. It is as a *function* that receives the *table* as parameter. This is indicated by the keyword *self*.

LUA has a powerful syntactical tool, called *constructor*. When the modeller writes *name{…}*, the *LUA interpreter* replaces it by *name({… })*, passing the table *{…}* as a parameter to the function *name( )*. This function typically initializes, checks properties values and adds auxiliary data structure or methods . In figure 5, it constructs the type *MyLoc*. When the table *L* is instantiated, the constructor initializes the attribute *desfPot*.

```
function MyLoc( loc )
      loc.desfPot = loc.distRoad + loc.distUrban;
end
l = MyLoc{cover = "forest", distRoad = 0.3, distUrban = 2 };
```

Figure 4 – The use of the *constructor* in *LUA*.

To build spatial dynamic models, TerraME includes new value types in *LUA* using the constructor mechanism. These values are: *CellularSpace*, *Cell, Neighbourhood, Scale, SpatialIterator*, *GlobalAutomaton*, *LocalAutomaton*, *ControlMode*, *JumpCondition*, *FlowCondition, Timer, Event* and *Message*. We describe the first three types and its operations in what follows. A description of the other types is available in .

3.2 The CellularSpace

A *CellularSpace* is a multivalued set of *Cells*. It consists of a geographical area of interest, divided into a regular grid. Each cell in the grid has one or more attributes. CellularSpaces are stored and retrieved from a *TerraLib* database, so the modeller should specify the properties of the *CellularSpace* before using it, as shown in Figure 5.

```
-- Loads a TerraLib cellular space
csCabecaDeBoi = CellularSpace {
      dbType = "MySQL",
      host = "localhost",
      database = "CabecaDeBoi ",
      user = "",
      password = "",
      layer = "cells90x90",
      theme = "cells",
      select = { "altitude", "infCap" }
      where = "mask <> 'noData'";
}
```

Figure 5 – Defining a CellularSpace in TerraME.

The *host* and *database* values indicate where the input data is stored. The *dbType* value identifies the database management system (*MySQL*, *PostgreSQL*, etc). The *layer* and *theme* valeus are the names of the *TerraLib* database *layer* and *theme* used as input data. A *layer* is a container of data in TerraLib. A *theme* is a set of spatial objects from that *layer*, selected by a restriction. Selection uses a database query over attribute values, spatial relations, and temporal relations. The *select* property contains the names of the cell attributes loaded into the model from the input data set. The property *where* filters the data, as in SQL statements. The *select* and *where* properties are optional.

In Figure 5, the code creates the *CellularSpace* "csCabecaDeBoi" from the "cells" *theme*, part of the "cells90x90" *layer* of the "CabecaDeBoi" database. For each cell, it loads two attributes: elevation (altitude) and infiltration capacity (infCap). It loads in the *CellularSpace* only cells whose "mask" attribute value is different from "noData".

A *CellularSpace* has a special attribute called *cells*. It is a one-dimensional *table* of references for each *Cell* in the *CellularSpace*. The first *Cell* index is 1. Figure 6 shows how to refer to the i-th *Cell* from a *CellularSpace*.

```
-- c is the seventh cell in the cellular space
```

```
c = csCabecaDeBoi.cells[ 7 ];
-- Updating the attribute “infcap” from the seventh cell
c.infcap = 0;
csCabecaDeBoi.cells[7].infCap = 0
```

Figure 6 – Referencing cells.

3.3 Database management for cell spaces

A *TerraME CellularSpace* provides three functions for database management. The load() function loads the cell attributes from the spatial database. The loadNeighbourhood() function loads a neighbourhood structure. The save() function stores the desired cell attribute values in the associated *TerraLib* database. The Figure 7 shows how these functions are invoked for the csCabecaDeBoi *CellularSpace*.

```
csCabecaDeBoi:load();
csCabecaDeBoi:loadNeighbourhood("Moore");
......
for time = 1, 10,1 do
   csCabecaDeBoi:save(time, “sim", {"water"});
end
```

Figure 7– Loading and saving cellular spaces in *TerraME*.

The load() function simply loads a previously defined cellular space in memory (see Figure 5 for an example of a cellular space). The loadNeighbourhood (name) loads a neighbourhood structure defined by name. By default, TerraME provides a VonNeumann (2x2) and a Moore neighbourhood (3x3). The user can create her own neighbourhood structures using the TerraView application, including the a generalized proximity matrix, where each cell has a different neighbourhood.

The syntax of the save function is save (time, themeName, attrNameTable). The function uses the value time as the data timestamp. It stores data in the *TerraLib theme* called themeName. It also saves the cell attributes in the table attrNameTable. If the third value is empty or a *nil* value, all cell attributes will be saved. The save(...) function also creates a *view* named Result in the *TerraLib* database. It inserts in this *view* a *theme* containing the saved data. The name of the theme is the union of themeName + time. In code shown in Figure 7, the values of the attribute "water" of all cells from the *cellular space "*csCabecaDeBo*i"* are saved in the *themes*: "sim1", "sim2", "sim3", and so on.

3.4 The Cell type

A *Cell* represents a spatial location, its properties, and its nearness relationships. A *Cell* is a *table* that includes persistent and runtime attributes. The persistent attributes are loaded from and saved to the database. The runtime attributes exist only in memory during the model execution. A *Cell* value has two special attributes: latency and past.

The latency attribute registers the period of time since the last change in a cell attribute value. It is useful for rules that depend of how long the cell remains in a state. The past attribute is a copy of all cell attribute values in the instant of the last change. For example, Figure 8 shows the command "if the cell cover is *abandoned land* during 10 year then the cover transit to *secondary forest*". Figure 8 also shows a rule for simulating rain in a cell, which adds 2mm of water to the past amount of water.

```
if( cell.cover == "abandoned" and cell.latency >= 10 ) then cell.cover =
"secFor"; end
cell.water = cell.past.water + 2;
```

Figure 8 – The *latency* and *past* attributes.

3.5 Traversing a cell space

TerraME provides two ways for traversing a cellular space:

- A "for...end" statement: "for i, cell in pairs (csQ.cells) do...end". The i and cell variable in the statement are the *index* and the *value* of a cell inside the cells attribute from the cellular space csQ.
- A second-order function (a function that has a function as an argument): ForEachCell(cs, function()) applies the chosen function to each cell of the cellular space. This function enables using different rules in a cellular space.

Both choices appear in Figure 9. The cellular space csQ is a terrain area where there is constant rain (2 mm/hour) during 10 hours. At the end of each iteration, the cell space must be synchronized (this is explained in section 3.8).

```
for time = 1, 10, 1 do
      for i, cell in pairs( csQ.cells ) do
        cell.soilWater = cell.past.soilWater + 2;
      end
  ForEachCell(csQ, function(cell)
                cell.soilWater = cell.past.soilWater + 2;
                return true; end);
      csQ:synchronize();
end
```

Figure 9 – Traversal of a cell space

3.6 The Neighbourhood type

Each cell has one or more *Neighbourhoods* to represent proximity relations. A *Neighbourhood* is a set of pairs *(weight, cell)*, where *cell* is a neighbour *Cell* and *weight* is the strength of relationship. There are two ways of creating a neighbourhood in TerraME:

- By creating a Von Neumann (2x2) ou Moore (3x3) neighbourhood, using CreateVonNeumannNeighbourhood() or CreateMooreNeighbourhood();
- By loading an existing neighbourhood, using the loadNeighbourhood () function.

In the latter case, the neighbourhood is created from a generalized proximity matrix or GPM . TerraLib has facilities for creating these types of flexible neighbourhoods. Please refer to the TerraLib documentation for more details on GPMs. We can operate on the neighbors of each cell using the function ForEachNeighbour(cell, number, function()), as shown in Figure 10. ForEachNeighbour receives a function as parameter and traverses the i-th *Neighbourhood* of a *Cell* applying this function to all cells in it. For each cell, the method getNeighbourhood (i) gets its i-th *Neighbourhood*. The method getWeight() returns the intensity of the neighbourhood relationship between the cell and its current neighbour. In Figure 10, we show a simple example where, for each cell, we print the weight of all its neighbours.

```
csq:loadNeighbourhood("GPM");
ForEachCell(csQ,
    function (cell)
        ForEachNeighbour(cell, 0,
            function( cell, neigh)
              print(neigh:getWeight());
          end;
        ); -- for each neighbor
        return true;
         end;
     ); -- for each cell
```

Figure 11 – Traversing a neighbourhood

The following TerraME code is Conway's Game of Life. The game uses on a field of cells, each of which has eight neighbors. A cell is occupied or empty. The rules for deriving a generation from the previous one are these:

- If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbours, the organism dies (0, 1: of loneliness; 4 to 8: of overcrowding).
- If an occupied cell has two or three neighbours, the organism survives to the next generation.
- If an unoccupied cell has three occupied neighbours, it becomes occupied.

The code starts by creating a Moore neighbourhood. Then it iterates until a final time. At each iteration, it traverses the cell space. For each cell, it applies Conway's rules. Note that it uses the cell's *past* value as input. Then it updates the present value of the cell. Finally, the cell space is synchronized.

```
CreateMooreNeighbourhood(csQ);
csQ:synchronize();
for time = 1, FINAL_TIME, 1 do
    ForEachCell(csQ,
            function(cell)
        count = 0;
        ForEachNeighbor(cell, 0,
              function(cell, neigh)
                 if (neigh.value == 1) then
                count = count + 1;
              end
              end;
         ); -- for each neighbor
        -- apply Conway's rules
        if (cell.past.value == 1) and
            ((count < 2) or (count > 3))  then
              cell.value = 0 --- cell dies
        end
         if (cell.past.value == 0) and (count == 3) then
           cell.value = 1 --- cell lives
        end
       return true; end;
      ); -- for each cell
      csQ:synchronize();
end
```

Figure 12 – Conway's Game of Life

3.7 Synchronizing a cell space

TerraME keeps two copies of a cellular space in memory: one stores the past values of the cell attributes, and another stores the current (present) values of the cell attributes. The model equations must read (the right side of the equation rules) the attribute values from the past copy, and must write (the left side of the equation rules) the attributes values to the present copy of the cellular space. At the correct moment, it will be necessary to synchronize the two copies of the cellular space, copying the current attribute values to the past copy of the cellular space. Figure 13 shows how synchronization works.

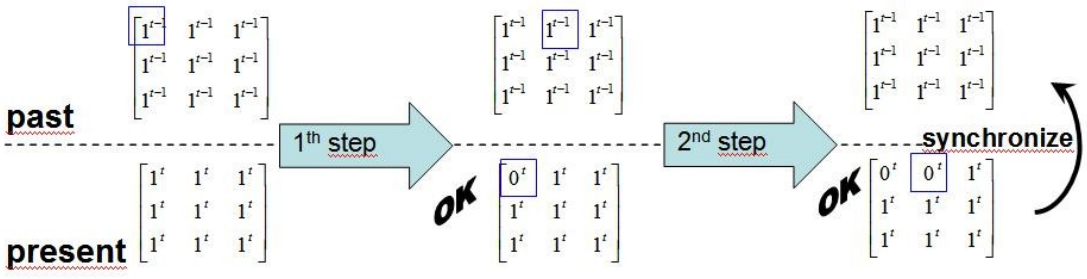


Figure 13 – Synchronizing a cell space in TerraME

Synchronization should occur after each iteration. For example, in the "Game of Life" code in Figure 10, after traversal of all cells, we have a "present" cell space which is different from the "past" cell space. Before the next iteration, it is necessary to synchronize the cell spaces. As a good modelling practice, in a neighbourhood based rule, the modeller should only update the attributes of the central cell. The neighbour 's attributes are read-only. *The flow of information is always from the neighbours to the central cell.*

4 Examples of physical models in TerraME

In this section presents four different rain drainage models. The examples evolve from a simples non-spatial model to a spatial model integrated in a geographic database.

4.1 The "Hello World" model

The simplest rain drainage model is a non-spatial model that considers all terrain as point in the space (see Figure 14). The water is a continuous variable Q that collects the rain input flow. The drainage is proportional to Q, where K is the flow coefficient constant. It follows that $\Delta Q_t = 2 - K * \Delta Q_{t-1}$, and $Q = \Sigma_0^t \Delta Q_t$. For a constant rain, Figure 15 shows the simulation results. For each time instant, it indicates the input rain flow (rain), the water in the system (Q), and the output flow (drainage). The model reaches a steady state after 10 minutes in the simulation clock. Figure 16 presents the TerraME source code for this model.

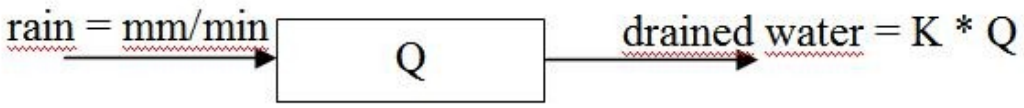
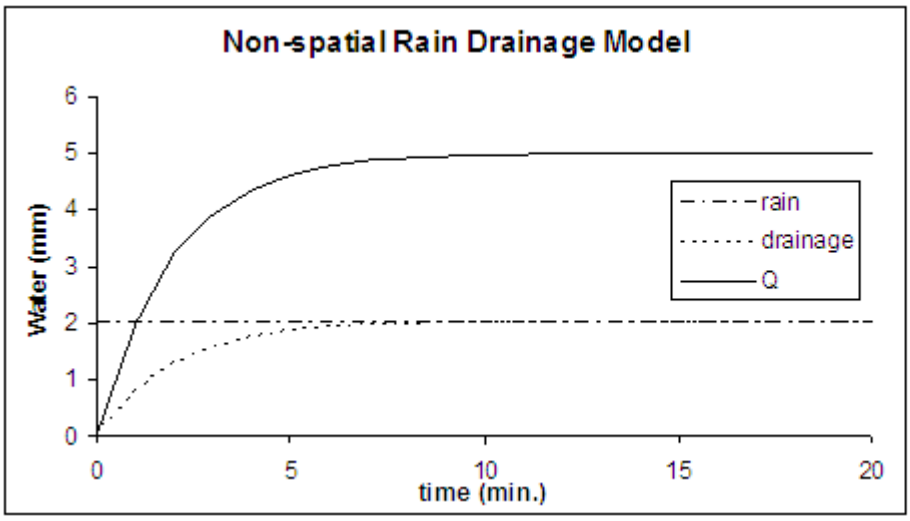


Figure 14 - A non-spatial rain drainage model.



time	rain	drainage	Q
0	2	0.0	0.0
1	2	0.8	2.0
2	2	1.3	3.2
3	2	1.6	3.9
4	2	1.7	4.4
5	2	1.8	4.6
6	2	1.9	4.8
7	2	1.9	4.9
8	2	2.0	4.9
9	2	2.0	4.9
10	2	2.0	5.0
11	2	2.0	5.0
12	2	2.0	5.0
13	2	2.0	5.0
14	2	2.0	5.0
15	2	2.0	5.0
16	2	2.0	5.0
17	2	2.0	5.0
18	2	2.0	5.0
19	2	2.0	5.0
20	2	2.0	5.0

Figure 15 - The amount of water in the system during the simulation.

```
C = 2; -- rain/t
```

```
K = 0.4; -- flow coefficient
-- GLOBAL VARIABLES
q = 0;  input = 0; output = 0;
-- RULES
for time = 0, 20, 1 do
	-- soil water
	q = q + input - output;
	-- rain
	input = C;
	-- drainage
	output = K*q;
	-- report
	print(time.."\t"..input.."\t"..output.."\t"..q);
end
```

Figure 16 - TerraME code for the non-spatial rain drainage model.

4.2 A simple spatial model

We now consider a 1D model. Space is modeled as a list of locations $Q = \{ q_i \mid \forall i = 1..N\}$, where N = 10. The model executes the same rules with the same parameters in each space location. The temporal variation of water in each location is equal to the graphic shown in Figure 15.

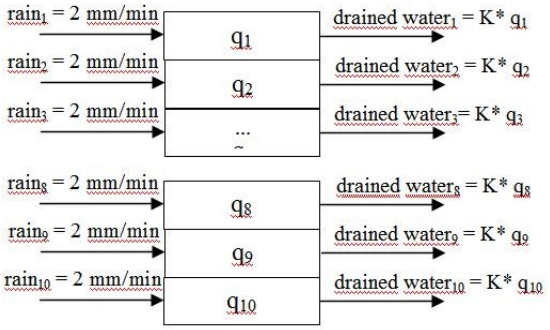


Figure 17 - A 1D spatial rain drainage model.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coeficient
-- GLOBAL VARIABLES
q = {}; -- a 1D table
-- RULES
for i = 1, 10, 1 do q[i] = 0; end
for time = 1, 20, 1 do
	-- rain and drainage
```

```
      for i = 1, 10, 1 do
       q[i] = q[i] + C;
      q[i] = q[i] - K*q[i];
   end
   -- report: soil water (Q)
      print("t: "..time );
      for i = 1, 10, 1 do print("["..i.."]: "..q[i]); end
end
```

Figure 18 - TerraME source code for a 1D model.

Figue 18 presents the TerraME source code for the unidimensional spatial drainage model. The variable q is a list, which locations q_i have been initialized with the value 0 (zero). The "for...end" statement from TerraME *Modelling Language* has been used to traverse the list.

4.3 A 2D spatial model

We can now extend the model to a 2D grid. Figure 19 shows the conceptual model for a 2D spatial drainage model, using a grid $Q = \{ q_{i,j} \mid i = 1..n \text{ and } \forall j = 1..n \}$.

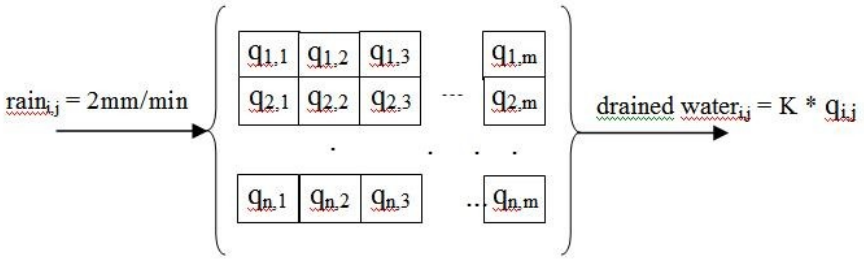


Figure 19 - A 1D spatial rain drainage model.

The TerraME source code for this model is in Figure 20. The variable *q* represents s a bidimensional grid. To traverse the space representation the modeller has used a block of code containing two nested "for...end" statements.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coefficient
-- GLOBAL VARIABLES
q = {};
-- RULES
for i = 1, 10, 1 do
      q[i] = {};
      for j = 1, 10,1 do
        q[i][j] = 0;
      end
```

```
end
for time = 0, 20, 1 do -- rain and drainage
      for i = 1, 10, 1 do
       for j = 1, 10,1 do
              q[i][j] = q[i][j] + C;
        q[i][j] = q[i][j] - K*q[i][j];
       end
      end
end
```

Figure 20 - The TerraME source code for the 2D drainage model

4.4 A spatial model integrated to a geographic database

In previous examples, we have not discussed how to read data. TerraME reads data from a TerraLib spatial database, as described in section 3.2. Figure 21 presents the TerraME code of the rain drainage model integrated to a *TerraLib* database.

Figure 20 - The spatial rain drainage model integrated to a geographical database.

The cell space csQ is retrieved from a layer in a *TerraLib* geographic database. For this example, we use the "cabecaDeBoi.mdb" database, available from the TerraME site. Each cell has an attribute called soilWater. The function load(), retrieves data and initializes the cells. We use the ForEachCell function to traverse the cellular space. The functions save() stores the soil water distribution at each simulation time step. A *view* called "Result" is created in the database. At each simulation step, it adds a new *theme* to this *view* to store the current values of the "soilWater" attribute. The reader may use the *TerraView* software to explore the data.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coefficient
```

```
FINAL_TIME = 20;
-- PART 1 – Retrieve the cell space from the database
csQ = CellularSpace{
        dbType = "ADO",
        host = "localhost",
        database = " c:\\TerraME\\Database\\cabecaDeBoi.mdb",
        user = "",
        password = "",
        layer = "cellsLobo90x90",
        theme = "cells",
        select = { "height", "soilWater" }
}
-- RULES
csQ:load();
CreateMooreNeighbourhood(csQ);
csQ:synchronize();

for time = 1, FINAL_TIME, 1 do
        -- PART 2: It's raining in the high areas
        ForEachCell(
                csQ,
                function( cell )
                        if(cell.height > 254) then
                                cell.soilWater = cell.past.soilWater + C;
                        end
                        return true;
                end
        );
        csQ:synchronize();

        -- PART 3: create a temporary variable to store the flow
        ForEachCell( csQ,
        function(cell) cell.flow = 0; return true; end);

        -- Calculate the drainage and the flow
        ForEachCell( csQ,
                function( cell )
       -- PART 4: calculate the drainage
       cell.soilWater = cell.past.soilWater –
                    K*cell.past.soilWater;
       end
   );


    for i, cell in pairs( csQ.cells ) do
     -- count the lower neighbors
                countNeigh = 0;
                ForEachNeighbour(cell,0,
                        function(cell, neigh)
                                if  (cell ~= neigh) and
                                   (cell.height >= neigh.height) then
                               countNeigh = countNeigh + 1
                                end
```

```
            end);
      -- PART 5: calculates the flow to neighbors
                if(countNeigh > 0 ) then
                        flow = cell.soilWater/countNeigh;
                -- send the water to neighbors
                    ForEachNeighbour(cell,       0,
                      function(cell, neigh)
                                    if  (cell ~= neigh) and
                                       (cell.height > neigh.height) then
                                            neigh.flow = neigh.flow + flow;
                                    end
                          end
                        );
                end
        end
        ForEachCell( csQ,
                function( cell )
                        cell.soilWater = cell.flow;
                        return true;
                end
        );
        csQ:synchronize();
        -- report: soil water
        print("t: "..time );
        if (time == FINAL_TIME) then
                csQ:save( time, "water", {"soilWater"} );
    end
end
```

Figure 21 - The TerraME source code for a spatial model inside a geographical database.

4.5 A Simple Hydrological Model

We now show a rain drainage model using TerraME. This model simulates the rain drainage using a digital terrain model of a village in Minas Gerais state, Brazil, called "Cabeça de Boi". The data is available from the TerraME site, in a database called "cabeça de boi.mdb" (see Figure 22).

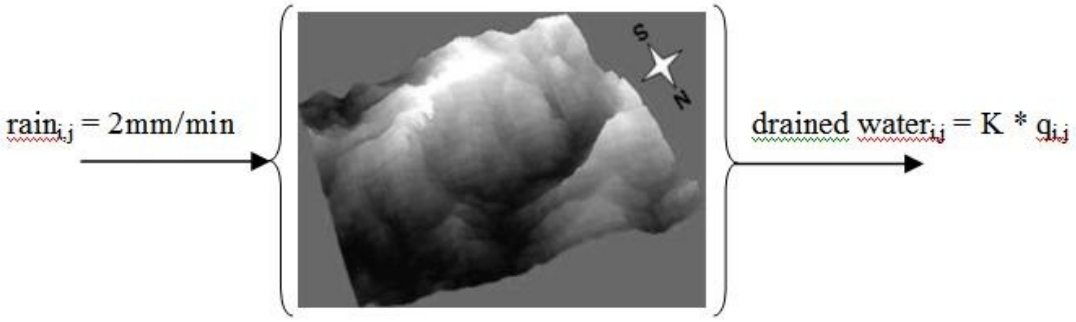


Figure 22 - The spatial rain drainage model using a DTM.

Figure 23 presents the TerraME source code the spatial rain drainage that uses the terrain elevation data in model rules. Each cell has two attributes: `soilWater` and `height`. At each simulation step, the code calculates the input rain flow. The cells send their water to their lower neighbours. The model calculates how much water each cell sends to its neighbours, and stores the result in a new cell attribute called `flow`. Then, the model traverses the cellular space again, and, at each cell, the value of this new attribute is added to the attribute `soilWater`. The program has the following parts:

- Retrieve the cell space from the database (using the `CellularSpace` constructor.
- Simulate rain in the high areas of the terrain using the `ForEachCell` function with `cell.soilWater = cell.past.soilWater + C`. Syncronize the cell space after this simulation.
- Create a temporary variable to store the flow, using `ForEachCell`.
- Calculate the drainage from the soil water stored in the cell.
- Calculate the flow, which is the non-drained soil water divided equally among the neighbours.
- Update the water in the cell from the accumulated values in the temporary attribute `flow`.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coefficient
FINAL_TIME = 20;
-- PART 1 – Retrieve the cell space from the database
csQ = CellularSpace{
	dbType = "ADO",
	host = "localhost",
	database = " c:\\TerraME\\Tutorial\\cabecaDeBoi.mdb",
	user = "",
	password = "",
	layer = "cellsLobo90x90",
	theme = "cells",
	select = { "height", "soilWater" }
}
-- RULES
csQ:load();
CreateMooreNeighbourhood(csQ);
csQ:synchronize();

for time = 1, FINAL_TIME, 1 do
	-- PART 2: It's raining in the high areas
	ForEachCell(
	  csQ,
	  function( cell )
		if(cell.elevation > 254) then
			cell.soilWater = cell.past.soilWater + C;
		end
		return true;
	  end
	);
	csQ:synchronize();

	-- PART 3: create a temporary variable to store the flow
	ForEachCell( csQ,
	  function(cell) cell.flow = 0; return true; end);

	-- Calculate the drainage and the flow
	ForEachCell( csQ,
	  function( cell )
	-- PART 4: calculate the drainage
	 cell.soilWater = cell.past.soilWater -
			K*cell.past.soilWater;
	cell.flow = 0;

      -- count the lower neighbors
	    countNeigh = 0;


      ForEachNeighbour(cell,0,
		function(cell, neigh)
			if(cell.elevation >= neigh.height) then
		       countNeigh = countNeigh + 1
```

```
                    end
        end);
      -- PART 5: calculates the flow to neighbors
        if(countNeigh > 0 ) then
                flow = cell.soilWater/countNeigh;
        -- send the water to neighbors
         ForEachNeighbour(cell,          0,
           function(cell, neigh)
                        if(cell.height > neigh.height) then
                                neigh.flow = neigh.flow + flow;
                        end
                end
         );
       end
      end
      ForEachCell( csQ,
       function( cell )
              cell.soilWater = cell.flow;
              return true;
       end
      );
      csQ:synchronize();
      -- report: soil water
      print("t: "..time );
      if (time == FINAL_TIME) then
       csQ:save( time, "water", {"soilWater"} );
   end
end
```

Figure 23 - The TerraME source code for a simple hydrological model.

5 Examples of land change models in TerraME

This section presents TerraME examples for a different problem: modelling of land change. We will consider the database "amazonia.mdb", which contains a 100 x 100 km2 cell space with data related to deforestation in Amazonia. Figure 24 shows a picture of the deforestation for each cell. This data is a simplified version of the database used in . The attributes of the cell space are:

- defor: percentage of deforestation;
- pop_dens_96: population density from 1996 census;
- pop_tx_urban_96: urbanization rate from 1996 census;
- pop_pc_migr_91_96: migration rate from 1991 to 1996;
- agr_area_small: percentage of cultivated area for small farms;
- agr_area_medium: percentage of cultivated area for medium farms;
- agr_area_large: percentage of cultivated area for large farms;
- dist_urban_areas: average distance to urban areas;
- dist_roads: average distance to roads;
- conn_markets_inv_p: strength of connection to markets
- clima_humi_min_3_ave: humidity in the three driest months;
- clima_precip_min_3_a: precipitation in three driest months;
- soils_fert_B1: average soil fertility
- prot_all1: percentage of protected areas in 1996
- prot_all2: proposed percentage of protected areas in 2006

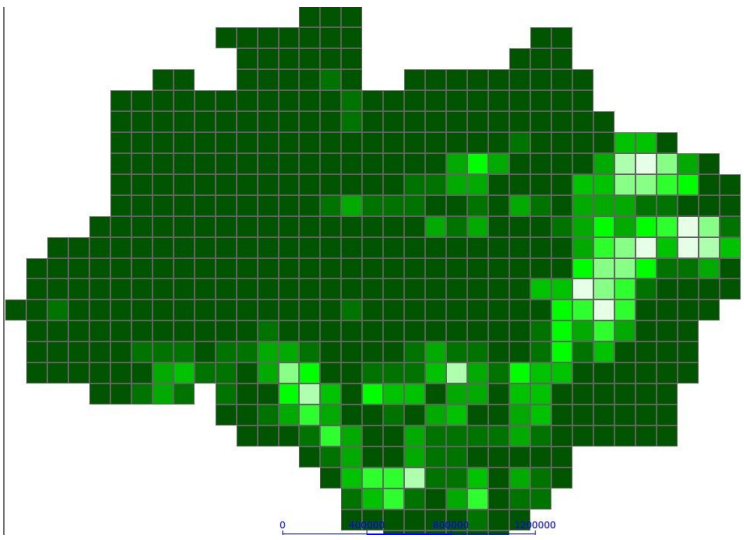


Figure 24 – A cell space of deforestation in Amazonia

The model considers a fixed demand for change, which will be allocated spatially. It calculates the potential for change at each cell. Then, it divides the demand as a proportion of the potential of change. We will consider three models: a simple diffusive model, a simple regression model, and a spatial regression model.

5.1 A spatial diffusive model for land change

Consider a spatial model that allocates 30.000 km2 of deforestation in Amazonia for 10 years. The potential of change for each cell is the average of neighbours' deforestation. The allocation function uses is proportional to the cell's potential, divided by the total potential for change. The result is shown in Figure 25 and the model is shown in Figure 26. It works as follows:

1. Reads the data from the database (command `csQ = CellularSpace{...);`
2. Creates a 3x3 neighbourhood (`CreateMooreNeighbourhood (csQ)`);
3. Defines a new attribute for potential for change (using the command `ForEachCell( .. cell.pot = 0 ... );`
4. Calculate the change potential for each cell. This requires a traversal of the cell space (`for...`). The potential for change for a cell is the average of its neighbour's deforestation.
5. Assign the demand based on the potential for each cell. This needs a second `for...` loop. This loop is inside an allocation loop that considers the case where the change potential for a cell may exceed 100% of deforestation.
6. Syncronize the cell space after each time step and save the last time step.

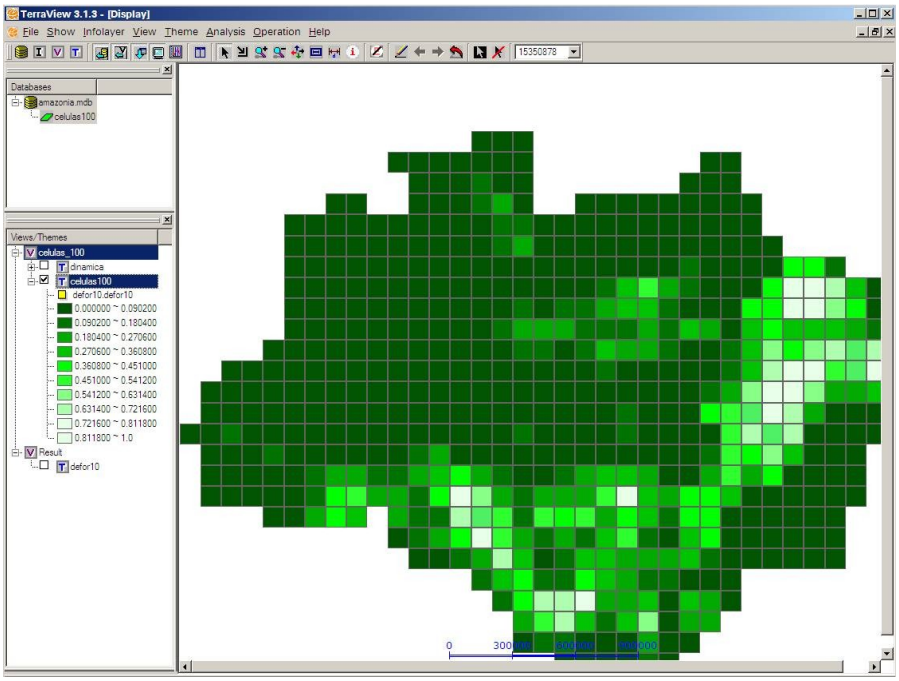


Figure 25 – Result of the diffusive model after 10 years.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000;
FINAL_TIME = 10;
ALLOCATION = 30000;
LIMIT      =   30;
-- GLOBAL VARIABLES
csQ = CellularSpace{
        dbType = "ADO",
        host = "localhost",
        database = "c:\\TerraME\\Database\\amazonia.mdb",
        user = "",
        password = "",
        layer = "celulas100",
        theme = "dinamica",
        select = {"defor"}
}
-- RULES
csQ:load();
CreateMooreNeighbourhood(csQ);
csQ:synchronize();
for time = 1, FINAL_TIME, 1 do
        print("t: "..time );
        -- initialize the potential
        for i, cell in pairs( csQ.cells ) do
       cell.pot = 0;
    end

        total_pot = 0;
        for i, cell in pairs( csQ.cells ) do
                -- Calculate the change potential for each cell
                countNeigh = 0;
                ForEachNeighbour( cell, 0,
                        function(cell, neigh)
          -- The potential of change for each cell is
          -- the average of neighbors' deforestation.
          -- fully deforested cells have zero potential
              if (cell.defor < 1.0 ) then
                cell.pot  = cell.pot  + neigh.defor;
               countNeigh = countNeigh + 1;
                        end
                ); -- for each neighbor
                if(cell.pot > 0 ) then
                        -- increment the total potential
                        cell.pot = cell.pot / countNeigh;
                        total_pot = total_pot + cell.pot;
                end
          end; -- for each cell


  -- ajust the demand for each cell so that
  -- the maximum demand for change is 100%
```

```
  -- adjust the demand so that excess demand is
  -- allocated to the remaining cells
  -- there is an error limit (30 km2 or 0.1%)
  total_demand = ALLOCATION;
  while (total_demand > LIMIT ) do
          print("total_demand: "..total_demand );
     for i, cell in pairs( csQ.cells ) do
        if (cell.pot > 0) then
           prop_cell = cell.pot/total_pot;
           newarea = prop_cell* total_demand;
           cell.defor = cell.past.defor +
                     newarea/CELL_AREA;
           if (cell.defor >= 1) then
              total_pot = total_pot - cell.pot;
              cell.pot  = 0;
              excess = (cell.defor - 1)*CELL_AREA;
              cell.defor = 1;
           else
              excess = 0;
           end
           -- adjust the total demand
           total_demand = total_demand - (newarea - excess)
       end
     end
     csQ:synchronize();
   end

   if  (time == FINAL_TIME) then
           csQ:save( time, "defor1", {"defor"} );
       end
end
```

Figure 26 - The TerraME source code for a simple diffusive land change model.

5.2 A regression model for land change

We will now consider a regression model based on three driving forces: distance to urban centres, connection to markets, and protected areas. The potential for change is based a linear regression between the cell's current deforestation and the expected deforestation, as follows:

- Calculate the expected deforestation as

  expected = -0.45*log (distance to urban areas) + 0.26*(connection to markets) -0.14*(protected areas) + 2.313

- Calculate the potential for change as

  cell.pot = expected – cell.defor

- Normalize the potentials (since there may be negative potentials) and allocate 30.000 km2 for 10 years.

This model is a simplified version of the detailed deforestation model developed by . Please see that document for details on the model. The model code is shown in Figure 27 and the result in Figure 28.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000;
FINAL_TIME = 10;
ALLOCATION = 30000;
LIMIT      =   30;
-- GLOBAL VARIABLES
csQ = CellularSpace{
        dbType = "ADO",
        host = "localhost",
        database = "c:\\TerraME\\Database\\amazonia.mdb",
        user = "",
        password = "",
        layer = "celulas100",
        theme = "dinamica",
        select= {"defor", "dist_urban_areas",
          "conn_markets_inv_p", "prot_all2" }
}
-- RULES
csQ:load();
CreateMooreNeighbourhood(csQ);
csQ:synchronize();
for time = 1, FINAL_TIME, 1 do
        print("t: "..time );
        -- initialize the potential
        for i, cell in pairs( csQ.cells ) do
       cell.pot = 0;
   end

        total_pot = 0;
        for i, cell in pairs( csQ.cells ) do
                -- The potential for change is the residue of a
```

```
                -- linear regression between the cell's
                -- current and expected deforestation
                -- according to the following model:
                        if (cell.defor < 1.0) then
                                expected =
            - 0.45*math.log10 (cell.dist_urban_areas)
            + 0.26*cell.conn_markets_inv_p
            - 0.14*cell.prot_all2
            + 2.313;
                                if (expected > cell.defor) then
                                  cell.pot = expected - cell.defor;
                                  total_pot = total_pot + cell.pot;
                            end
                             end
        end; -- for each cell
  -- adjust the demand so that excess demand is
  -- allocated to the remaining cells in an error limit (0.1%)
  total_demand = ALLOCATION;
  while (total_demand > LIMIT ) do
     for i, cell in pairs( csQ.cells ) do
        if (cell.pot > 0) then
           prop_cell = cell.pot/total_pot;
           newarea = prop_cell* total_demand;
           cell.defor = cell.past.defor +
                     newarea/CELL_AREA;
           if (cell.defor >= 1) then
              total_pot = total_pot - cell.pot;
              cell.pot  = 0;
              excess = (cell.defor - 1)*CELL_AREA;
              cell.defor = 1;
           else
              excess = 0;
           end
           -- adjust the total demand
           total_demand = total_demand - newarea + excess;
       end
      end
      csQ:synchronize();
   end
   if  (time == FINAL_TIME) then
           csQ:save( time, "defor2_", {"defor"} );
   end
end
```

Figure 27 - The TerraME source code for a linear regression land change model.

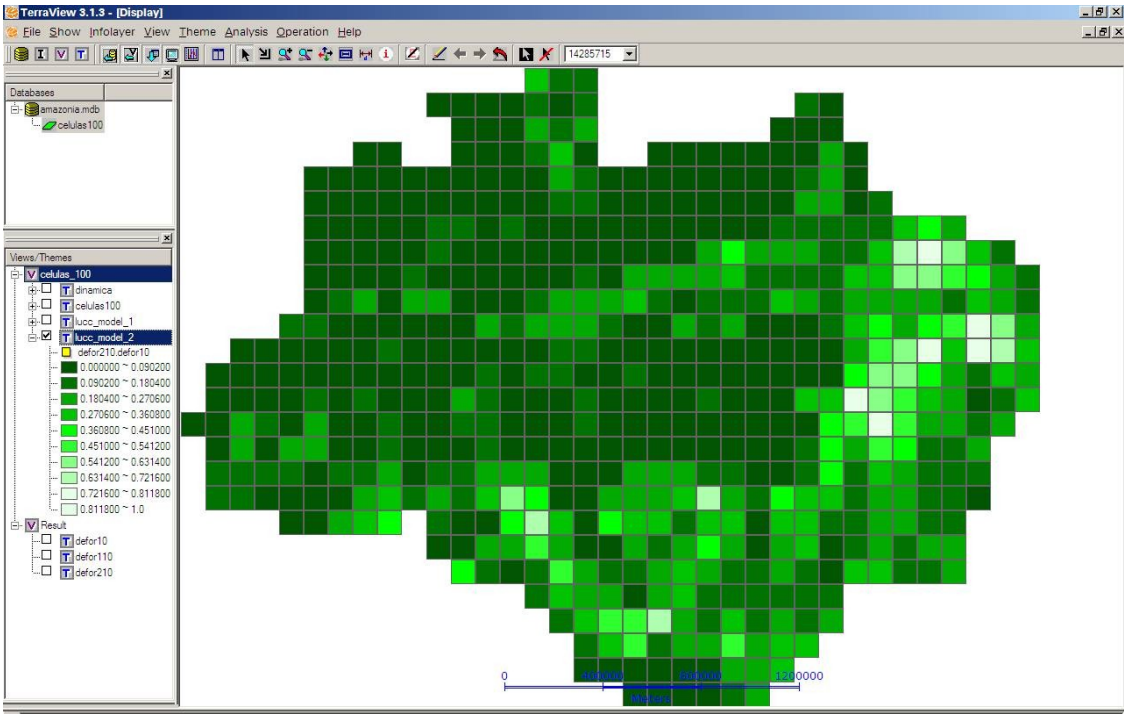


Figure 28 – Result of land change model based on linear regression

5.3 A combined diffusive/regression model

We will now consider a spatial regression model based on four driving forces: the deforestation on the neighbours, distance to urban centres, connection to markets, and protected areas. For a detailed discussion of the impact of neighbours on deforestation, see . The potential for change is based on the residues of a spatial regression between the cell's current deforestation and the expected deforestation according to the following model:

- Calculate the expected deforestation as

```
expected =  0.73*log10(AVERAGE(neighbor deforestation))
          - 0.15*log10 (distance to urban centres)
          + 0.05*(connection to markets)
          - 0.07*(protected areas) + 0.7734;
```

- Calculate the potential for change for each cell as

```
potential = expected – deforestation
```

- Allocate 30.000 km2 for 10 years for all cells with positive potentials. Note there may be negative potentials, which are cells with more deforestation than expected. In this case, there is no change for the cell.

This model is a simplified version of the detailed deforestation model developed by . The model code is shown in Figure 29 and the result in Figure 30.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000;
FINAL_TIME = 10;
ALLOCATION = 30000;
LIMIT      =   30;
-- GLOBAL VARIABLES
csQ = CellularSpace{
        dbType = "ADO",
        host = "localhost",
        database = "c:\\TerraME\\Database\\amazonia.mdb",
        user = "",
        password = "",
        layer = "celulas100",
        theme = "dinamica",
        select= {"defor", "dist_urban_areas",
          "conn_markets_inv_p", "prot_all2" }
}
-- RULES
csQ:load();
CreateMooreNeighbourhood(csQ);
csQ:synchronize();
for time = 1, FINAL_TIME, 1 do
        print("t: "..time );
        -- initialize the potential
        for i, cell in pairs( csQ.cells ) do
       cell.pot = 0;
       cell.ave_neigh = 0;
   end
   for i, cell in pairs( csQ.cells ) do
                -- Calculate the average deforestation
                countNeigh = 0;
                ForEachNeighbour( cell, 0,
                        function(cell, neigh)
         -- The potential of change for each cell is
         -- the average of neighbors' deforestation.
              if (cell.defor < 1.0 ) then
                cell.ave_neigh  = cell.ave_neigh
                   + neigh.defor;
                countNeigh = countNeigh + 1;
              end
                        end
                ); -- for each neighbour
     -- find the average deforestation
                if(cell.defor < 1.0 ) then
                        cell.ave_neigh = cell.ave_neigh / countNeigh;
                end
        end; -- for each cell

    total_pot = 0;
        for i, cell in pairs( csQ.cells ) do
                -- Potential for change
                        if (cell.defor < 1.0) then
                                expected =  0.73*cell.ave_neigh
```

```
                - 0.15*math.log10(cell.dist_urban_areas)
                + 0.05*cell.conn_markets_inv_p
                            - 0.07*cell.prot_all2
                            + 0.7734;

                if (expected > cell.defor) then
                                cell.pot = expected - cell.defor;
                                total_pot = total_pot + cell.pot;
                        end
                    end
            end; -- for each cell

    -- adjust the demand for each cell

    while (total_demand > LIMIT ) do
            print("total_demand: "..total_demand );
       for i, cell in pairs( csQ.cells ) do
          if (cell.pot > 0) then
             prop_cell = cell.pot/total_pot;
             newarea = prop_cell* total_demand;
             cell.defor = cell.past.defor +
                        newarea/CELL_AREA;
             if (cell.defor >= 1) then
                total_pot = total_pot - cell.pot;
                cell.pot  = 0;
                excess = (cell.defor - 1)*CELL_AREA;
                cell.defor = 1;
             else
                excess = 0;
             end
             -- adjust the total demand
             total_demand = total_demand - newarea + excess;
          end
         end
         csQ:synchronize();
     end
     if  (time == FINAL_TIME) then
             csQ:save( time, "defor3_", {"defor"} );
          end

end
```

Figure 28 – Code for land change model based on spatial regression

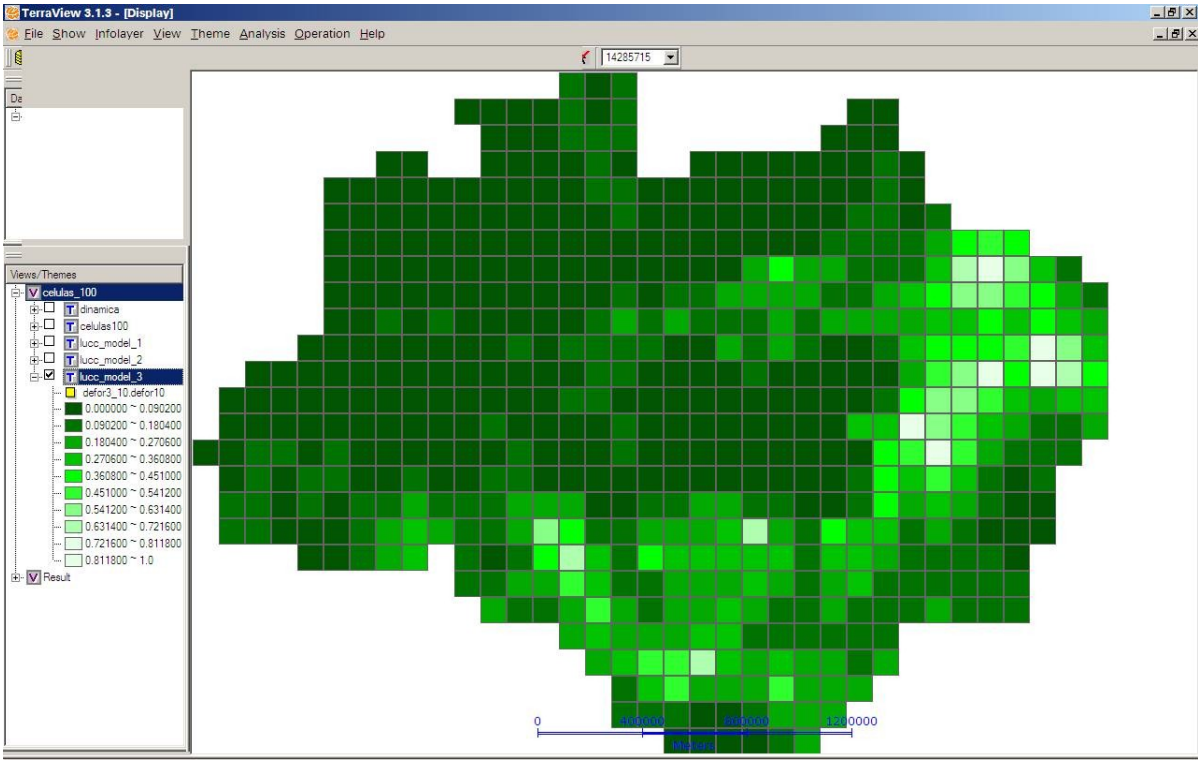


Figure 30 – Result of land change model based on spatial regression

References

ANNEX – INSTALLING TERRAME

This section shows how to install TerraME architecture in the Microsoft Windows platform. Unfortunately, the TerraME version for Linux is not ready yet.

Installing the TerraME Environment

Download TerraME.zip from www.dpi.inpe.br/cursos/environmental_modelling. Decompress the file to the C:\TerraME directory. It contains the directory TerraME, with the LUA interpreter and the TerraME development environment.

Installing the Databases

Download from www.dpi.inpe.br/cursos/environmental_modelling the databases:

- "CabecadeBoi.mdb" – a TerraLib database in Access format, containing data for the examples in hydrological modeling.
- "amazonia.mdb"- a TerraLib database in Access format, containing data for the examples in LUCC modeling.

Copy theses files the C:\TerraME\Database directory.

Installing the TerraView application

Install the TerraView application available from www.terralib.org. After TerraView is installed, visualize the databases cabecadeBoi and amazonia.

Installing the Eclipse Software Development Kit

Install the *Eclipse Software Development Kit* (Eclipse SDK) application, obtaining the latest version from the Eclipse site (www.eclipse.org). At the time of this writing, the SDK is contained in the file "eclipse-SDK-3.2.1-win32.zip". Copy this file to the "c:\" directory and uncompress it (Figures A.1 and A.2). Create a short-cut to the "eclipse.exe" application and move this short-cut to the desktop, Figure A.2.

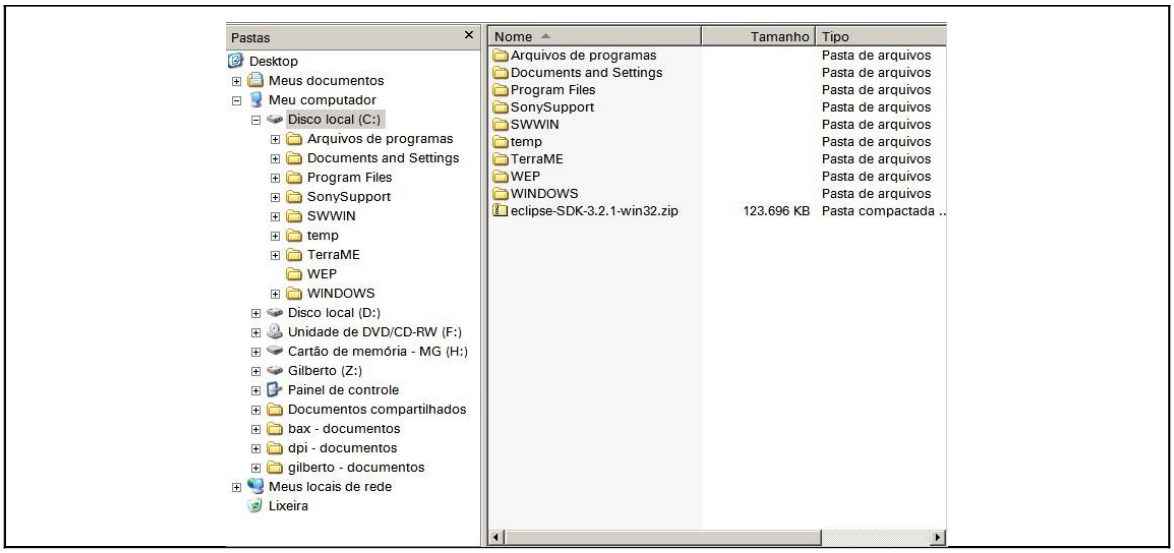


Figure A.1- Move the file "eclipse-SDK-3.2.1-win32.zip" to the C:\ directory.

Figure A.2 - Uncompress the "eclipse-SDK-3.2.1-win32.zip" and create a shortcut the the “eclipse.exe” file.

Installing the LUA Plugin for Eclipse

Next, you should install the LUA *plugin* for Eclipse, available from the site http://www.ideais.com.br/luaeclipse/. Unzip the file "luaeclipse_0.5.0.zip" and move the "luaeclipse_0.5.0" uncompressed directory to the "c:\eclipse\plugins\" subdirectory under the Eclipse installation directory (Figure A.3).

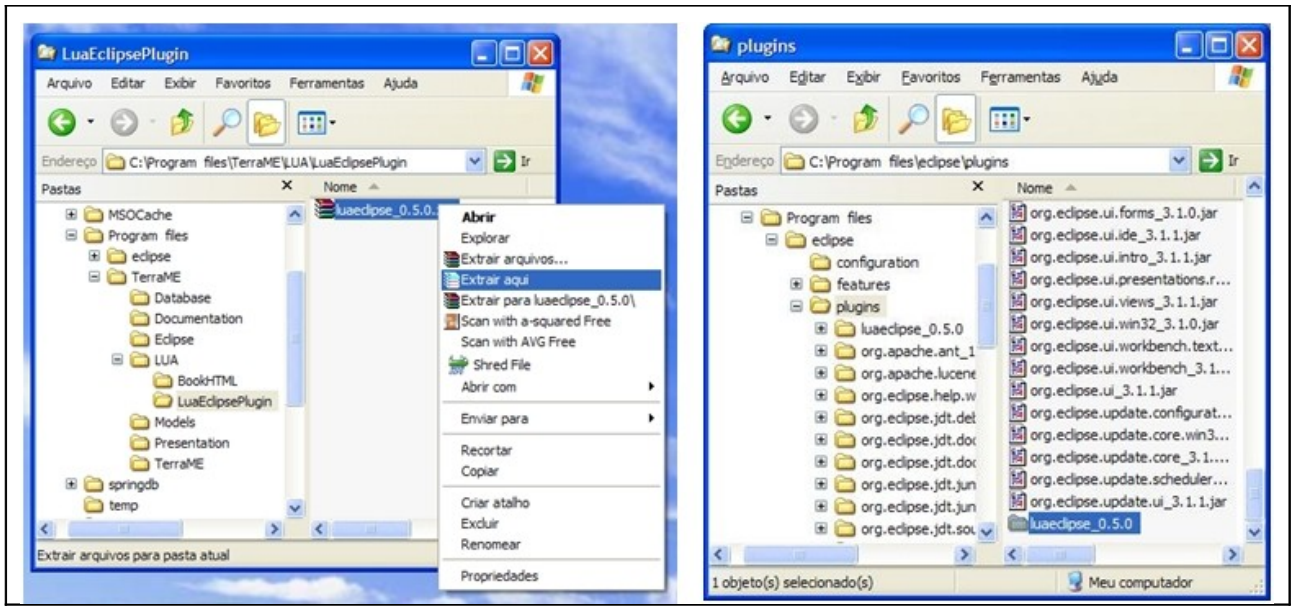


Figure A.3 – Installing the LUA plugin for eclipse.

Configuring a TerraME project in Eclipse

Next, you should configure an Eclipse to use TerraME. You should start Eclipse. Then, Eclipse asks to user to select the working directory. This directory is the "workspace" (Figure A.4). You should choose the TerraME installation directory as your workspace directory.

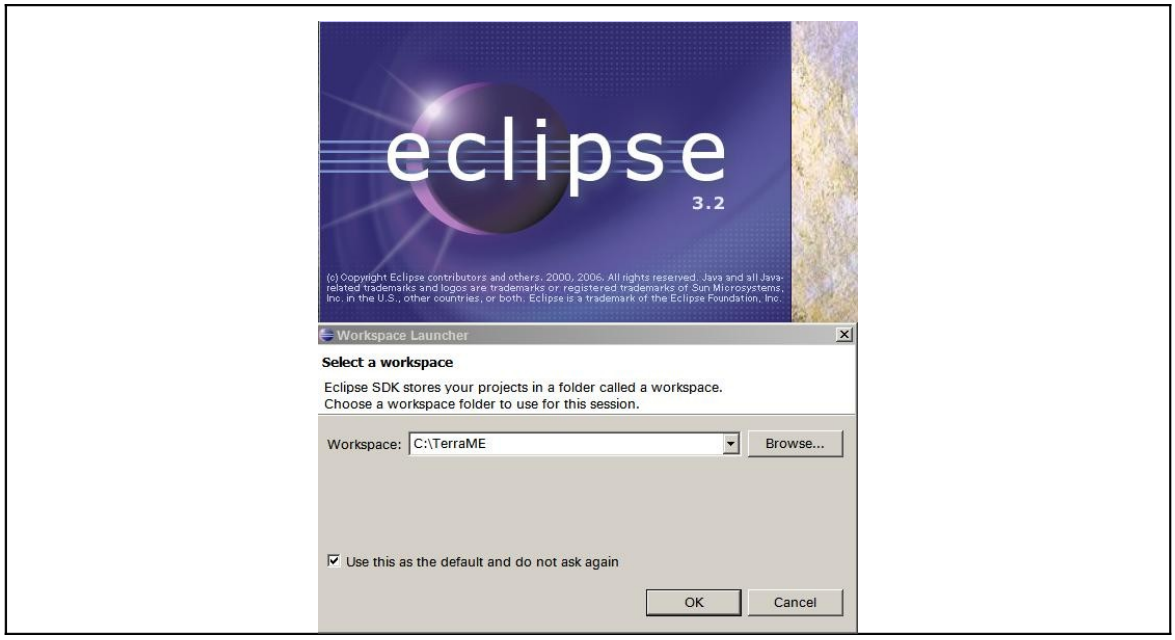


Figure A.4 - Choose the "c: \TerraME" as your workspace directory and close the Eclipse SDK Welcome screen.

Inside the workspace, create a new LUA project, clicking in the "New...Project" option from the "File" menu. Select a LUA Project (see Figure A.5) and in the dialogue box "New Project", type the name of your project (see Figure A.6). We have used the name "MyProject". Then, click on the buttons "Finish" and "Cancel". Create a new model file using "New...\File" from the "File" menu. In the "New file" dialogue box, enter the model name (Figure A.7). We have used the name "MyModel.lua". The ".lua" extension is required.

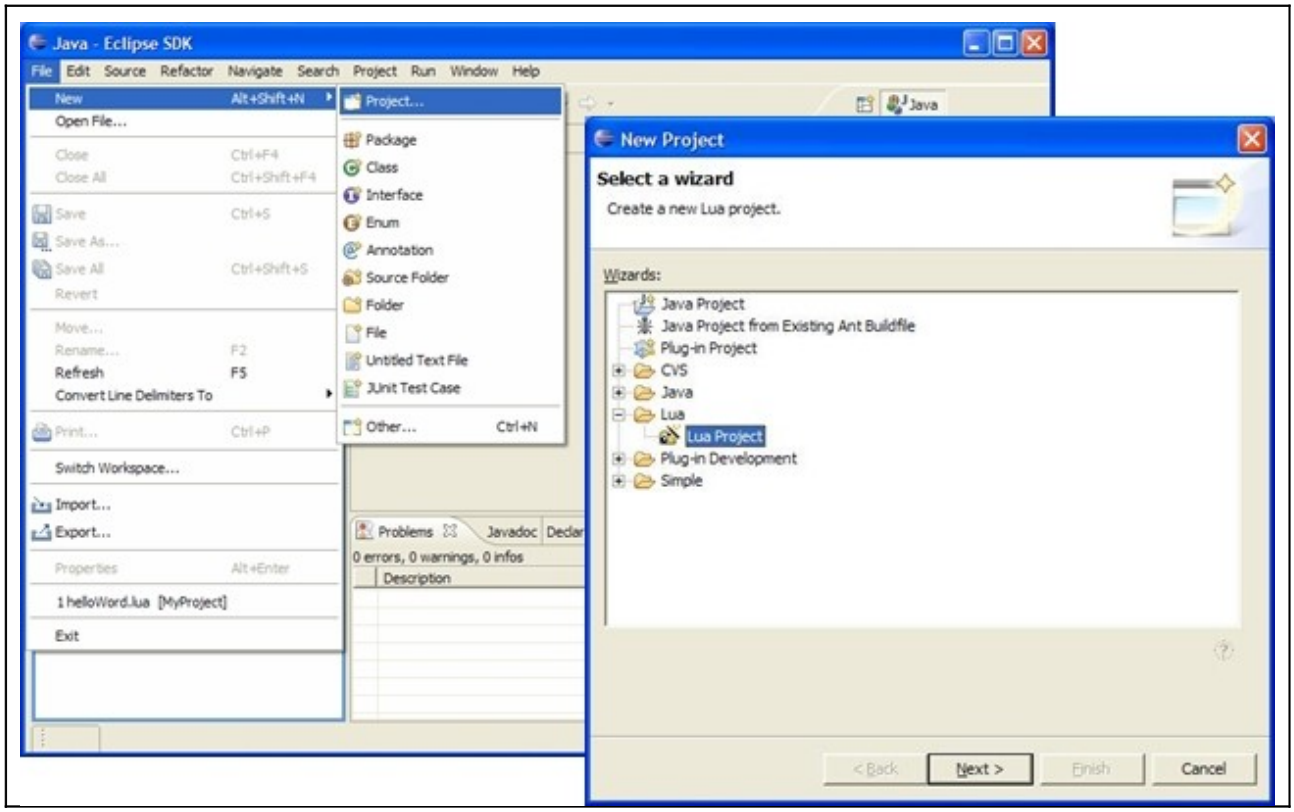


Figure A.5 - Create a new LUA project.

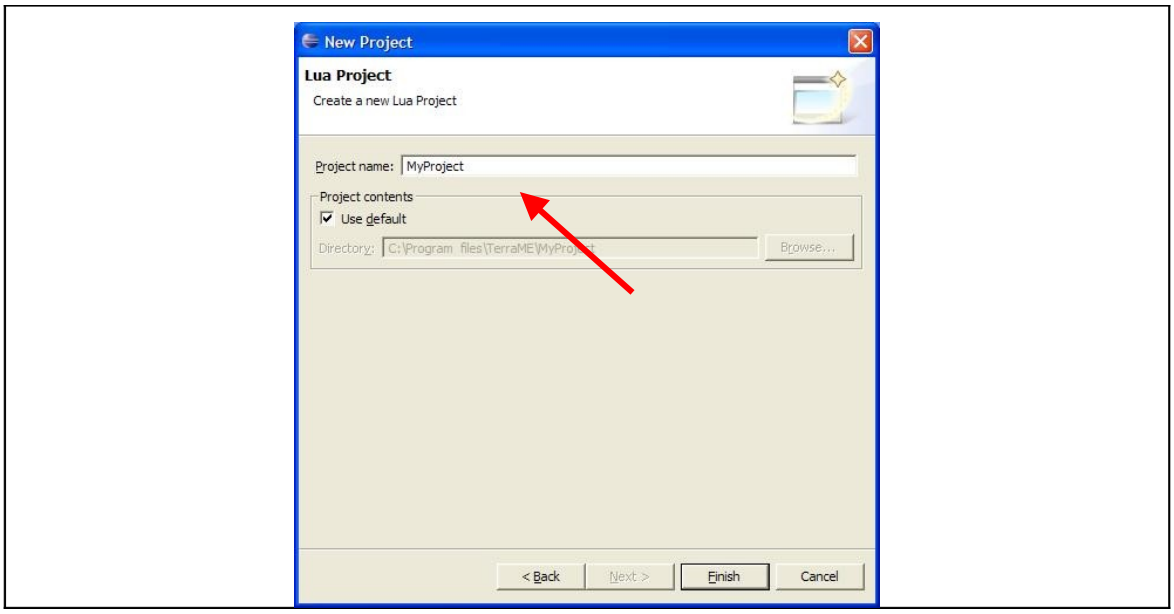


Figure A.6 - Choose your project name and click on the buttons "finish" and "cancel".

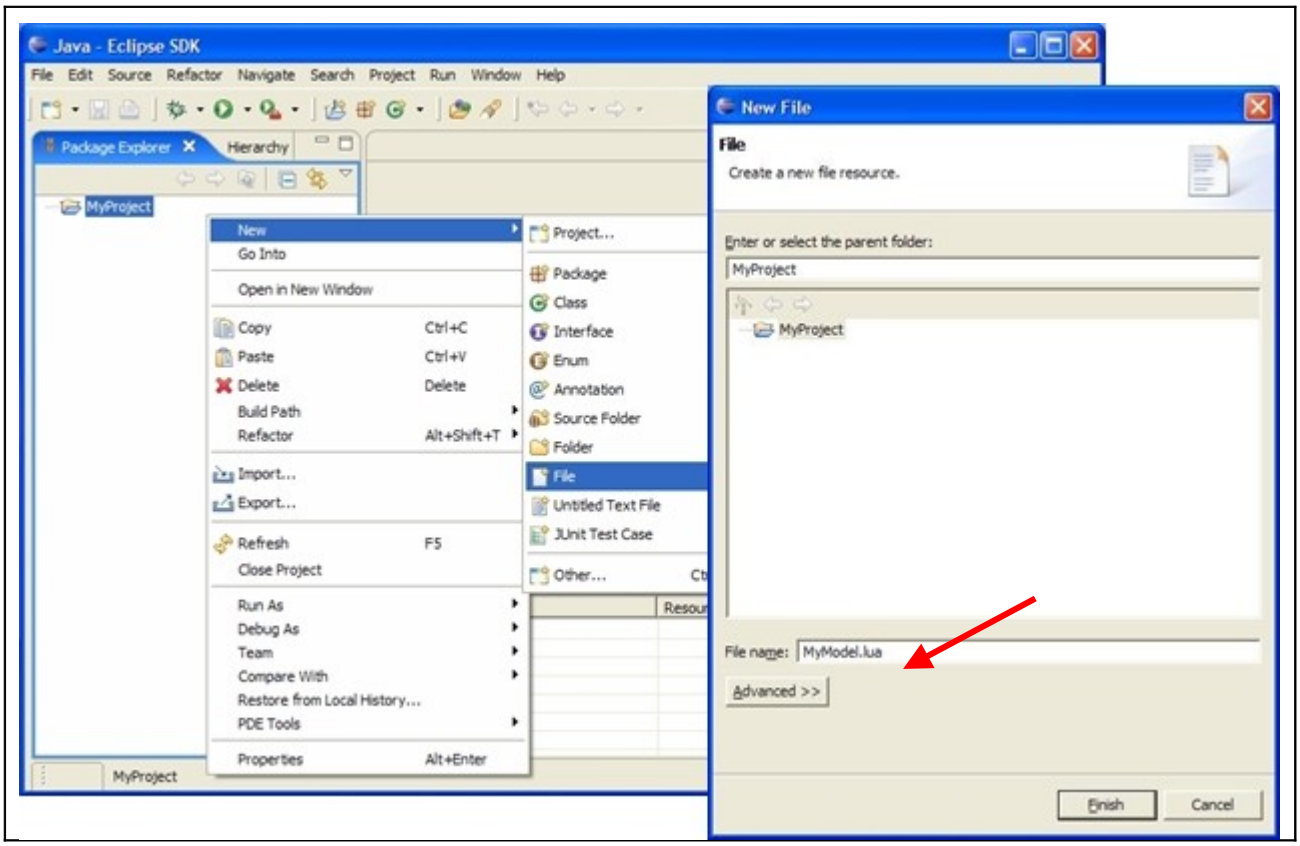


Figure A.7 - Create a new model file and provide its name.

Type the program shown in Figure A.8 inside the "MyModel.lua" file.

```
print("Hello World");
```

Figure A.8- This model in TerraME prints "Hello World".

To run this model, you should configure Eclipse to use the *TerraME Interpreter* to run the model file. Create a new configuration as shown in Figure A.9. Then, select the option "Run..." from the "Run" menu. Next, select the option "LUA Application" in the "Configurations" list box and click on the button "New" from the "Run" dialog box. Figure 2.16 shows the dialogue boxes where you can choose the configuration name and the file that should be executed when the Eclipse is asked to run the project.

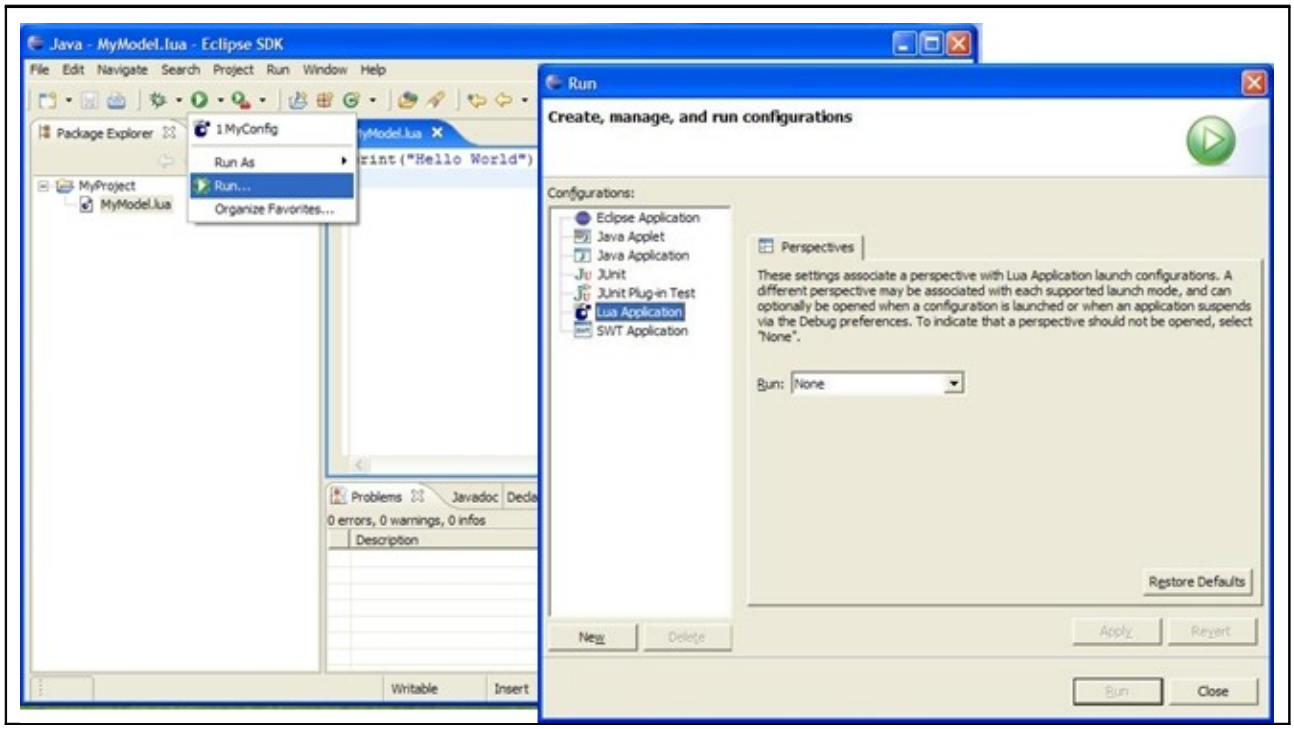


Figure A.9 - Choose the option "Run..." from the "run" menu. In the "Run" dialogue box, select "Lua Application" and click on the button "New".

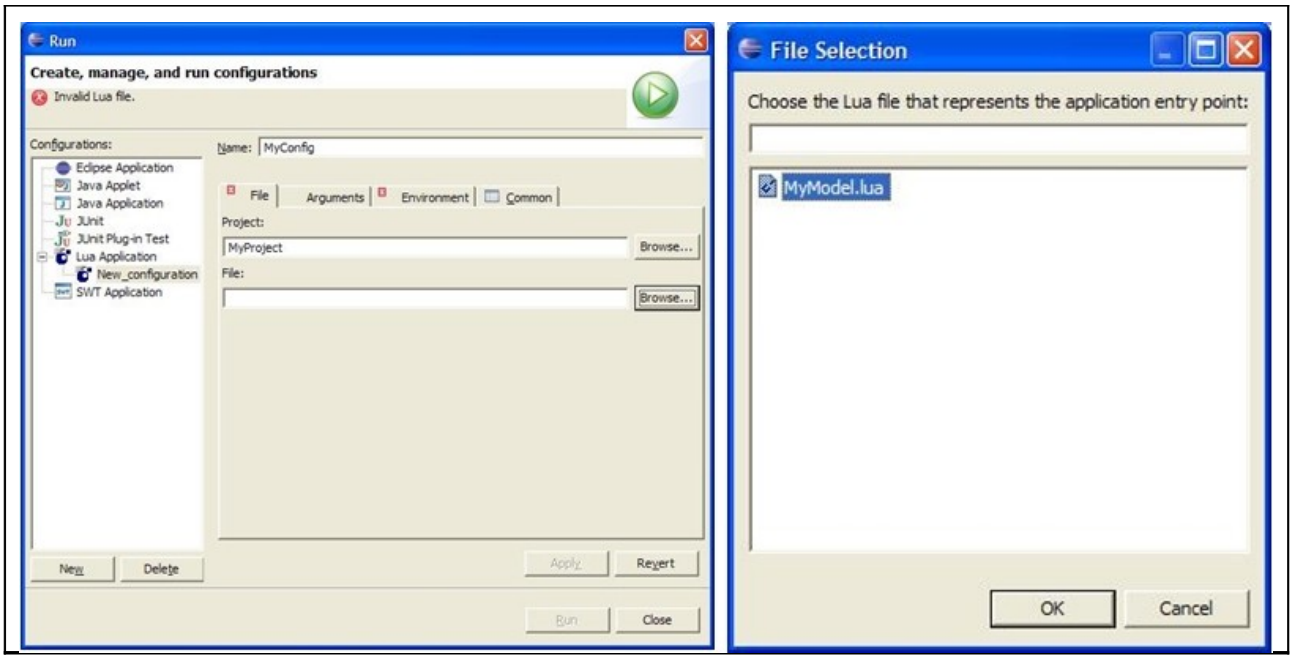


Figure A.10 – Define the file to be executed in the configuration

In the panel "Environment" from the dialogue box "Run", include the "TerraME.exe" application as the interpreter (Figure A.11). Click on the button "New" and provide the name and the complete path to the "TerraME.exe" interpreter.

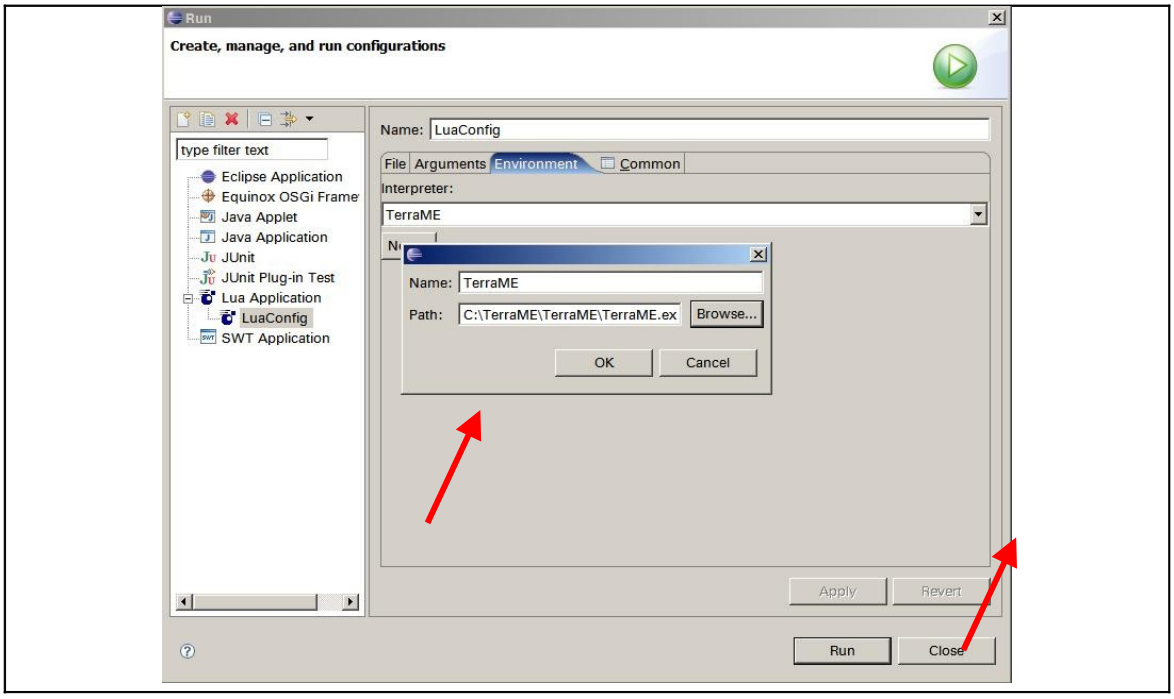


Figure A.11 – Select the interpreter for the Lua program.

Check the button "Run" in the panel "Common" from the dialog box "Run" to create a short-cut to your configuration on the Eclipse toolbar (Figure A.12). Then, click on the buttons "apply" and "run" to execute the project. Figure A.13 shows the short-cut for your project created in the Eclipse toolbar. The outcome from the model execution is shown in the Eclipse console.

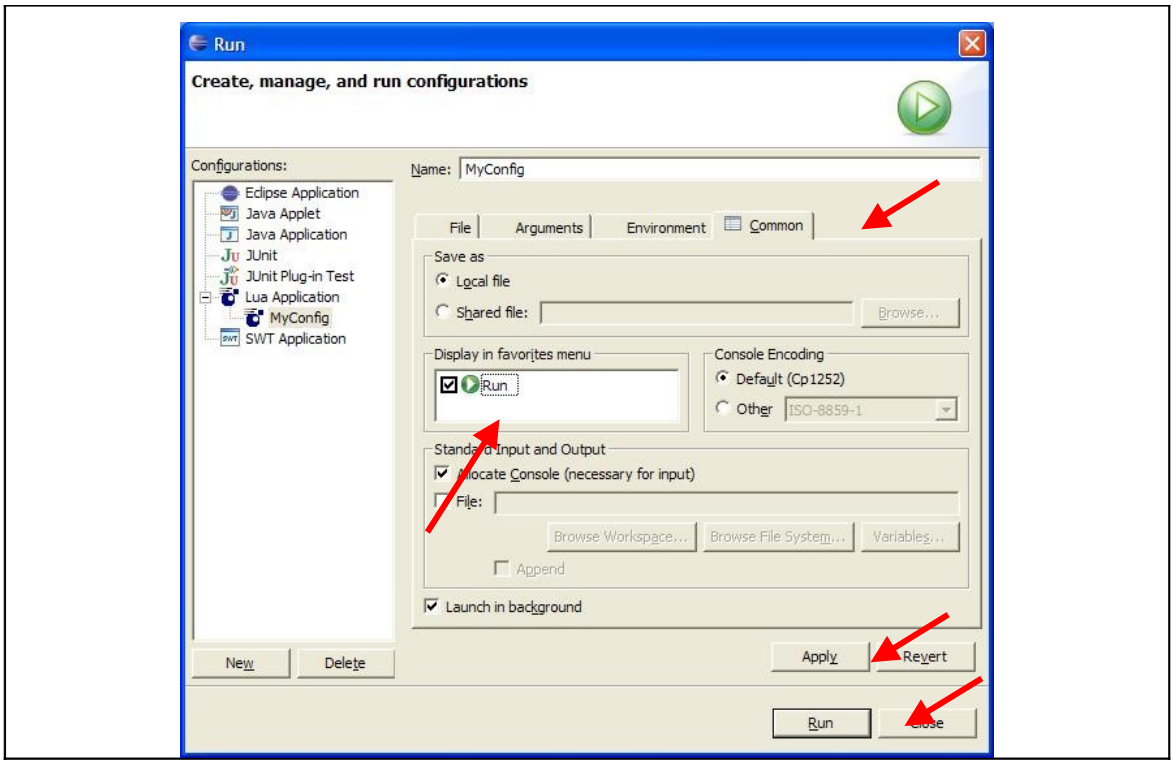


Figure A.12 - Check the button "run" to create a short-cut for your configuration in the Eclipse toolbar.

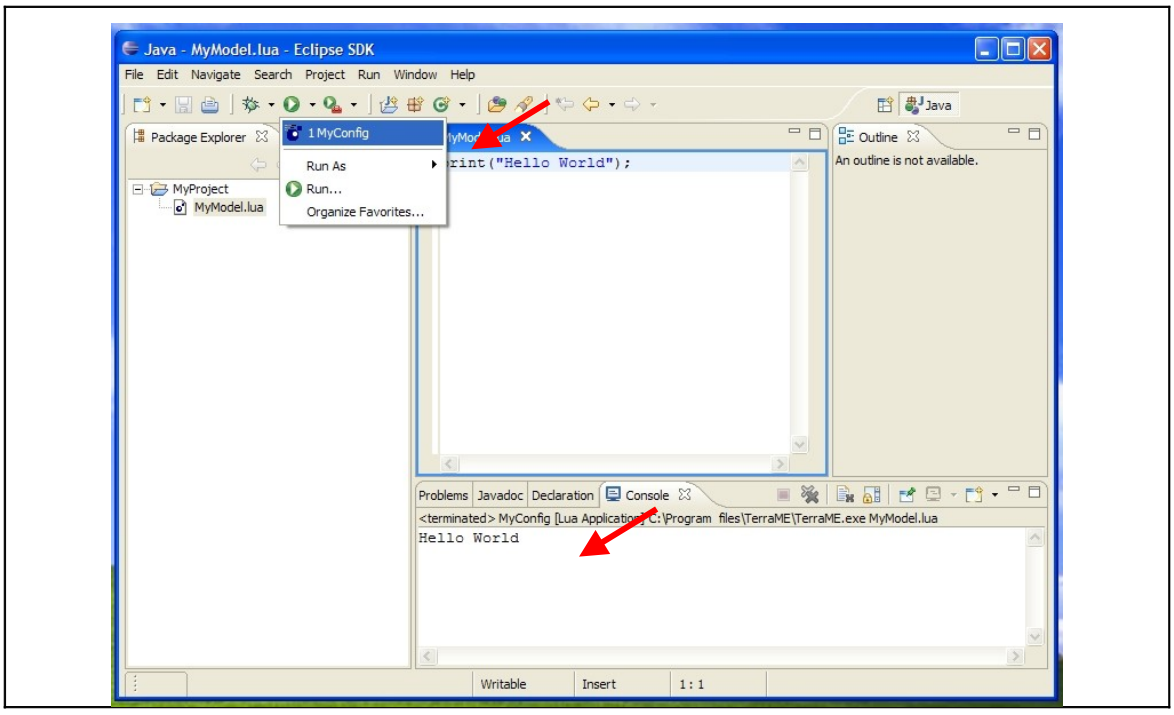


Figure A.13- A short-cut for "MyConfig" has been created in the Eclipse toolbar. The result from the project execution is shown on the "console" window.