Agent-Based Modeling and Simulation

Testing your programs

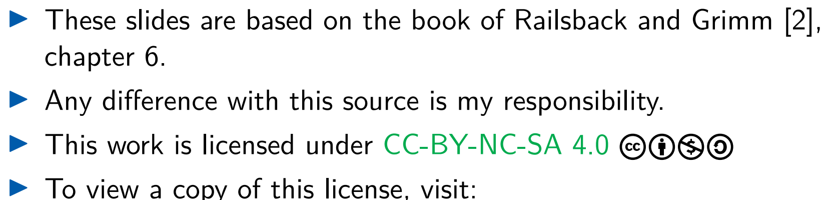
Dr. Alejandro Guerra-Hernández

Instituto de Investigaciones en Inteligencia Artificial
Universidad Veracruzana
Campus Sur, Calle Paseo Lote II, Sección Segunda No 112,
Nuevo Xalapa, Xalapa, Ver., México 91097

mailto:aguerra@uv.mx
https://www.uv.mx/personal/aguerra/abms

Doctorado en Inteligencia Artificial 2024

Credits



- These slides are based on the book of Railsback and Grimm [2], chapter 6.
- Any difference with this source is my responsibility.

- This work is licensed under CC-BY-NC-SA 4.0
- To view a copy of this license, visit:

  https://creativecommons.org/licenses/by-nc-sa/4.0/

Universidad Veracruzana

Ideas

- Many of the techniques in this session are about debugging, i.e., figuring out and fixing the causes of obvious mistakes.
- But then, focus is on software verification, *i.e.*, verifying if your software accurately implements your model formulation.
- In addition to testing the program, this requires:
  - An independent description of the model, *e.g.*, the ODD protocol.
  - Documenting tests in order to convince our clients, *e.g.*, thesis adviser, reviewer, decision maker, etc.
- All this makes modeling more efficient, *e.g.*, following a bottom-up approach.
- Software testing is impossible to separate from model analysis.

Objectives

- To understand seven common types of software errors;
- To understand ten important debugging techniques, including writing intermediate model results to output files for analysis in other software; and
- Why and how to document software tests.

Universidad Veracruzana

Typographical Errors

- Typing the wrong text in NetLogo is solved by the syntax checker.
- Example. Writing `ask turtlse` instead of `ask turtles`.
- However it is easy to missname variables when copy-pasting.
- Example. Suppose you have the following line:

```
1       set xcor random-normal meanxcor stddev
```

  and you copy-paste the code for working with y, editing it:

```
1       set ycor random-normal meanxcor stddev
```

  You've forgot to change `meanXcor`!
- Sometimes, the View can help detecting such errors.

Syntax Errors

- Forgetting to use brackets when required.
- Not leaving spaces between numbers and operators in mathematical expressions.
- Example. Writing `set xcor max-pxcor/2` instead of `set xcor max-pxcor / 2`.
- The Syntax Checker is quite good detecting such errors, but it does not warrant to find all syntax errors.

Universidad Veracruzana

Use of brackets

- A list of literal values, *e.g.*, `let l [a b c]`
- A sequence of commands, *i.e.*, a command block:

```
1 ask turtles [
2    forward 1
3    set pcolor blue
4 ]
```

- A conditional expression, *e.g.*, `if [xcor < 5]`
- As part of the globals, extensions, breeds, and breeds-own declarations, *e.g.*, **globals** `[v1 v2]`
- A list of procedure arguments, *e.g.*, **to** `move-agent [agent dist] ...` **end**
- An anonymous task: `let t [ [x y] -> (x + y)n ]`

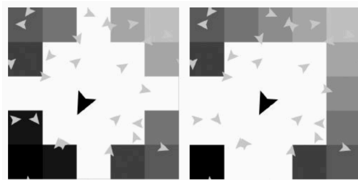
Universidad Veracruzana

Missunderstanding Primitives

- A primitive does not do what you think it does.
- Example. Forgetting that patch coordinates are integers, while turtle coordinates are floats. So that the following statements can have different results:

```
1 let neighbor-turtles turtles in-radius 2
2 let neighbor-turtles turtles-on patches in-radius 2
```

Moreover, they work differently in a turtle and a patch contexts:



Universidad Veracruzana

More misunderstandings

- Differences between `neighbors` and `in-radius`.
- Excercise. Open an Agent Monitor for a patch close to the center on the environment. Execute the following command:

```
1 ask neighbors [set pcolor red]
```

Observe the output when executing:

```
1 ask patches in-radius 1.5 [set pcolor blue]
```

Report your observations.

Universidad Veracruzana

Wrong Display Settings

- The configuration of the world through the Interface tab is not visible in the program.
- Changing the size, wrapping properties, or the origin can affect the behavior of our programs.
- The primitive `resize-world` can be used explicitly in the setup procedure to enhance visibility.

Universidad Veracruzana

Run-Time Errors

- The program is free of syntax and logic errors, but eventually does something that the computer can't handle.
- Examples:
  - Dividing by a variable that has a value of zero;
  - Trying to put an object outside the world;
  - Asking an agent to do something after dead;
  - Raising a number to a power bigger than the computer's memory;
  - Trying to open a file already closed;
  - Trying to delete a file that does not exists.
- Usually catch by the interpreter, but it is better to try to anticipate such errors.

Logic Errors

- The kind of error most likely to cause real problems.
- The program executes and produces results, but these are wrong because a logic mistake.
- Examples:
  - A variable is not initialized;
  - An equation is programmed wrongly;
  - An incorrect condition in an `if` statement.
- Sometimes wrong results are obvions, but most of the time demand careful testing.

Formulation Errors

- These are not programming errors, but become apparent only after the model software is written and executed.
- In building almost any model, some assumptions, algorithms, or parameter values will have unexpected consequences and require revision after the model is implemented and tested.
- Once these errors are detected and corrected, the model's written documentation must be updated accordingly.

Universidad Veracruzana

Syntax Checking

- Use the Check button in the Code tab.
- It is very good, but sometimes:
  - Issue error statements that are unclear;
  - Point to places in the code far away from the actual mistake, particularly in embedded contexts.
- The key to efficiency is checking often (every line or two).
- Keep your code free of syntax errors, even if it is incomplete, *e.g.*, add skeleton procedures if required.
- Use also skeletons for control structures.

Universidad Veracruzana

Visual Testing

- Take a look at the model's results on the Interface to see if anything is unexpected.
- Examples:
  - Are there obvious problems, e.g., all turtles born in the same place instead of randomly.
  - Do strange patterns emerge after the model runs for a while, e.g., all turtles die.
- You have to design your program to use the View effectively, e.g., use color, shape, label, size to emphasize visually the characteristics of the agents.

Universidad Veracruzana

Tricks for Visual Testing

- You can start using the View to test your software long before the software is complete, e.g., as soon as implementing setup.
- The primitive `scale-color` makes easy to shade the color of the agents based on variable values.
- Labels in turtles and patches are very useful. e.g., `set label energy-reserves`.
- Reduce the size of the world or the number of agents when testing.
- Slow execution down if needed.
- Add a step button to your Interface.
- It is also possible to add buttons for each procedure in the scheduling.

Universidad Veracruzana

Print Statements

- Insert statements that write information out to the display or to a file so you can see what is going on, e.g., `export-output`, `output`, `print`, `show`, `type`, and `write`.
- They can be used to detect the procedures when some problem occurs, *e.g.*, printing begin-end statements.
- Output the value of key variables at different times to diagnose why a model behaves unexpectedly. See `word`.
- Plots and Monitors can be used in the same sense.

Universidad Veracruzana

Command center rearranged

NetLogo

Interface | Info | Code

Edit Delete Add Button

normal speed

ticks:

view updates

continuous

Settings...

Command Center

Clear

observer>

Spot Tests with Agent Monitors

- Right-click your mouse on the View and open an Agent Monitor to any patch, and to the turtles and links on the patch.
- They show the values of the variables of the selected objects.
- They can be used to spot test calculations by manually recording the values, calculating by hand how they should change, and then stepping the model to see if it match your expectations.
- Monitors can be created programmatically:

```
set size size + growth
if size < 0
[
  inspect self
  user-message (word "Turtle" who "has negative size")
]
```

See also stop-inspecting.

Universidad Veracruzana

Stress Tests

- Running a program with parameters and input data outside the normal ranges.
- We can make very clear predictions of how the model should behave under extreme conditions, *e.g.*, $q = 1$ in the butterflies model means they do not wander.
- Environmental conditions outside their normal range can also be used, *e.g.*, unrealistically variable or constant.

Universidad Veracruzana

Test Procedures

- They produce intermediate output (graph, text, or file) used only for testing.
- They may require adding new state variables that are only for observing what is going on.
- They can be invoke by adding buttons or through the command center.

Universidad Veracruzana

Test Programs

- It can be hard to test a certain primitive, procedure or algorithm within a large program.
- Write a separate short program to test the particular programming idea.

Example: Path recall I

- The code:

```
turtles-own [ path ]

to setup
  ca
  crt 1
  [
    set color red ; so we can tell initial path from return
    set path (list patch-here) ; initialize the path list
    pd;

    repeat 100
    [
      rt (random 91 - 45)
      fd 1
      set path fput patch-here path
    ]
  ]
end ; end setup
```

Example: Path recall II

- An the test procedure:

```
1  to go-back
2    ask turtles
3    [
4      set color blue ; turtle returns trace blue
5      foreach path
6      [
7        a-patch -> set heading towards a-patch
8        fd 1
9      ]
10   ]
11 end ; go-back
```

Output

NetLogo — backward-test {/Users/aguerra/Documents/cursos/abms/code/abms-06}

Interface | Info | Code

Edit | Delete | Add | Button | normal speed | ticks: | view updates | continuous | Settings...

setup

go-back

Command Center | Clear

```
turtles> show path
(turtle 0): [(patch 0 3) (patch 1 3) (patch 2 4) (patch 3 5) (patch 4 5) (patch 4 5) (patch 5 4)
turtles>
```

Code Reviews

- Have your code read and reviewed by peers. Building a public code and documentation repository is a good idea.
- Reviewer's tasks include:
  - Compare the code with the model formulation;
  - Look for logical or typographical errors;
  - Identify parts of the formulation left out or misrepresented in the program;
  - Identify important assumptions that are in the code but not in the formulation;
  - Make sure that the code is well organized and easy to understand.

Universidad Veracruzana

Statistical Analysis of File Output

- One of the most systematic way to search for problems.
- Do stochastic events happen with the expected frequency?
- Do variables stay within the expected range?
- These are hypothesis-testing experiments.
- Example: Since the butterflies move randomly with probability $1 - q$ they are supposed to move to the highest neighbor with a frequency of $q + \frac{1-q}{8}$. How do you test it?

Universidad Veracruzana

Example: Testing butterflies movements I

- It requires creating an output file in the `setup` procedure:

```
; Creating output file
if file-exists? "butterfliesTest.csv"
[ file-delete "butterfliesTest.csv"]
file-open "butterfliesTest.csv"
```

- and closing it before stoping the system in th `go` procedure:

```
if ticks >= 1000
[
  file-close
  stop
]
```

Universidad Veracruzana

Example: Testing butterflies movements II

- As well as producing the output in the move procedure:

```
to move ; The butterfly move procedure
  ; write the elevation of all neighbors in the output
  ask neighbors [file-type (word elevation ",")]

  ifelse random-float 1 < q ; q is the probability of moving uphill
      straighfordwardly.
  [ uphill elevation ] ; move deterministically uphill
  [ move-to one-of neighbors ] ; move randolmly around current location.

  ; write the elevation turtle moved to
  file-print elevation
end ; end of move
```

Output Imported in Excel

J2 =SI(I2=MAX(A2:H2),1,0)

	A	B	C	D	E	F	G	H	I	J
1	Elevation of Neighbors								Move to	Highest?
2	14.7721701	15.7655145	15.6139822	14.7239776	15.4955622	16.2623143	15.6343194	15.474647	16.2623143	1
3	14.7239776	16.2623143	14.7721701	15.4955622	15.6139822	15.7655145	15.474647	15.6343194	15.6343194	0
4	15.7655145	15.6139822	14.7721701	14.7239776	16.2623143	15.6343194	15.4955622	15.474647	16.2623143	1
5	15.474647	15.6139822	15.4955622	15.6343194	16.2623143	14.7239776	14.7721701	15.7655145	15.474647	0
6	14.7721701	16.2623143	15.4955622	15.474647	15.6139822	15.6343194	15.7655145	14.7239776	16.2623143	1
7	15.474647	14.7239776	15.6139822	15.6343194	16.2623143	15.7655145	14.7721701	15.4955622	16.2623143	1
8	14.7721701	15.4955622	14.7239776	15.6343194	15.7655145	15.6139822	15.474647	16.2623143	15.7655145	0
9	16.2623143	15.6139822	14.7721701	15.7655145	15.4955622	15.6343194	14.7239776	15.474647	16.2623143	1
10	14.7721701	15.4955622	16.2623143	15.474647	15.6139822	15.7655145	15.6343194	14.7239776	15.6343194	0
11	15.4955622	15.474647	16.2623143	14.7239776	15.7655145	15.6343194	14.7721701	15.6139822	15.7655145	0
12	14.7239776	15.6139822	15.7655145	15.4955622	16.2623143	15.6343194	14.7721701	15.474647	16.2623143	1
13	15.4955622	15.474647	14.7239776	15.6139822	15.6343194	16.2623143	15.7655145	14.7721701	15.7655145	0
14	15.6343194	15.6139822	15.7655145	14.7721701	16.2623143	14.7239776	15.4955622	15.474647	14.7239776	0
15	15.7655145	15.474647	14.7721701	14.7239776	16.2623143	15.6343194	15.4955622	15.6139822	14.7721701	0
16	14.7239776	15.6343194	15.7655145	16.2623143	14.7721701	15.4955622	15.6139822	15.474647	16.2623143	1
17	15.6343194	15.4955622	15.474647	16.2623143	14.7239776	15.7655145	14.7721701	15.6139822	16.2623143	1
18	16.2623143	15.6343194	15.6139822	14.7721701	14.7239776	15.7655145	15.474647	15.4955622	16.2623143	1
19	15.7655145	15.6139822	15.4955622	15.474647	15.6343194	14.7721701	14.7239776	16.2623143	16.2623143	1
20	15.6139822	15.474647	15.4955622	16.2623143	14.7721701	15.7655145	14.7239776	15.6343194	15.6343194	0
21	14.7239776	15.474647	15.4955622	15.6343194	15.7655145	15.6139822	14.7721701	16.2623143	16.2623143	1
22	15.6139822	14.7721701	15.4955622	16.2623143	15.474647	14.7239776	15.6343194	15.7655145	15.6343194	0
23	15.6343194	14.7721701	14.7239776	15.474647	15.6139822	15.4955622	15.7655145	16.2623143	14.7239776	0
24	15.6139822	15.7655145	14.7721701	16.2623143	15.4955622	15.6343194	14.7239776	15.474647	15.4955622	0
25	15.474647	15.4955622	15.7655145	14.7239776	16.2623143	15.6139822	15.6343194	14.7721701	15.474647	0
26	15.4955622	14.7721701	15.474647	15.6343194	15.6139822	16.2623143	15.7655145	14.7239776	16.2623143	1
27	15.4955622	16.2623143	15.474647	14.7721701	15.7655145	15.6343194	14.7239776	15.6139822	15.7655145	0
28	15.6343194	16.2623143	15.6139822	15.7655145	15.474647	14.7239776	14.7721701	15.4955622	16.2623143	1
29	15.474647	15.4955622	14.7721701	14.7239776	15.7655145	16.2623143	15.6139822	15.6343194	15.6139822	0
30	14.7721701	15.7655145	15.6343194	16.2623143	15.6139822	15.474647	14.7239776	15.4955622	16.2623143	1
31	15.6343194	14.7239776	15.7655145	16.2623143	14.7721701	15.4955622	15.6139822	15.474647	15.7655145	0

Independent Reimplementation of Submodels

- The only reliable way to verify software for any model is to program the model at least twice, independently, and test if the same results are obtained for all the implementations.
- Problem: Cost!
- At least reimplement key submodels in your language of preference (other than NetLogo).
- Example. Following the previous examples, implement the butterfly decision in the spread-sheet for comparison. Surprise, surprise, it is not the same:

```
1 [ uphill elevation ]
```

than

```
1 [ move-to max-one-of neighbors [ elevation ] ]
```

Universidad Veracruzana

Documentation of Tests

- Document only the most important tests and debugging as the model is programmed.
- But once it is ready, conduct and document a conclusive set of final tests.
- Reasons to document:
  - Gaining credibility by showing that the model was tested adequately;
  - To help youself to repeat the tests.
- Recall the TRACE format [3].

Universidad Veracruzana

Description of Tests

- Documentation should describe:
  - The kinds of tests that were used; and
  - The methods and results of the comprehensive tests of the "final" model .
- As simple as listing:
  - Who delivered the code, at what stage(s) in its development;
  - What patterns were observed visually and investigated;
  - The parts of the model that were tested statistically against an independent implementation; and
  - The kinds of errors found and how they were corrected.

Universidad Veracruzana

Purpose and patterns

- Axelrod [1] proposed this model to show the consequences of a few simple assumptions about how people (or groups) are influenced by those around them.
- The model assumes that people or societies share culture locally, and share more with others that are more similar to themselves.
- The models intends to explore one general pattern: that, even if people tend to become more alike when they interact, differences in culture persist over time.

Entities, state variables, and scales

- Agents can be thought as homogeneous villages represented by sites (patches) on the grid.
- The sites have state variables for each of five cultural features.
- Each feature has a value that is an integer between 0 and 9.
- A site's culture is defined concatenating these values as a five-digit string, *e.g.*, 93452.
- The grid of sites is 10×10 patches in extent, with the space not wrapped.
- Time and distance is not defined explicitly.
- Model runs continue until the system is stable, *i.e.*, executing 1000 ticks with no change in the site state variables (which can take approx. 80K ticks).

Universidad Veracruzana

Process overview and scheduling

- The model includes two actions executed each time step:
  - **Cultural interaction.** Executed by one random chosen site. This site randomly chooses a neighbor to the north, east, south, or west. The site calculates how similar is this neighbor in culture, and then stochastically decides whether to interact by adopting one cultural feature value of the neighbor.
  - **Output.** Several measures of how similar individuals are to their neighbors, and how homogeneous the entire model is, are updated.

Universidad Veracruzana

Technical novelties I

- The primitive `myself`, see the Dictionary to understand this reporter and its difference with `self`.
- Example. `ask turtles-here [set color [color] of myself]`.
- The primitive `neighbors4` as an alternative to `neighbors`.
- The logical expression `!=` to check whether two variables are not the same.
- Creation and concatenation of string variables to create output.
- The primitive `and` to create a conjunction of boolean expressions.
- Local variables that contain an agent, using a name (*e.g.*, `the-neighbor`) that makes the code easier to understand.
- Example. `let the-neighbor one-of neighbors4`

Technical novelties II

- A test procedure that can be executed form an Agent Monitor.
- A histogram plot.
- A switch on the Interface, which controls a boolean global variable, *i.e.*, to produce or not a set of test output files.

Universidad Veracruzana

Testing the Software

- The version provided by our book's web site has several error that you should try to find as you try the testing approaches on it.
- Some errors are obvious but others are not at all.
- These steps are suggested to find and report the errors:
    1. Review the code. Read the model description in the Info tab and compare the code to it.
    2. Run the model and watch it no the World display. Look for unusual patterns and think why they might occur if the model is or is not implemented correctly.
    3. Conduct spot tests of the cultural similarity calculations using some test procedure.
    4. Use the optional file output to test the key procedures. Find ways to test whether the code works correctly.

Conclusions

- Not all programming errors have effects on the ultimate use of a model, *e.g.*, those in the butterflies model have little effect on any general conclusion drawn from the model.
- Unfortunately, the possibility of the opposite makes software verification very important.
- However, the approaches you are now using to find mistakes are also important when it is time to analyze and understand your model.
- Paying serious attention to software reliability and reproducibility is, therefore, just part of the job when we are doing scientific modeling instead of just playing around with NetLogo.

Key strategies

- Find your mistakes early, not late. Make testing a pervasive part of software development.
- Plan for testing: make sure there is time for it, and time to repeat it as the model evolves.
- Write your code so it is clear and easy to understand, so you never have to hesitate before asking someone to review it for you.
- Save and document your tests, assuming that you will need to repeat them as the code evolves and that you will need to prove that your software is reliable.

Referencias I

[1] R Axelrod. "The Dissemination of Culture: A Model with Local Convergence and Global Polarization". In: *The Journal of Conflict Resolution* 41.2 (1997), pp. 203–226.

[2] SF Railsback and V Grimm. *Agent-Based and Individual-Based Modeling*. Second. Princeton, NJ, USA: Princeton University Press, 2019.

[3] A Schmolke et al. "Ecological models supporting environmental decision making: a strategy for the future". In: *Trends in ecology & evolution* 25.8 (2010), pp. 479–486.