Juan Carlos García Vázquez
Fernando Sancho Caparrini

NetLogo

A modeling tool

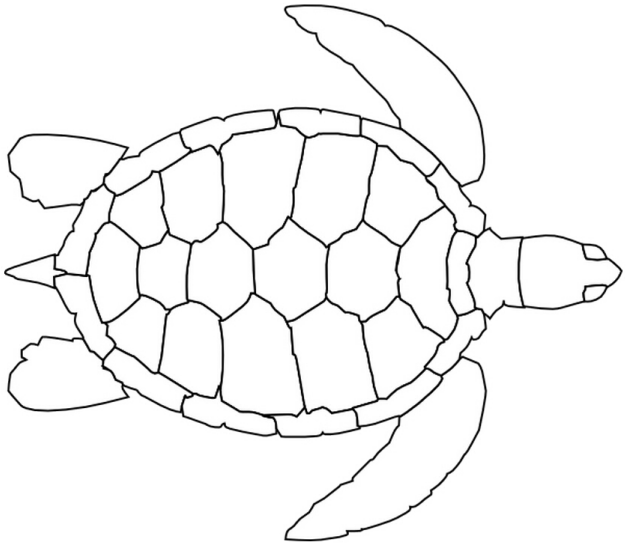
Acknowledgements

To me, for being the creator of everything.
The Observer

To the patches, for being always right there.
The Turtles

To the turtles, to lighten our paralysis with their adventures.
The Patches

To the turtles, our beginning and end.
The Links



To the turtle that all children have had.

Prologue

When, some time ago, we considered using NetLogo to complete the academic development of our students, we did this undoubtedly due to the good results that it was giving us, in our role as researchers.

Our experience of using this software has been as diverse as NetLogo itself allows: we have been using it for years on the ESTALMAT project[1] to foster in children a way of tackling problems based on algorithmic solutions that a flexible programming language can offer; on different university courses to introduce concepts covering such seemingly disparate areas as complex systems, artificial intelligence, biological modeling, cultural modeling, mathematical and physical experimentation, etc. at both undergraduate and master's level, in the process of modeling complex behaviors in a scientific production environment; to develop tools for analyzing and representing complex information in digital humanities projects[2]; as algorithm prototyping to analyze and predict socioeconomic behaviors, etc. And in all of these experiences, the results have proven to be fully satisfactory.

Our students on the various courses and those who have collaborated in the research work cover areas of knowledge that are supposedly dispersed and incompatible: ranging from humanists who are interested in analyzing literary texts or mathematicians engrossed in their abstract models, to ecologists and biologists who are interested in the evolution of a particular population of species or organisms, or computer programmers interested in implementing a distributed algorithm to establish its validity. And with all of these we have found that, after an initial moment of skepticism surrounding the "play tool" aspect involved, NetLogo enabled them to enter into a direct conversation with their problem, mitigating to a large extent the harsh reality that there must be an artificial device on the computer between their ideas and their projection, obliging them to formalize their knowledge in a completely new way that is alien to them.

This is perhaps the best thing that can be said of a tool of this type: NetLogo becomes a fine, almost transparent, layer between the individual and their subject matter, and can soon end up forming a natural part of their regular tools.

[1] This project, named after the Spanish acronym for Encouraging Mathematical Talent, aims to identify children with an aptitude for mathematics (which normally coincides with scientific skills in general) and to encourage them to learn mathematical methods, techniques and strategies. Further information can be found at *http://thales.cica.es/estalmat/*

[2] All of these applications are to be found within the Hispanic Baroque project (*http://www.hispanicbaroque.ca/*), funded by the SSHRC in Canada and led by Juan Luis Suárez, a Professor at the University of Western Ontario and Director of the CulturePlex laboratory, a center for the study of cultural complexity with a focus that falls within the field of Digital Humanities (*http://cultureplex.ca/*).

Our view has been confirmed by the wide range of similar experiences that we have come across through other academics and researchers. Netlogo is one of the few development platforms with such a wide range of users, both in terms of the academic levels they cover and the goals they are pursuing.

On an educational level, thanks to its proximity to natural language, it was originally used for teaching basic programming concepts to children and young people of very different ages; although this use still continues, we now find more and more university courses and higher education training modules in which this tool is used to introduce general modeling and problem-solving concepts in different areas of knowledge.

In the context of academic research, we find it in the process of generating and verifying results from scientific projects, with references to its use in specialist academic articles becoming increasingly common.

On a more professional level, there are numerous cases in which NetLogo has been used as a tool for validating experiments on real projects, many of these in the area of ecological modeling and group dynamics.

This diversity of use certainly gives us an idea of its versatility and potential, as well as its simplicity and appropriate learning curve.

In spite of its limitations with respect to other more powerful and lower-level languages, the programmer using Netlogo as an experimentation system will find many advantages during the development and maintenance of the solutions they prepare on the platform. Prechelt[3] showed that the number of lines of code that a programmer can write per unit of time is independent of the language used. Consequently, given that the faster and more powerful low-level languages require a higher number of lines in order to perform the same task, scientific programming aimed at discovery should make use of languages of the highest level possible that are capable of meeting their needs, and only move to more efficient languages, but which cost more to develop (such as C++ or Fortran), when there is certainty that an improved performance is essential, translating the model from one language to another. This initial choice enables the programmer to write more code, and to check it, in the same amount of time, and even in the knowledge that a low-level language will be needed in order to obtain final results; fast prototyping in a high-level language such as NetLogo can help modelers to evaluate design decisions more quickly, reducing the overall development time and improving the knowledge of the problem. And so the choice of NetLogo as a working and exploration tool for students and researchers seems appropriate and wholly justified.

However, when we decided to write down our experiences of Net-

[3] Lutz Prechelt. **Two Comparisons of Programming Languages**. Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*. O'Reilly, 2010.

Logo in the form of a book with a view to helping others to become familiar with its use, we realized that it would not be sufficient to project these as a series of problems and solutions making use of the language; leaning on the experience gained over previous years, we decided that we would need to start by taking a step back and to engage in the creation of a programming guide for users who, in many cases, had no experience of other programming languages[4].

We have therefore attempted to write an ambitious book with the aim of meeting the needs of readers who are seeing NetLogo for the first time, perhaps with no prior knowledge of programming, perhaps with no prior knowledge of modeling, and at the same time going into less apparent detail, for those who have already had some experience of this tool. We hope that we have managed to achieve our goal and that we do not disappoint.

[4] Although the tutorial provided by the creators of NetLogo on their website is very comprehensive and indispensable to every user, our students and collaborators were demanding a more structured learning process.

How the book is structured

The first few chapters of the book, approximately chapters 1 to 6, are dedicated to an introduction to the tool, to the concepts on which it is based and to the basics of programming required in order to start developing solutions in it.

Then, in chapters 7 to 12, we introduce increasingly elaborate uses of its capacities, covering the whole spectrum of functionalities that a normal modeling project may require.

The third part of the book, chapters 13 to 15, is dedicated to describing certain advanced features of the tool, some of which will only be of use in specific cases, but which are worth covering in order to broaden the limits of our experience with the tool.

Finally, we close the book with a chapter in which 8 complete projects are set and which can be developed with NetLogo, similar to those we can find in professional and academic projects, and which can be extended through different channels.

Throughout each chapter, the reader will find a series of activities included, the aim of these being to help them grasp the concepts and methods explained up to that point and to make it easier for them to understand the next steps in the book. Along with these activities, in each chapter we set a group of exercises that can be solved by using the resources explained in it (up to a total of over 120 exercises).

Acknowledgements

We must start by thanking Elena Varela. If there is anything in this book worth reading it is due to her meticulous efforts in proofreading our drafts. Any errors that can be found are without doubt a result of

our inability to express the ideas following her advice; her work has undoubtedly improved the book that is now in your hands, but she has also taught us how difficult and beautiful it can be to write a text in which the clarity of ideas is based on the good use of language. This book would certainly have a completely different appearance without Elena's invaluable help.

We would, of course, like to thank Juan Luis Suárez (and the projects led by him) for various complementary reasons, because he is a friend who always supports and motivates us with problems that are worth addressing, and because he has been encouraging us for some time to write something similar to the outcome of this book. We cannot forget that we remain in his debt (an old debt which is now growing) and we remind him in these pages that we have one project outstanding, perhaps a continuation of this one, to show the applications of modeling with NetLogo for the analysis of culture.

We would like to thank Judy Kerry for translating the book into English and for the (numerous) misprints that she has found in the Spanish version during the translation process. Her patience in dealing with the source code of the book (in LaTeX), with the Spanish texts and with our working methods seems to have no limits.

We cannot forget the different readers who have been proofreading the book in its early versions, even when it was no more than just a few poorly drafted pages with no clearly-defined objective. Of these, although we have not forgotten any of them, we would like to highlight the pertinent comments and the support of both Antonio Pérez, from the University of Seville, and Marta Ginovart, from the Polytechnic University of Barcelona.

Contents

1

An overview of NetLogo. Its background and environment.

We need to begin this book with a quick look at what it means to model, at the role it plays in the process of experimenting with and representing phenomena, and at the difference that it can make to have a suitable tool available to facilitate the transition from the real world in which the problem exists to the formal world in which we are able to understand it and, in the best-case scenario, solve it.

Modeling and solving problems

Human knowledge is mainly based on the observation and analysis of real-world phenomena in the broadest sense: from the sensory experiences of the natural world, which relate to the applied areas, to the most abstract conceptualizations, the subject of theoretical areas. Even though the distance between some of the phenomena is seemingly unassailable, the methodologies used to understand and predict their behavior are surprisingly similar. At the root of this is the formalization during the 17th century of what we know as the scientific method; its advantages: it ensures the robustness of the conclusions obtained, can be applied to all areas of knowledge (science in particular) and is independent of the size of the phenomenon observed. Although there are variations depending on the historical point in time and the objectives, the scientific method is developed in four stages:

- observation: the stage involving the careful examination of all of the facts and phenomena that take place in the real world and which

can be perceived by the senses and measured;

- formulation of hypotheses: in which a provisional explanation of the phenomena observed and its possible causes is formed;

- experimentation: in which the phenomenon being studied is repeatedly reproduced and observed, modifying the circumstances as deemed necessary and thus confirming what circumstances cause it to occur;

- thesis or generation of a scientific theory: in which the phenomena observed are interpreted according to the test data.

If the hypothesis is found to be false, it must be rejected once the data that invalidate it are provided. The likelihood of success is greatly increased the larger the number of examples or cases that enable the phenomenon studied to be illustrated. In many cases, the experience is enhanced by the observation and manipulation of the real-world phenomenon; in many others, this option is physically impossible or inadvisable, perhaps because the phenomenon is unfeasible in terms of time or space, because it is based on concepts that cannot be manipulated, or for ethical reasons. It is precisely those cases in which the manipulation of reality in order to reconstruct and test hypotheses is vetoed, that have obliged the researcher to resort to the use of replicas or models. In this book we will be using the following definition of model:

> *A model is an abstract representation of a certain aspect of reality, essentially formed by the elements that characterize this aspect of reality and by the relationships that these elements maintain.*

Of all the options available for creating models, there is one that stands out for its effectiveness: mathematical modeling. It is displayed in such disparate forms as mathematical theories (based on axiomatic systems and demonstrations, etc.), numerical models (based on linear equations, differential equations, stochastic calculus, etc.) or computational models (based on systems of agents and their interactions, particle systems, evolutionary algorithms, etc.) which have undergone spectacular development over recent years due to the emerging technological capacity.

In all of its possible forms, some of the advantages offered by the construction of mathematical models are:

1. the modeling process often reveals relationships which are not apparent at first sight: the act of modeling is often accompanied by a greater understanding of the phenomenon being represented;

2. once the model has been constructed, it is possible to discover properties and characteristics of the relationships between the elements of the model which would otherwise remain hidden;

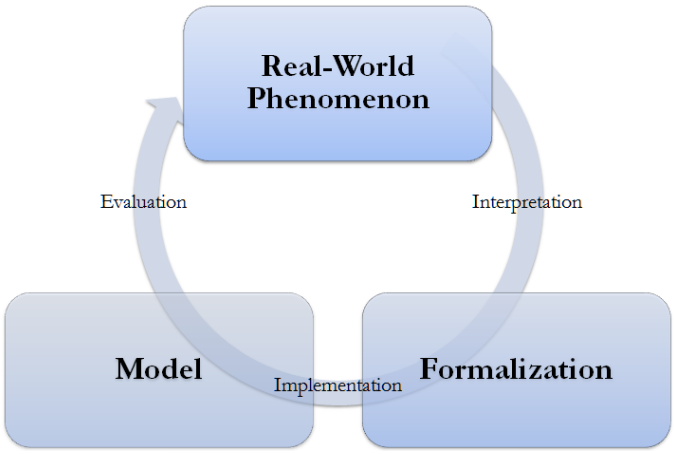
3. it is possible to simulate complex situations that cannot be represented in other types of models;

4. they often provide a solution to the problem, even if it is not an analytical solution, but a numerical solution, created by computer.

Models can be classified in many ways according to the characteristic to be highlighted:

- depending on the way in which the elements of the model are represented, we can talk about iconic, analogue, symbolic models, etc.
- depending on their function, they can be predictive, evaluation, or optimization models[1], etc.;
- depending on the type of reality you want to model, we can talk about deterministic models as opposed to stochastic models[2];
- depending on the way in which time is represented, we distinguish between static models and dynamic models[3].

In general, a good model is one which adjusts to the real-world phenomenon so as to enable us to gain a better understanding of its properties and to broaden our knowledge of it. From the experience gained in various disciplines that use modeling as an essential tool, the following phases are important for building a good model:



1. From the real-world phenomenon, following a phase of interpreting the observations, you enter another phase, the formalization phase (using a formal language).

[1] Predictive models inform us about the behavior of its parameters in the future. Evaluation models aim to measure the different alternatives, in order to compare their results. And optimization models try to identify an optimum solution to the problem, i.e., the best of the possible alternatives.

[2] In deterministic models, the future evolution of the model is entirely determined by its current state, whereas in stochastic models, this evolution can depend on random behaviors that can be measured by statistical means.

[3] There is no temporal evolution in static models, unlike in dynamic models.

2. If the model you want to achieve has dynamic properties or allows a simulation process to help you to understand the reality, you then implement this formalization in a system that will enable you to consider this dynamism.

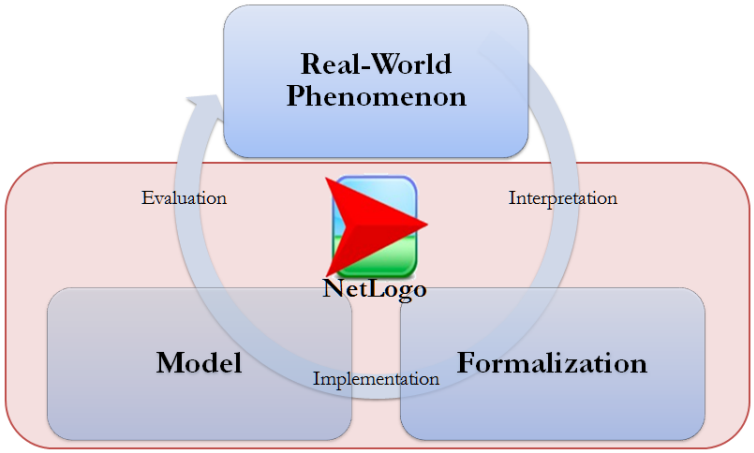
3. Finally, and once you have executed simulations in the model, you evaluate the results obtained by comparing them with those seen in the real world.

You must bear in the mind that the norm is not to carry out these steps according to a linear or direct process, but a life cycle is formed, in which, through successive approaches, you obtain a set of models that are increasingly true (at least, in terms of the characteristics that are the focus of your hypothesis) to the real-world phenomenon.

Due to its importance as a research tool and taking into account the phases required in order to achieve an accurate model, there is no doubt that modeling is an activity that lies at the heart of the process of creating and gaining knowledge, both on an educational level and on a research level[4]: being able to model a problem is proof of having understood it correctly, as well as a demonstration of having the necessary technical tools to represent it in a formal system.

[4] Which should be no surprise, as the research process is no more than a continuation of the education process, based on discovery, and in which the figure of the teacher and disciple are often the same person.

The aim of this book is to provide you with the necessary basics in order to enjoy the autonomy to generate models that will help you to propose and test plausible hypotheses. You will see how Netlogo gives you the great advantage of offering in a single block a tool capable of addressing practically the whole modeling process that we have described.



As it is a programming language, the implementation process is natural in NetLogo, although, due to the type of elements with which it works, it will also influence the process of conceptualizing and formalizing the real-world phenomenon. The platform features mean

that the first part of the process is simplified considerably; but it goes further and enables you to evaluate the results of its execution in order to compare them with the real world by means of the automatic generation of experiments and their execution.

What is NetLogo?

The best definition of this tool has been provided by the creators themselves[5]. As explained in the online manual, NetLogo:

- is a programmable modeling environment for simulating natural and social phenomena.
- is simple enough for students and teachers, yet advanced enough to serve as a powerful tool for researchers in many fields.
- lets students open simulations and*play* with them, exploring their behavior under various conditions. It is also an authoring environment which enables students, teachers and curriculum developers to create their own models.
- is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of *agents*[6] all operating independently. This makes it possible to explore the connection between the (**micro** level) behavior of individuals and the (**macro** level) patterns that emerge from their interaction.
- is the next generation of the series of *multi-agent*[7] modeling languages, including **StarLogo** and **StarLogo TNG**.
- has extensive documentation and tutorials. It also comes with the **Models Library**, a large collection of pre-written simulations that can be used and modified. These simulations address content areas in the natural and social sciences including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. Several model-based inquiry curricula using NetLogo are available and more are under development.
- can also power a classroom participatory-simulation tool called **HubNet**[8] which is built in, enabling each student to control an agent in a simulation.
- runs on the Java virtual machine (JVM), so it works on all major platforms (Mac, Windows, Linux, et al). It is run as a standalone application. Models and HubNet activities can be run as Java applets in a web browser. Command line operation is also supported.

[5] It was created in 1999 by Uri Wilensky and has been under constant development ever since at the *Center for Connected Learning and Computer-Based Modeling*, at Northwestern University, Evanston, Illinois.

[6] For now, you can just think of an *agent* as an individual that is capable of working autonomously and which interacts with its environment and other agents.

[7] *Multi-agent* means that it does not work with just one individual, but with a group of individuals.

[8] Through the use of networked computers or handheld devices. Currently, it only supports the graphing calculators from Texas Instruments as an example of a mobile device, but its adaptation to tablets and smart phones is under way.

It must be added that NetLogo is now a free software project that is being maintained by a growing community, led by *Seth Tisue*, with a continuous development process and in which there is large amount of collaboration on all levels (from the creation of models for the rest of the community to the creation of extensions or updates to the core application). At the time of writing this book, its code and work plans can be found in a repository on Github[9].

[9] https://github.com/NetLogo

From Logo to NetLogo

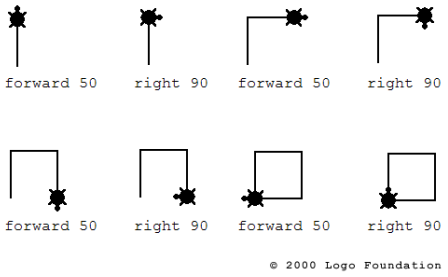
Figure 1.1: Using turtle hardware to draw.

Logo is a high-level programming language created in the 1960s and based on **Lisp**. It is very easy to learn and so has long been used to introduce children and young people to the principles of programming (figure 1.1).

However, in spite of these educational purposes, we should not think of it as a poor language, as it has all the necessary features to practice and implement all the usual concepts of advanced functional programming[10].

[10] Such as: recursion, advanced data structures, higher-order functions (i.e., functions that operate on other functions), etc.

Certainly, one of the most recognizable features of Logo is its ability to manage small *graphic objects* which enable geometric shapes to be formed by following basic instructions, and which, as we can see in figure 1.1, have sometimes even adopted a physical form.



Although we actually talk about Logo as *a language*, it corresponds to a family of languages that are similar to one another and there is no standard definition for it. NetLogo shares enough characteristics with this family of languages to be considered as one more of them, albeit with certain key differences which we will outline below[11]:

- The precedence of mathematical operators is different. The infix operators (those written between the operands, such as `+`, `*`, etc.) have less precedence than the named functions. For example, `sin x + 1 = (sin x) + 1`.
- The functions `and` and `or` are special functions used as a shortcut for the calculation, which means that they evaluate the second argument only if necessary[12]. This characteristic means that the evaluation of strings of conditionals is more efficient.

[11] If you have never used any version of Logo you can skip the following and go straight to the next section, then wait until later, when you know a bit about NetLogo, to come back to this.

[12] With the operator `or`, once an argument is found to be true, it does not evaluate the others because the overall result is true. With the operator `and`, once an argument is found to be false, it does not evaluate the others because the overall result is false.

- Procedures can only be defined in the **Code** tab, and not interactively in the **Command Center**. We will be looking at these two elements in detail in the following chapters.

- The functions that return values must be defined with the keyword **`to-report`**, and in these cases, the command to return a value must be **`report`**.

- When defining a procedure that accepts input data, these must appear between square brackets.

- The names of the variables must always be chosen starting without punctuation marks, and so the uses `:var` or `"var`, so common in other Logos, are not valid.

- The local variables and the input data are verified lexically, rather than dynamically (i.e., if there is an error, it is highlighted before execution).

- NetLogo does not have the data type `word` (also known as `symbols`), but has the type `string`[13].

- The **`Tasks`** (or *lambda* functions[14]) provide a new feature which is now normal in modern Lisp languages, but not in the classic Logos.

- The control structures, such as **`if`** or **`while`**, are special forms, and not ordinary functions. Special forms cannot be defined, and so no new control structures can be created[15].

- The NetLogo command **`run`** acts on **`tasks`** and strings, but not on lists, and does not allow procedures to be defined or redefined.

And, of course, NetLogo gives us a fundamental capacity that is not found in most of the other languages of the Logo family: it works directly with agents and is capable of differentiating *families of agents* in order to design individualized behaviors for each family.

Installing and running NetLogo

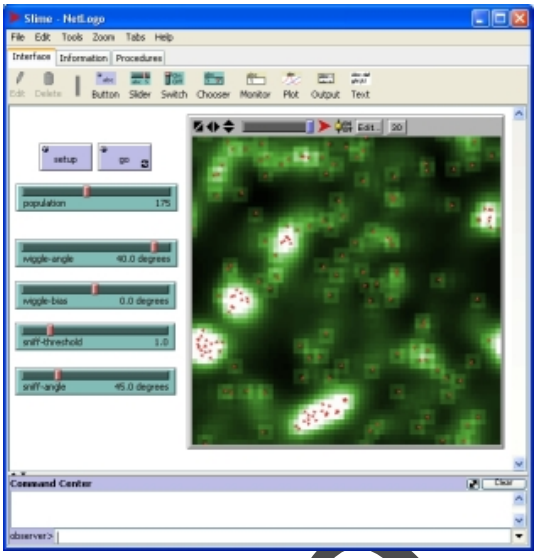
As NetLogo runs on the Java virtual machine and enables you to save models as Java **applets** (deprecated), there are two ways of running them:

1. With the program installed, open the model from the application.

[13] For example, what in Logo would be `[a phrase]` (a list of words), in NetLogo it should be `"a phrase"` (a string) or rather `["a" "phrase"]` (a list of strings).

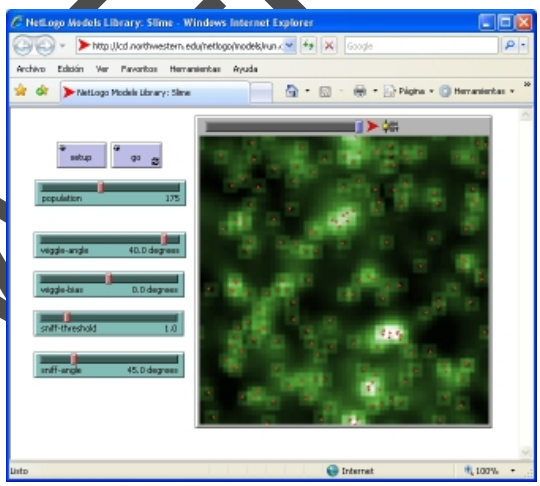
[14] A *lambda* function is an anonymous function, which is not given a name, and which can be treated like any other program data.

[15] Part of this functionality can be achieved by means of the **tasks**, as we will see later.



[16] In addition to the actual restrictions imposed by security measures when running applets, some extensions may not work in applet mode.

2. Through a web browser by accessing (locally or on an Internet server) the file associated with the applet. This method has certain limitations if you want to make use of special functionalities[16], but it accepts almost all of the usual examples.



Whilst both are valid for running the models interactively and using the interface, there are some key differences between them. The most obvious one is that, whereas when running the applet you can only access the view of the model interface (and, therefore, cannot make any modifications), when running it from the complete program, not only do you access this interactive view, but you can access the code tab (which enables you to modify the model) and the additional tools provided in the application and which we will be looking at later on in this chapter.

There is, in fact, a third way of running NetLogo models, or even of working with it, which is what is known as running **headless**. This

does not load the NetLogo graphical interface but it does enable you to fully run all of its functionality as a programming language and runtime engine. Also, taking into account the fact that it is run completely in Java, it can be used as a library for the development of applications which, when necessary, make use of the NetLogo engine through its API[17].

For most of the content of this book we will be using the full application, and so it is advisable to install it by following the steps as indicated on the NetLogo official website.

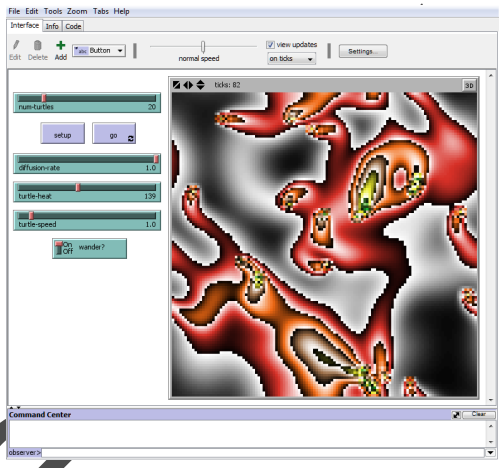
[17] API (**Application Programming Interface**) is the set of functions and procedures that offer certain libraries for use by other software.

A quick look at the work environment

Following installation, you will find several applications in the folder alongside **NetLogo x.x.x**[18], which are **NetLogo 3D**, a version that enables you to work with 3D models, and **Hubnet**, which we will explain briefly at the end of this chapter and in more detail in the relevant chapter.

[18] The numbers depend on the version. Version 5.3.0 is the stable version at the time of writing this text.

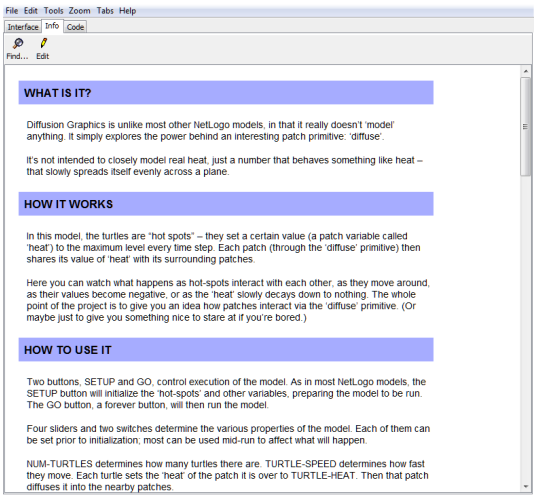
When you open the application, the first thing you will notice is that there are three tabs containing the main sections on which you will be defining your models:



1. **Interface tab**: displaying the world in which the the simulations are run, and where you can define the interface that will enable the user to interact with the model.

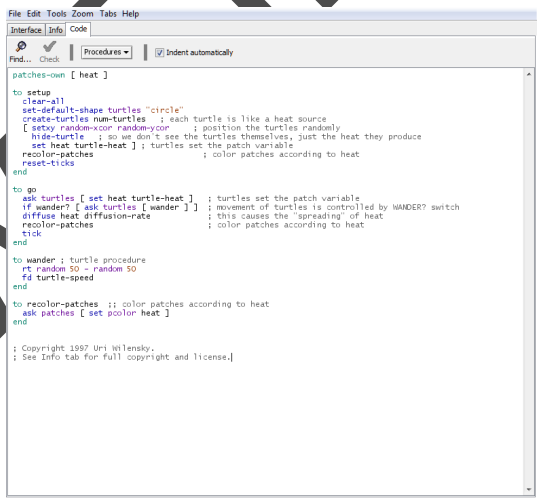
2. **Info tab**: where you can add information relating to your model in order to clarify programming concepts, explain how they work, the real-world model it is simulating, etc.[19]

[19] The editor it provides is very simple but by using a format similar to that of wikis, you can obtain similar results to those of any HTML page.



3. **Code tab**: where you write the procedures and statements that define the behavior of the model[20].

[20] It provides the minimum tools required for editing code correctly.

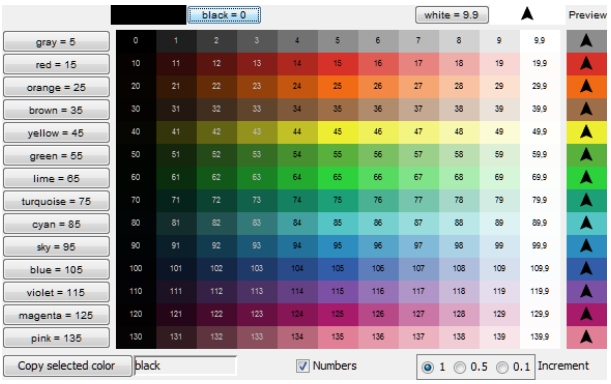


As is normal in all programs working with windows, in the top menu you will find, as well as the usual options of **Files**, **Edit** and **Help**, the **Tools** menu through which you can access the additional tools that complement the experience with NetLogo.

Some of the most important ones are:

- **Color Swatches**: Enables you to browse the color system used by NetLogo[21]. Also, when you access it from certain sections of the program, it can be used to select colors interactively.

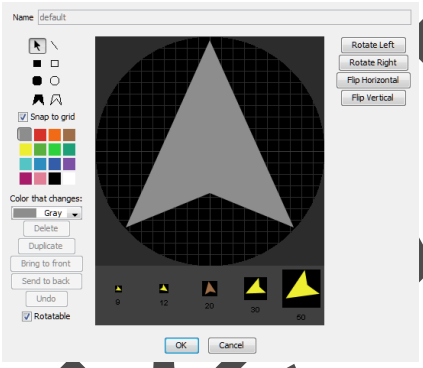
[21] Later we will look at the particular features of this system.



ACTIVITY: Open this tool and familiarize yourself with the NetLogo color system. Change the resolution of the swatches using the control `Increment` and note the contrast of the chosen color against the black and white backgrounds.

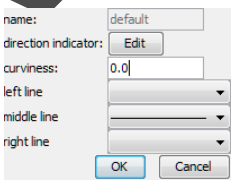
- **Turtle Shapes Editor**: Enables you to edit the appearance[22] of the turtles on the screen so that how they are represented makes it easier to recognize what type of turtle it is (and with it, its possible functioning and behavior). These shapes are stored in each model, and the system also enables you to import shapes from other models that have already been created and from a library provided with the program.

[22] This appearance is defined by vectorial shapes, so that they can be scaled without loss of resolution.



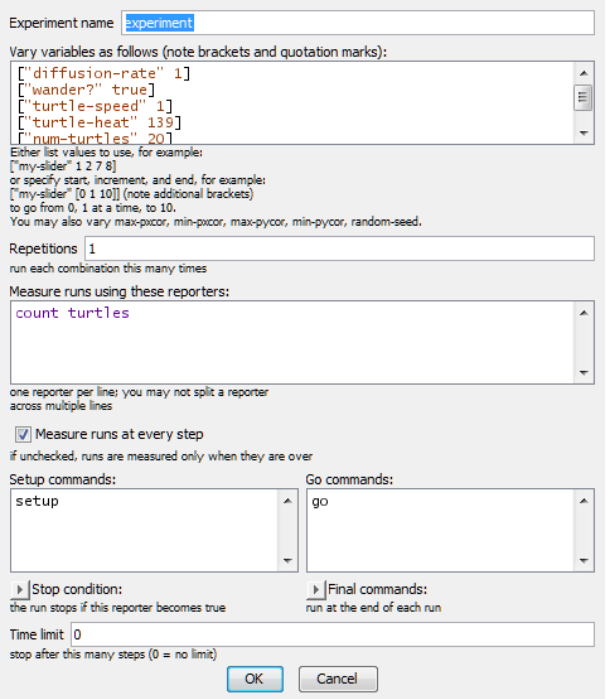
ACTIVITY: Load some shapes in the editor to see how they are formed and create some of your own. Focus on the different simple shapes that can be used and the use of the basic color palette.

- **Link Shapes Editor**: Enables you to edit the appearance of the links that can connect the turtles to one another, with parameters to control their curvature and the visual characteristics of the lines and indicators.



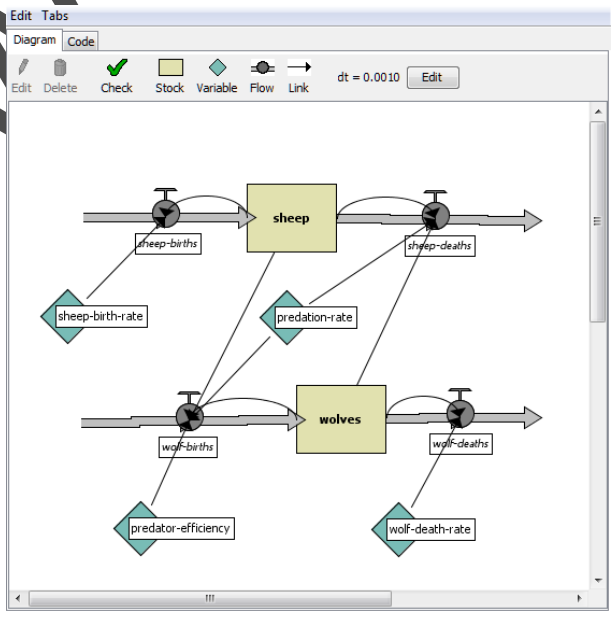
- **BehaviorSpace**: Enables you to define and run sets of experiments on the model and to automatically record all of the results for subsequent analysis using external tools[23].

[23] Furthermore, if you have a multiprocessor computer, this will enable you to create real parallelism in its execution, running separate experiments on each of them.

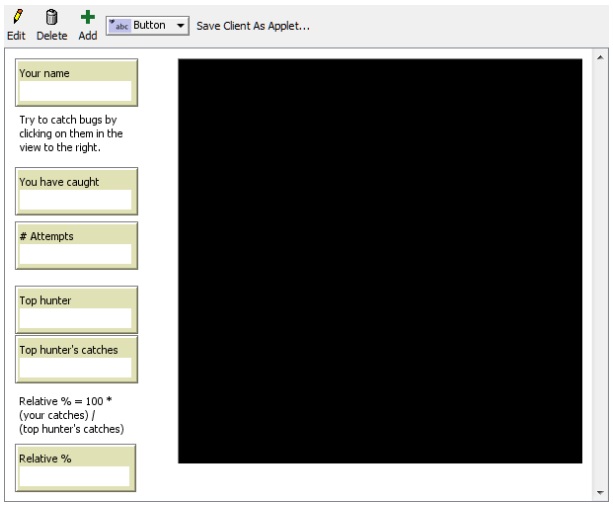


- **System Dynamics Modeler**: Opens a functionality of NetLogo that enables you to design a NetLogo model using regular standardized techniques of system dynamics [24].

[24] This is currently rarely used, although it is a functionality that can be very useful for certain types of problem.



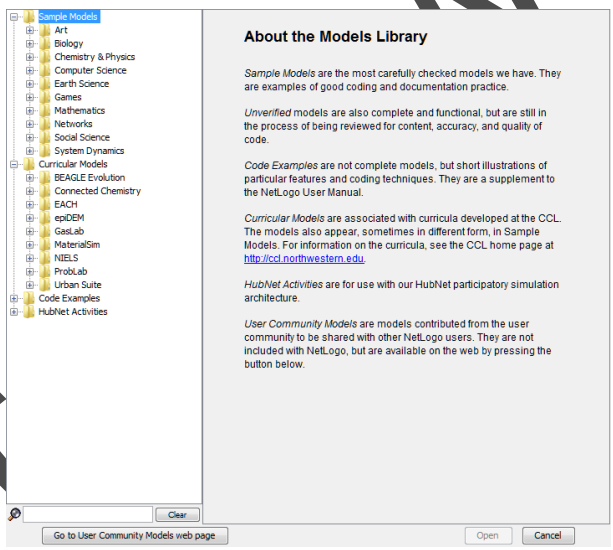
- **HubNet Client Editor**: Opens the editor of client interfaces for collaborative experiments that can be run through **HubNet**.



The Models Library

As mentioned earlier, one of the most useful features of NetLogo is the number of complete examples provided by default on installation (all of these ready to be run and modified), and covering almost any area of knowledge in which the modeling of experiments would be meaningful.

Through the **File** menu (figure 1.2) you can access the **Models Library**, which shows you a folder tree with the models grouped according to very clear criteria:



- **Sample Models**: Contains the models classified according to the area of knowledge to which they belong (Art, Biology, Chemistry and Physics, Computer Science, etc.)

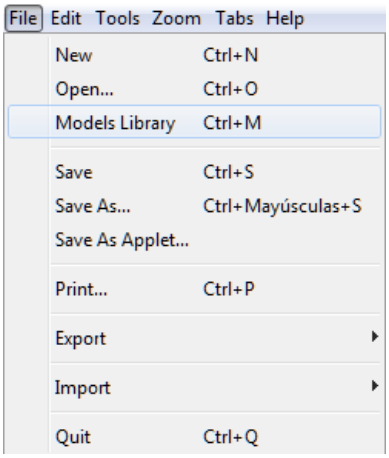


Figure 1.2: File menu.

- **Curricular Models**: Groups together those models that can be framed within a particular curricular project. These curricular projects do not necessarily fall within one specific area of knowledge, but tend to be cross-disciplinary and normally address specific projects that are being undertaken at the *Center for Connected Learning and Computer-Based Modeling* (the same center at which NetLogo is based)[25].
- **Code Samples**: provides certain models whose purpose is to highlight, and practice, some of the characteristics of the programming language, and so they can be used as small samples to look at certain key concepts of the language in more detail.
- **HubNet Activities**: shows certain models that are ready for use as collaborative activities through the HubNet tool in NetLogo.

[25] Some of the models shown under this classification may also appear in the previous classification under one of the areas of knowledge.

We suggest that you start off by exploring extensively the models provided in the **Sample Models** folder, both to see the diversity of possible fields of application and to get an idea of the potential of what can be developed by this small programming language without too much effort.

ACTIVITY: To do this, open the model, launch the simulations and interact with the controls made available on the interface; then go to the info tab to find out more about the purpose of the model, to learn a bit about how it has been programmed and to explore the possible extensions that are offered to look more deeply into both the real-world model it is trying to explain, and the programming techniques that can be used to address it; then go to the code tab to see how it has been programmed. Even if you know nothing yet about the programming language, try to read it and you will be surprised to see how easy it is to understand the code written in NetLogo.

Online resources

As mentioned earlier, one of the finest qualities of NetLogo is its user community (which has led, after a considerable amount of work, to the complete reprogramming of the tool to turn it into an open source application), and which has resulted in a large number of resources becoming available online.

We are now going to list only those which show most activity, are updated most frequently, or offer the most reliable content:

- The first essential resource is the official website of NetLogo itself[26], from where you can access a vast collection of external resources (some of which we are listing here).

[26] *http://ccl.northwestern.edu/netlogo/*

- On the official website you will find the online manual[27] (which is also installed locally along with the application and can be accessed from the **Help** menu) which contains a complete reference manual of the language, an introduction to the application and some basic tutorials to familiarize yourself with its use.

- There is also a NetLogo User Group (housed on Yahoo Groups[28]) where you can share your queries and ideas with other users in the community, as well as a NetLogo Users Group within the field of education[29].

- The official website also serves as a repository for models generated by the community, so that they can be downloaded or run as applets on the same page (where possible). A website has recently been launched, where you can share NetLogo models in a collaborative way, and its name is *NetLogo Modeling Commons* [30]

- On Yutzu[31] you will find a package containing a variety of information on the tool, as well as a set of videos designed in the form of tutorials by Carl Boettiger[32].

- As we said, in GitHub you will find the repository associated with the source code of the application[33] (only for advanced users with knowledge of Java and/or Scala).

- *James Steiner*, one of the programmers of NetLogo, has his own interesting library of NetLogo models called TurtleZero[34], as well as a wiki which goes into detail on certain characteristics of the language which are not addressed in the official manual.

- INSISOC (Social Systems Engineering Center) provides a small manual/course on the basics of NetLogo[35].

[27] *http://ccl.northwestern.edu/netlogo/docs/*

[28] *http://groups.yahoo.com/group/netlogo-users/*

[29] *http://groups.yahoo.com/group/netlogo-educators/*

[30] *http://modelingcommons.org*. From version 5.0.4 onwards, NetLogo has added an option in the Files menu to make it easier to upload models directly to this repository.

[31] *http://www.yutzu.com/*

[32] They were designed for version 4 of NetLogo, but are still largely valid for the current version.

[33] *https://github.com/NetLogo*

[34] *http://www.turtlezero.com*

[35] *http://www.insisoc.org*. It is one of the few resources to be found in Spanish.

Exercises

1. Open the turtle shapes editor tool and create one that you can use later in the models that you are going to make as you work through the book.

2. Visit the NetLogo repository in GitHub and look for information on the project and its community of developers.

3. Models Library:

   (a) **Diffusion Graphics** (figure 1.3):

      i. Locate the model **Diffusion Graphics** that comes under the classification of **Art** and load it in NetLogo.

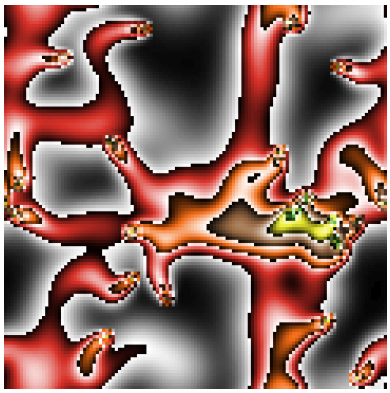


Figure 1.3: Diffusion Graphics.

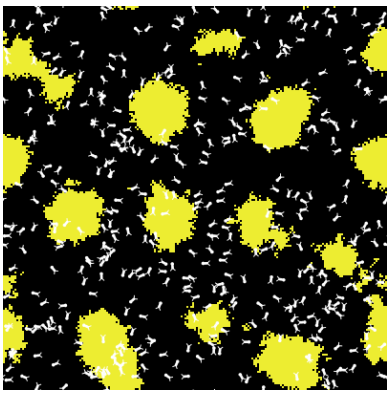


Figure 1.4: Termites.

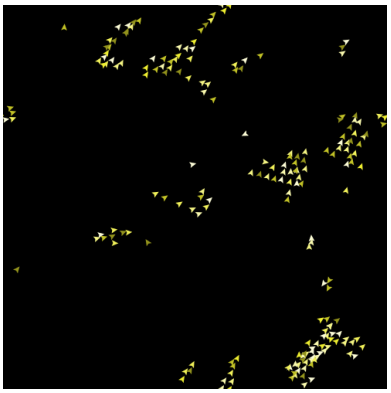


Figure 1.5: Flocking.

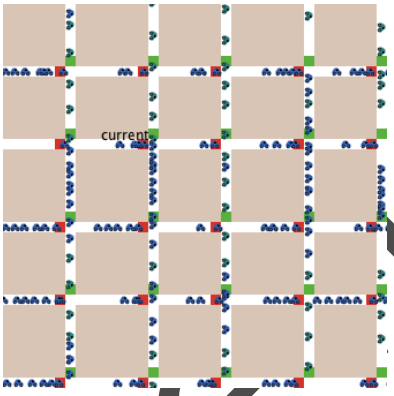


Figure 1.6: Traffic grid.

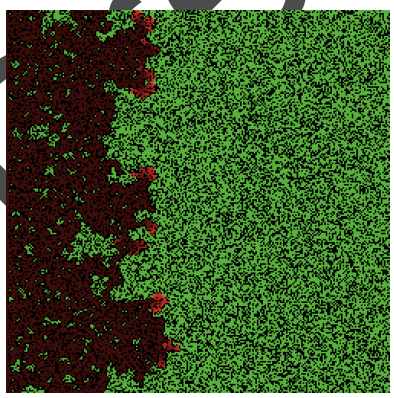


Figure 1.7: Fire.

ii. Run various experiments, changing the *number of initial turtles*, the *diffusion coefficient* and the *speed* of the turtles.

iii. Open the **Info tab** and read the explanations and comments about the model.

iv. Open the **Code tab** and have a look at the procedures and the instructions appearing in them.

(b) **Termites** (figure 1.4):

i. Locate the model **Termites** in the **Biology** section and load it in NetLogo.

ii. Read the information section to give you an idea of the purpose of the model and the role played by the parameters.

iii. When a termite gathers wood, in what color is it shown? Look in the **Code tab** to see where this color is assigned and change it to green.

(c) **Flocking** (figure 1.5):

i. Locate the model **Flocking** in the **Biology** section and load it in NetLogo.

ii. Read the information section to give you an idea of the purpose of the model.

iii. What is the role of each of the six parameters?

iv. Note how the values of the default parameters produce an organizational behavior in the turtles and how over a relatively short period of time they all face the same direction.

v. How does the time taken by the group to arrange themselves completely vary with respect to the parameters *vision* and *minimum separation*? And with respect to the *number of turtles*?

vi. Once there is a consensus with regard to the orientation of flight, what happens if you vary the value of the above parameters?

(d) **Traffic grid** (figure 1.6):

i. Locate the model **Traffic grid** which comes under the classification of **Social Science** and load it in NetLogo.

ii. Inspect the **Info tab** to see what the model does and the significance of its parameters, and of the role played by the various controls on its interface (**buttons**, **sliders** and **switches**).

iii. Note the differences in the simulation depending on whether or not the traffic lights are lit.

(e) **Fire** (figure 1.7):

i. Locate the model **Fire** in the **Earth Science** section and load it. This model simulates the propagation of a fire through a forest.

ii. Run the model for different values of the **slider** `density` and note the proportion of the forest that is burned in each case.

iii. Note how there is a *critical density* value (around 60%) so that for densities higher than this the fire spreads throughout almost the whole forest, whereas if the density stays below this, the proportion of burned forest is relatively small.

4. Join the NetLogo Users Group and look at the queries and solutions that have been posted recently.

2

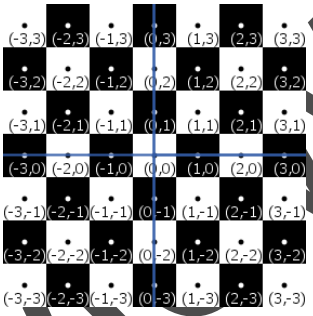
Familiarizing yourself with NetLogo.

The world of NetLogo

As in Logo, where the main element you worked with was the **turtle**, in NetLogo this interaction is extended so that you can operate with what are known generically as agents[1].

In NetLogo there are three types of agents:

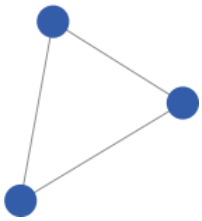
1. **Patches**, which make up the ground defining the world. They are arranged in the form of a grid that fills up the world and they are stationary.



2. **Turtles**, which are an extension of the Logo turtle and can move around the world and interact with the patches on the ground and with one another.

3. **Links**, which define the relationships between the mobile agents (turtles) like edges on a graph. They are not mobile, in the sense that you cannot move them around the world individually, as their location totally depends on the location of the agents they are linking.

[1] An **agent**, in this context, is an artificial, autonomous individual whose behavior and decision-making ability are governed by certain rules or characteristics. These agents interact with one another and with the environment, following a set of rules. They are flexible and have the ability to learn and adapt their behavior based on experience. This ability requires some kind of memory. Agents can also have rules to modify their rules of behavior, although all of this will of course depend on the programmer's ability to make them operate in this way.



For the moment we will be focusing on patches and turtles; we will look at the use of links in more detail in a later chapter.

The world of patches

Although there are language instructions enabling the characteristics of the world to be modified (its size and shape, in number of patches), for now we are going to access its properties using the visual tool provided by the environment.

To do this, you simply click on the "**Settings...**" button on the toolbar in the **Interface** tab (or right click in the world and select "**Edit...**"). This is where you can define the dimensions of the world, the location of the origin of coordinates, the visual size of the world (indicating how many pixels each patch will occupy on the screen), the topology of the world, the font size of the labels to be used by the agents and some other options that we will not be discussing for now (figure 2.1).

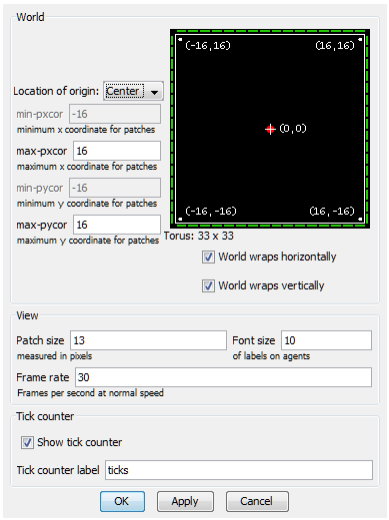
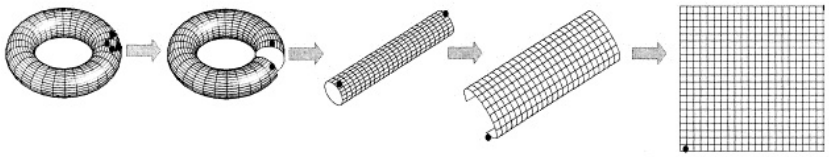


Figure 2.1: Edit world characteristics window.

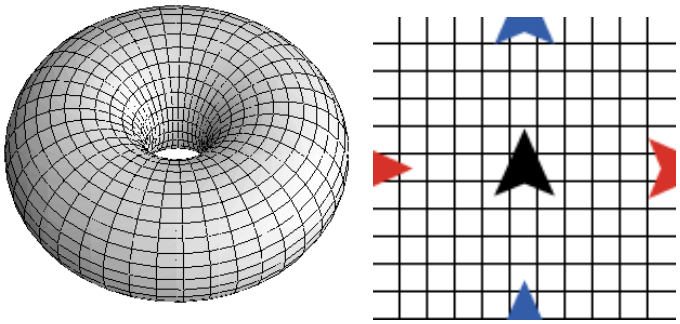
With regard to the **topology** of the world, the following must be taken into account: *the worlds that can be defined in NetLogo are always finite, i.e., they are of a finite size and are made up of a finite number of patches, but that does not mean to say that they have to be limited*. In other words, you can configure the topology of the world so that it has no borders. This can be done by identifying its lateral limits, so that if you identify the left and right limits, you will have a vertical cylinder, if you identify the top and bottom limits, you will have a horizontal cylinder, and if you identify the left with the right and the top with the bottom simultaneously, you will end up with a figure known mathematically as a **torus**.



Once you have established some of these settings, the mobile agents that move beyond one of these will appear on the opposite side, and the patches of any of the limits will have as neighbors the patches of

the other limit identified with it.

The following figures show a 3D representation of a torus and the connection between its limits in the 2D world. You can see how the red turtle crosses the right-hand side and appears on the left-hand side (identification on the horizontal axis), whereas the blue one crosses the top limit and appears at the bottom (identification on the vertical axis).



Every patch making up the world (like every agent we will be looking at) has a set of basic properties. To access these properties through the interface, you simply right click on the patch that you want to inspect, and select "**Inspect patch...**" (figure 2.2).

The pop-up window shows you a close-up view of the selected patch and its basic properties: coordinates in the world (**`pxcor`**, **`pycor`**), color (**`pcolor`**), the label it shows (**`plabel`**) and the color used for this label (**`plabel-color`**)[2]. By using this window you can modify some of the properties of the patches, but as they are static agents, you will not be able to modify their coordinates. In fact, given that the patch coordinates are unique (there is no other patch in the world with the same coordinates), these are used to identify it and to refer to it.

When the world is created, the patches are black (although in some of the images used in this book we will show them as white); if you want to change their color you just need to assign an appropriate value to the `pcolor` property of the patch.

Although NetLogo is capable of working with the RGB representation of colors[3], the default system it uses to work with them is slightly different and only requires one number to identify them. Although this system has certain limitations, it also has many advantages as it groups together, every 10 units, ranges of the same color. The following table shows the values of each of the associated colors (only the whole values are shown, but it accepts decimals for intermediate colors). In addition, for the 14 basic colors, NetLogo has constants with predefined values. So, to all intents and purposes, saying `orange` and `25` is the same thing[4]:

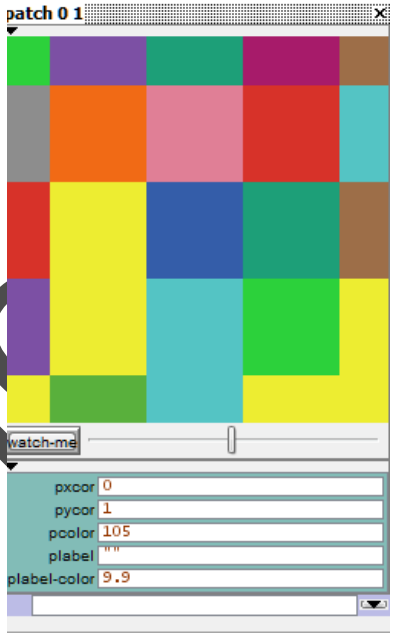
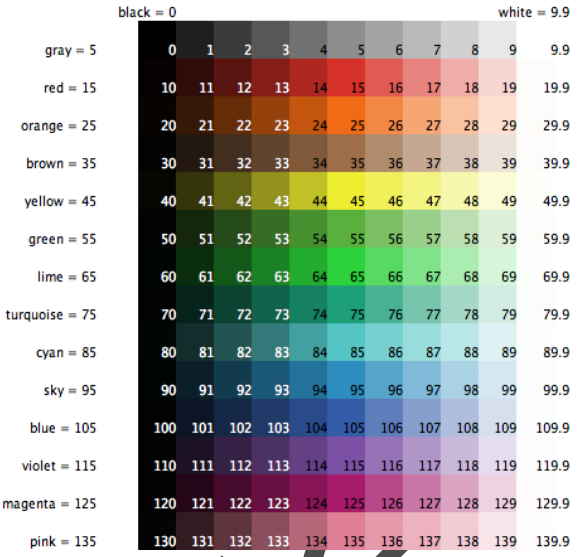


Figure 2.2: Inspect patch window.

[2] At a later stage we will look at how to define new properties, but for now the default ones provided by the system will suffice.

[3] **RGB** is a representation typically used when working with digital systems and indicates the proportion of each primary color using three numbers: **R**ed, **G**reen and **B**lue, normally with values ranging from 0 to 255. So, a shade of yellow is $(255, 255, 0)$, white is $(255, 255, 255)$ and black $(0, 0, 0)$.

[4] Consequently, if you are looking for a lighter shade of green, you just need to enter something like `(green + 2)`, a dark red could be `(red - 2)`, giving you a straightforward way of working with colors which you will find extremely beneficial.



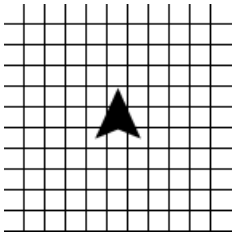
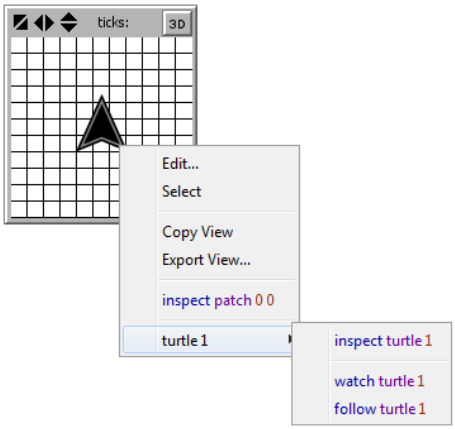


Figure 2.3: Newly created turtle in the centre of the world.

Introducing turtles

We are now going to take a closer look at turtles. Unlike patches, which are created automatically when the world is set up, turtles need to be created manually in order for them to appear in the world, and they can be created at any time. Before we see how to create and control them, let's assume that there is a turtle in the world and you are going to analyze its default characteristics (figure 2.3).

By right clicking on the turtle you can access the context menu showing the existing turtles below the cursor (you will also see the patch on which the turtle is located) so that you can select the relevant turtle.



As you want to access the turtle's properties, you click on "**inspect turtle x**" (we will see what the other options are for later), which will open a window similar to the one we saw to show the properties of the patch (figure 2.4).

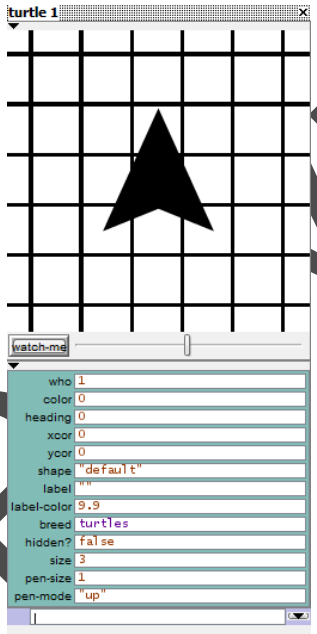


Figure 2.4: Turtle inspection window.

The default properties of a turtle (a few more than patches have) are:

- **`who`**: unlike patches, which are identified by their location, turtles have no property to differentiate them at first glance from one another. Consequently, NetLogo assigns a unique identifier to each turtle in the world by means of this number, which increases sequentially and without repetition. So, the identifier of the first turtle will be `0`, the next one created will have `1` as its identifier, and so on.
- **`color`**: the color of the turtle according to the color code followed by NetLogo. Similar to the color seen for patches.
- **`heading`**: the orientation of the turtle on the map. This is measured in degrees and it is important to note that it does not follow the same criteria as normal trigonometry, but starts counting from the positive Y axis and increases in a clockwise direction (figure 2.5).
- **`xcor`**, **`ycor`**: coordinates of its location in the space. This is measured in units of patches but, unlike these, it can take decimal values, representing all of the intermediate values between the coordinates of two consecutive patches (figure 2.6).
- **`shape`**: shape of the turtle. The default shape is the one shown in the images but the models come with different shapes that can be used and a shape editor to create them. Some of the predefined shapes are:

- **`label`**: label that the turtle will show on the screen (if it is not empty)[5].
- **`label-color`**: color of the label.
- **`breed`**: family of turtles to which it belongs. This characteristic of NetLogo is one of its strong points, as it enables you to define families of agents that can have different sets of properties and behaviors. We will be looking at this in a later chapter.

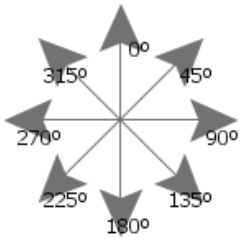


Figure 2.5: NetLogo orientation system.

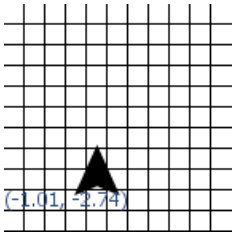


Figure 2.6: Coordinates of the turtle.

[5] Figure 2.6 shows the coordinates of a turtle by means of its label.

- **`hidden?`**: tells you whether the turtle is hidden or visible. Turtles are visible by default but can be hidden individually in the representation of the world.

- **`size`**: determines the size of the turtle (in patches). Its default value is 1.

- **`pen-mode`**: as in other versions of Logo, turtles can leave a trace on the ground as they move around. The possible values of this parameter control this option are: `up` (default value), the turtle's pen is up and will not leave a trace; `down`, the turtle's pen is down and will leave a trace; `erase`, the turtle lowers its eraser and will delete, along its path, any traces it may come across.

- **`pen-size`**: This parameter determines the width, in pixels, of the pen used to trace the path. Its default value is 1 (figure 2.7).

Figure 2.7: A figure drawn with a variety of pen sizes.

As with patches, you can change the values of a turtle's properties by using this window, although you would normally do this through programming language instructions. As a turtle is a mobile agent, you can change its spatial coordinates (which will make the turtle appear instantly in the new location), but the `who` property cannot be modified as it is the identifier.

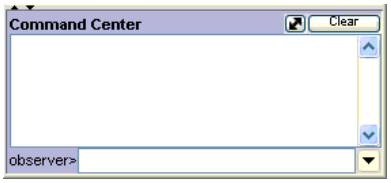
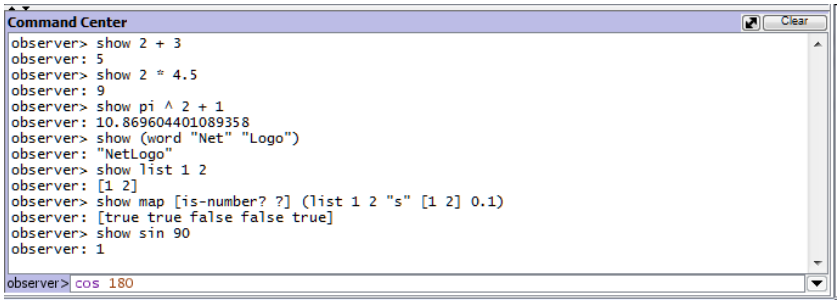


Figure 2.8: By default, the Command Center occupies the bottom horizontal part of the Interface tab, although you can move it to the right-hand vertical area. You can change its size by dragging the bar that separates it from the interface, and you can delete its content by clicking on the **Clear** button.

Command Center

In this section we are going to interact with NetLogo through the **Command Center**, which will enable you to take a closer look at the programming language in an interactive way, without having to enter full procedures or needing to know much about the language (figure 2.8).

We can use this center to communicate with NetLogo by entering instructions which will be executed when you press the Return key. Let's begin with a few simple examples to show the type of data that NetLogo is capable of handling, i.e., like an extended calculator:



The Command Center is made up of a bottom bar, where the commands are entered, and an area showing the results of any instructions

that produce a return. To all intents and purposes, it behaves like a text console similar to those of other systems.

As you can see on the bottom line, the system automatically colors the instructions according to whether it recognizes the text as a valid instruction, as a constant, etc. Furthermore, although you do not need to add the **`show`** command, so that the result is shown, the interpreter will add it automatically to your request where the instruction returns a result and it is not an action[6].

[6] We will see how the NetLogo instructions can be divided into two main groups: **actions**, which do not return anything but make changes in the world, and functions, (**reports**) which return values.

Working with numbers. Random numbers

We will make use of certain mathematical functions of NetLogo which will be useful later for practicing the syntax of the language in the interpreter.

NetLogo does not differentiate internally between whole and decimal numbers, but they are all stored internally as double-precision floating-point numbers[7]. Often, if a function is expecting a whole number as entry data and you give it a number with a decimal part, it will simply ignore the decimal part.

Important: **All operators in NetLogo must be separated by spaces. This means that `sin(x+1)` must be written:`sin (x + 1)`.**

The number range within which it can work is approximately -2^{53} to 2^{53}, i.e., `+/-9007199254740992`. If a calculation exceeds this amount it will not return an error, but the result may not be accurate. For example:

[7] Consequently, a whole number for NetLogo is simply a number without a decimal part, so that `1` and `1.0` are considered to be the same, which is not the case in many other languages. What does happen is that NetLogo hides the `.0` terminations to make it easier to identify the whole numbers.

```
observer> 2 ^ 52 + 1 = 2 ^ 52
observer: false
observer> 2 ^ 53 + 1 = 2 ^ 53
observer: true
```

There may also be problems with the accuracy of fractions[8]. For example:

[8] This is a common problem with all Java-based tools.

```
observer> 1/6 + 1/6 + 1/6 + 1/6 + 1/6 + 1/6
observer: 0.9999999999999999
```

And, as is normal in almost all programming languages, dividing by `0` will return an execution error.

With respect to random numbers, and as with other programming languages, NetLogo works with **pseudo-random** numbers[9]. The way in which to ensure that the same values are not always obtained is to make this generator function dependent on an initial value (*seed*), which means that if you always start with the same seed, you will always obtain the same number[10]. To solve this problem, the need to give a seed is normally hidden from the user by making use, for example, of the time on the computer (which is a quantity that is constantly changing). The following sequence of instructions may help you to understand this phenomenon:

[9] **Pseudo-random** numbers, whilst appearing to be random, are generated by a deterministic function that returns numbers within a range with uniform probability.

[10] Although this characteristic may seem counterproductive, in the case of scientific modeling, it converts to a quality that is beneficial to us, as it enables all experiments to be reproduced by storing the seeds with which they have been created.

From now on, and in order to give a more compact representation of the interactions, we will use the symbol => to show the output from the Command Center.

```
observer> random 100              => 9
observer> random 100              => 44
observer> random-seed 1
observer> random 100              => 81
observer> random-seed 1
observer> random 100              => 81
```

As can be seen from the above, the **`random`** function receives a number, `n`, as the entry data and returns an integer pseudo-random number in `[0,n)`[11] (or in `(n,0]` where `n < 0`) following a uniformly distributed probability. There is a non-integer (floating-point) version, called **`random-float`**, which operates on decimal numbers. Also, the **`random-seed`** function enables the generator seed of pseudo-random numbers to be fixed.

[11] i.e., one of: `0,1,2,...,n-1`.

Given that the last value is never returned, the instruction that returns a random integer number between `1` and `10` would be `(1 + random 10)`. The general formula for obtaining a random integer number between `a` and `b` would be:

```
a + random (b - a + 1)
```

For example, to generate an integer number between -5 and 5 the instruction would be: `(random 11) - 5`

And what happens if you want the returned number to be a decimal number? For example, if you want to randomly obtain a decimal number between `1` and `10`, the instruction would be: `1 + random-float 9`. And the general formula for returning a random decimal number between `a` and `b` would be[12]:

```
a + random-float (b - a)
```

[12] Can you explain why the expressions are different depending on whether or not you want decimals?

So, the generation of a random decimal number between -5 and 5 would be: `(random-float 10) - 5`.

In addition to the uniform distribution provided by the above functions, NetLogo has pseudo-random number generators that follow other usual distributions[13]. But these are not the only ones that make use of pseudo-random numbers; throughout this book we will be looking at functions that make a *random* selection from a set of objects, and all of these use internally the same random number generator from one seed, and their effect can therefore also be reproducible.

[13] Some of these functions are: **`random-exponential`**, **`random-poisson`**, **`random-gamma`**, and **`random-normal`**.

ACTIVITY: Use the Command Center like a calculator to perform certain numerical operations and familiarize yourself with the particular characteristics of NetLogo syntax.

Working with text strings

The second major type of data with which NetLogo can work are *text strings*. A **string** is simply a set of characters between quotation marks.

For example `“this is a string”`[14]. An empty string is represented by `“”`[15].

Below are some of the more usual functions for working with strings:

```
length "NetLogo"                => 7
first "NetLogo"                 => "N"
but-first "NetLogo"             => "etLogo"
last "NetLogo"                  => "o"
but-last "NetLogo"              => "NetLog"
item 4 "NetLogo"                => "o"
item 0 "NetLogo"                => "N"
substring "NetLogo" 3 5         => "Log"
member? "e" "NetLogo"           => true
member? "ne" "NetLogo"          => false
position "e" "NetLogo"          => 1
remove "L" "NetLogo"            => "Netogo"
remove "s" "NetLogo"            => "NetLogo"
replace-item 3 "NetLogo" "l"    => "Netlogo"
reverse "NetLogo"               => "ogoLteN"
word "Net" "Logo"               => "NetLogo"
```

The following points are worth noting:

1. The first position of the characters in strings is `0`.
2. When working with strings and characters, NetLogo differentiates between upper and lower case.
3. Use the instruction **`word`** to join strings together.

ACTIVITY: Take any phrase, express it as a text string and execute the necessary instructions so that you can separate it out word by word.

Working with lists. Compound data

When you are working with simple blocks of data it may be sufficient to use numbers and strings, but you will often need to store or handle many single values at the same time, in which case, the best solution is to work with *lists*.

A **list** is an ordered structure that stores many units of information. For example, `[0 2 4 6 8]`[16] is the list of even numbers lower than `10`. One interesting characteristic of lists is that they do not have to be homogeneous, but can contain different types of data, even other lists. For example:

```
[0 2 “Net” 6 [1 3 5] “Logo”]
```

We will see how lists can become highly efficient allies for working very easily with structured information [17].

[14] NetLogo does not differentiate between strings, words or characters. Only the length of the string and the existence or not of spaces makes us interpret it in one way or another. From version 5.0 NetLogo can work with *Unicode* characters.

[15] We have already seen empty strings in the patch and turtle labels to generate empty labels (or not to use them, which would amount to the same thing.

If you need to insert a special character (tab, new line, etc.) into a string, use an **escape sequence**:

1. `\n` = new line
2. `\t` = tab
3. `\"` = double quotation marks
4. `\\` = backslash

[16] In other programming languages, round brackets are used to delimit lists. NetLogo uses square brackets.

[17] We will also see how they are the natural format for handling information from agents or groups of agents.

Constant lists (those which do not vary) can be created by typing them directly between square brackets[18]:

[18] Note that the last of these is the list with no elements, i.e., the **empty list**.

```
observer> [0 2 4 6 8]
observer> [0 2 "Net" 6 [1 3 5] "Logo"]
observer> []
```

But if you want to create a list using the result of a function, or a value that is not constant, in some of its terms, you will need a constructor in order to create it. In NetLogo, this constructor is **`list`**[19]:

[19] Note that if you want to create a list with more than two data (or with one only) you must include the full instruction between round brackets. We will see that there are certain other NetLogo instructions that allow an indeterminate quantity of entries, and you will need to include in all of these the full instruction between round brackets if the number of entries is other than 2.

```
observer> list (random 10) (random-float 10)
observer: [2 2.04452253035499]
observer> (list (random 10) (random-float 10) 5)
observer: [8 0.27387597179620027 5]
observer> (list (random 10))
observer: [4]
```

A very useful way of constructing lists is by means of the **`n-values`** function, which enables you to generate a list of a determined length by means of a function. For example:

```
observer> n-values 10 [2 * ?]
observer: [0 2 4 6 8 10 12 14 16 18]
```

constructs a list of size 10 and uses the function $2x$ to fill it[20]. The general format is:

[20] Note the use of the special variable, `?`, as a variable of the function used.

```
n-values k [ f(?)  ]
```

which generates the list `[f(0) f(1) ... f(k-1)]`.

Some examples of its use are:

```
observer> n-values 5 [0]
observer: [0 0 0 0 0]
observer> n-values 5 [?]
observer: [0 1 2 3 4]
observer> n-values 3 [sin (? * 360 / 3)]
observer: [0 0.8660254037844387 -0.8660254037844385]
```

Most of the instructions we have seen for handling strings work with lists:

```
observer> length [0 1 2 3 4 5]        => 6
observer> first [0 1 2 3 4 5]         => 0
observer> but-first [0 1 2 3 4 5]     => [1 2 3 4 5]
observer> last [0 1 2 3 4 5]          => 5
observer> but-last [0 1 2 3 4 5]      => [0 1 2 3 4]
observer> item 4 [0 1 2 3 4 5]        => 4
observer> sublist [0 1 2 3 4] 2 4     => [2 3]
observer> member? 1 [0 1 2 3 4 5]     => true
observer> position 1 [0 1 2 3 4 5]    => 1
observer> remove 1 [0 1 2 3 4 5]      => [0 2 3 4 5]
observer> replace-item 1 [0 1 2] "a" => [0 "a" 2]
```

```
observer> reverse [0 1 2 3 4]          => [4 3 2 1 0]
```

The positions of lists also start with `0`.

But we also have a broader set of functions that can be applied, for example, to numerical lists:

```
observer> max [1 2 3 4 5 4 3 2 1]       => 5
observer> min [1 2 3 4 5 4 3 2 1]       => 1
observer> sum [1 2 3 4 5 4 3 2 1]       => 25
observer> variance [1 2 3 4 5 4 3 2 1] => 1.944444
observer> mean [1 2 3 4 5 4 3 2 1]      => 2.777777
observer> modes [1 2 3 4 5 4 3 2 1]     => [1 2 3 4]
observer> sum [1 2 3 4 5 4 3 2 1 "a"]   => 25
```

In the last example, the functions that are expecting numerical lists ignore the elements that are not numerical.

ACTIVITY: Try the following examples and work out why they each function differently:

```
observer> sum ["a" "b"]
observer> max ["a" "b"]
```

We can also extend the lists by adding elements to either of its two extremes[21]:

```
observer> lput "a" [1 2 3]          => [1 2 3 "a"]
observer> fput "a" [1 2 3]          => ["a" 1 2 3]
```

[21] These primitives will be very important, as together with `first`, `but-first`, `last` and `but-last` you can construct and deconstruct lists systematically.

The **`sentence`** instruction is used to join lists together.

ACTIVITY: Deduce from the following examples how it works and how it is different from `list`:

```
observer> list [1 2 3] [4 5]          => [[1 2 3] [4 5]]
observer> sentence [1 2 3] [4 5]      => [1 2 3 4 5]
observer> list 1 [4 5 6]              => [1 [4 5 6]]
observer> sentence 1 [4 5 6]          => [1 4 5 6]
observer> (sentence 1 [4 5 6] 7)      => [1 4 5 6 7]
observer> (sentence [1 2] [3 4] 5)=> [1 2 3 4 5]
observer> (sentence [1] [4 [5]] 7)=> [1 2 4 [5] 7]
```

In a later chapter we will look at how you can maximize the potential of the lists through programming, seeing how you can design recursive procedures to solve complex problems[22].

[22] Some of which can be written in a very concise way using certain additional instructions available in NetLogo.

Working with logic. Boolean logic

Like other programming languages, NetLogo is also capable of working with the Boolean values **`true`**/**`false`** which are obtained by conditional operations, such as comparisons[23]:

[23] Comparisons can also be made between strings, where the lexicographic order is automatically applied character by character.

```
observer> 3 > 2                => true
observer> "a" > "b"            => false
observer> "a" < "b"            => true
observer> n-values 4 [? < 3]=>[true true true false]
```

ACTIVITY: Write the instruction that will be capable of deciding which terms in the list `[0 1 2 3 4 5 6]` are even and which are not.

Recognizing the data type

In order to know automatically whether a piece of data is of one type or another, NetLogo provides certain Boolean functions for recognizing the type[24]:

[24] We will see that these kind of recognizers also exist to tell us what type of agent an object is (a turtle, a patch, a link, etc.)

```
observer> is-number? 3      => true
observer> is-number? "a"    => false
observer> is-string? "a"    => true
observer> is-string? 3      => false
observer> is-list? 3        => false
observer> is-list? [3]      => true
observer> is-boolean? 3     => false
observer> is-boolean? 0     => false
observer> is-boolean? true  => true
```

ACTIVITY: Establish what type the following data are: `[true false]`, `[[1]]`, `1 < 2`.

Execution contexts

ACTIVITY: If you have not yet asked yourself this question, now is the time to do so: what is the function of `observer>` when it appears beside the command entry bar? If you click on it you will find it easier to understand its significance (figure 2.9).

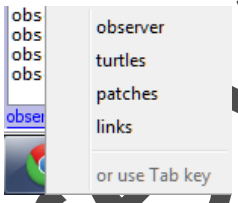


Figure 2.9: Command Center Contexts.

You will see that three of the four options appearing are the various types of NetLogo agents, whilst the fourth one is **`observer`**.

In NetLogo, execution is aimed at the agents; in other words, the different language instructions, as well as the procedures that we are defining, can be executed by any of the agents, or you can execute them yourself. The role of the *observer*, and there is only one, is the role that we play ourselves, outside of the artificial world that is executed but with the ability to control them all.

Obviously, not all types of agents can execute all language instructions as it can depend, for example, on whether the execution affects

properties that do not correspond to them. For example, if you execute `set color blue`, this instruction only makes sense in the context of the moving agents, as they are the ones with this property, but neither the patches nor the observer will be able to execute it[25].

Figure 2.10 will help to show you the difference. What has happened when "`1 + 1`" has been executed in the context of `patches`? If we had done this in the `observer` context, the system would have simply responded `observer: 2`, but having executed it in the context of patches, what has in fact happened is that each patch has executed the operation and has returned the result.

ACTIVITY: We are going to make use of this characteristic to carry out some tests on patches. To do this, execute the following instructions and justify the results obtained (although we have not yet seen some of the instructions used, it is easy to work out what they do from their names):

```
patches> set pcolor blue
patches> set pcolor one-of base-colors
patches> set plabel (word "(" pxcor "," pycor ")")
patches> set pcolor [pcolor] of one-of neighbors
patches> set plabel pxcor + world-width * pycor
```

We are now going to create a turtle to play around with in the world[26]:

- `observer> clear-all`: Let's start by deleting everything so that we can begin with a clean world[27].
- `observer> crt 1`: You then create a turtle (**`create-turtles`**) where you indicate the number of turtles to be created. The effect of this instruction is to create a turtle which, by default, will be located at the origin of coordinates but with random initial orientation (**`heading`**) and color.

Now that there is a turtle in the world, you can move to the `turtles` context to control the turtle, using the basic movement instructions (which are similar to those existing in other Logo languages). The figure below shows an interactive execution in which each line has been entered in order (changing between contexts accordingly as indicated).

The meanings of the unknown instructions are:

1. **`fd n`** : (**`forward`**) makes the turtle move forward `n` units (in patches).
2. **`rt n`** : (**`right`**) makes the turtle turn to the right (*its* right) `n` degrees.
3. **`pd`** : (**`pen-down`**) lowers the turtle's pen so that it leaves a trace as it moves.

[25] Although we will be seeing the ways in which an agent/observer can ask other agents to perform operations.

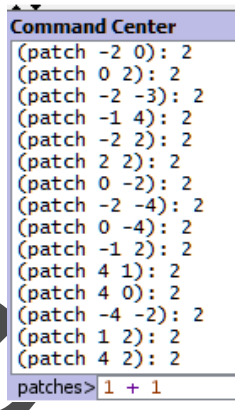
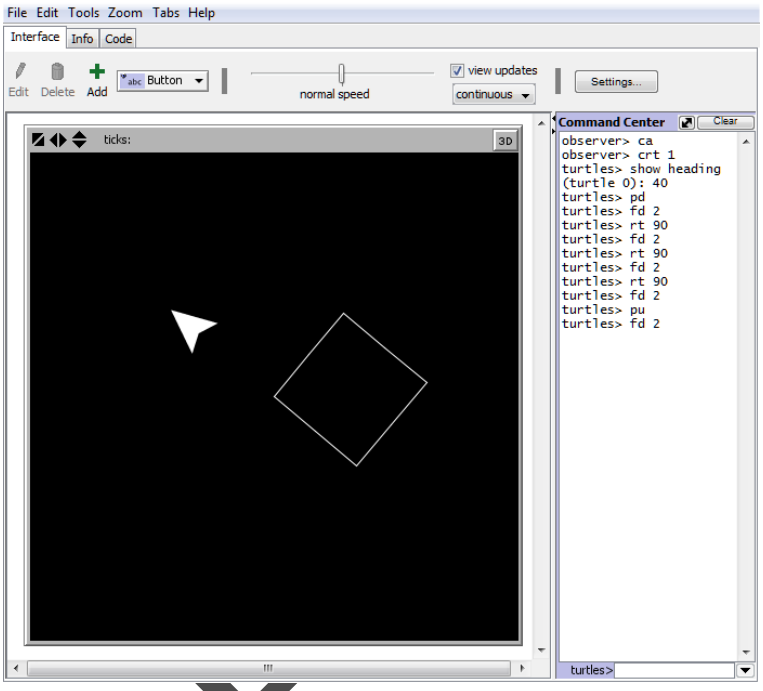


Figure 2.10: Execution of a command by patches. The window shows the last patches with their return; for example, the patch located at coordinates `(4, 2)` returns: `(patch 4 2): 2`, the same as the patch with coordinates `(1,2)`. Don't worry if when you execute it you obtain a different order in the list of patches; we will find out why later.

[26] Make sure that you change the interpreter context to `observer`, as the following instructions cannot be executed by the agents

[27] This is a good time to explain that NetLogo instructions (especially the most common ones) accept a shortened version, for example, **clear-all** and **ca** have the same effect.

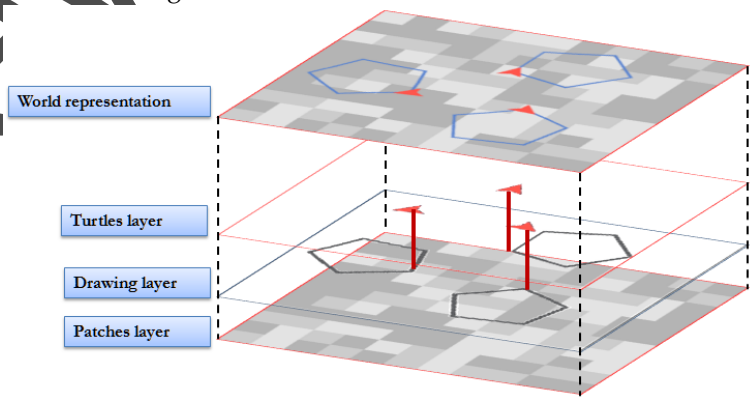
4. **pu** : (**pen-up**) raises the turtle's pen so that it does not leave a trace as it moves.



Note that on executing `heading` the turtle actually executes the instruction `show heading`, which means that the value of its heading property will be shown in the window.

It is important to note that the drawings that the turtles can create with their pen down are stored on a different layer, the drawing layer, with which the agents cannot interact[28].

[28] This layer can be deleted by using **clear-drawing**, or **cd** in its abbreviated version.



Unlike other Logos, in NetLogo there are no instructions for filling areas, and you set a background color by coloring the patches.

Here are some other instructions that you may find useful for controlling the turtle:

1. **bk n** : (**back**) makes the turtle move backwards `n` patches.

2. **ht** : (**hide-turtle**) hides the turtle (but if it moves with its pen down, it will still leave a trace).

3. **home** : takes the turtle to the origin of coordinates, maintaining its orientation and leaving a trace if its pen is down.

4. **lt n** : (**left**) turns n degrees to *its* left.

5. **pe** : (**pen-erase**) lowers the turtle's eraser, so that its movement will erase any traces that it comes across.

6. **st** : (**show-turtle**) shows the turtle (useful for when it has been hidden).

7. **stamp** : makes the turtle leave a mark on the world with its shape. This mark is merely a drawing on the ground, not an actual turtle.

8. **move-to patch x y** : makes the turtle move to the patch with coordinates (x,y)[29].

9. **die** : removes the turtle from the world.

10. **hatch n** : makes n exact copies of the turtle (all with exactly the same values in all properties).

[29] There are more general uses of the move-to command to place the turtle in the position occupied by any other agent, for example: move-to one-of patches.

Exercises

Although we have only just scratched the surface of the capabilities of NetLogo so far, we can already do some interesting things[30]:

1. **Different topologies**[31]:

   (a) Create a turtle and unselect in the world setting the options **World wraps horizontally** and **World wraps vertically**.

   (b) Set the max-pxcor and max-pycor values so that the size of the world is 21×21.

   (c) Orientate the turtle northwards and move it in that direction. What happens to the turtle if you move it further than the size of the world?

   (d) Return the turtle to the origin of coordinates, orientate it eastwards and see how the same thing happens as in the above case.

   (e) Do the same but this time using the options **World wraps horizontally** and/or **World wraps vertically**[32].

   (f) Taking into account all three topologies provided by NetLogo (*Euclidean plane*, *cylinder* and *torus*), try to predict what the shortest path would be between two points in any of the topologies.

[30] If you have never worked with turtles before, the best way to familiarize yourself with how they work is by interacting with them and seeing how they respond to the different orders in real time.

[31] *Topology* is the branch of mathematics dedicated to the study of the properties of geometric bodies that remain unaltered by continuous transformation. In other words, those which remain true even if you stretch and deform the world, as long as you do not break it or stick two separate parts together; this is why this is sometimes known as *rubber sheet geometry*.

[32] Do it with all possible combinations.

(g) For fixed dimensions of the world, if you imagine a torus, can you think of a rule that would enable you to occasionally return to the point of origin a turtle that moves in a straight line?

(h) Once you have drawn some figures on the cylinder or torus, can you imagine how they would look on the 3D representation of the surface?

2. **Drawing a polygon**[33]:

(a) Create a turtle from the Command Center.

(b) Use the turtle inspector to change the turtle's color, size and shape[34].

(c) Draw with the turtle a pentagon with side 4, in blue, and with a line width of 2 pixels. You may need to increase the size of the world.

(d) Calculate the rotation angle for drawing any regular polygon with n sides.

3. **Random paths**[35]:

(a) Create a turtle, activate the trace and choose a color and a line width.

(b) Use the `random` command to orientate the turtle with a random angle of between -90 and 90 degrees, and move it one step forward

(c) Repeat the action several times and observe the random path described.

(d) Do the same but this time restricting the rotation angle in a different way and advancing by a random length, let's say between 0 and 10 units.

4. **Patch context**:

(a) Select the `patch` context in the Command Center.

(b) Create different patterns of colors on the patches by using their coordinates and the instruction `set`. Try this with different formulas[36].

(c) Use `random` to assign random colors to the patches from the range of greens.

(d) Use the `x mod 2` operation, which returns the remainder of x divided by 2, to draw a checkerboard in two colors, assigning one color to each patch according to the value of `(pxcor + pycor) mod 2`. Find out what the effects are if you replace 2 with another natural number.

[33] The **intrinsic geometry** of the turtle, something that we will be looking at in more detail later, means that how it is controlled enables you to understand the geometrical figures independently of a rigid reference system, and so its use, deciding on the best reference system for each figure you want to create, can simplify the process of generating them.

[34] Remember that if you want to know what shapes are available in your model, you can open the tool that comes with NetLogo for working with shapes. Try importing a new shape from the library.

[35] Mathematically speaking, random paths are any random process in which the position of the object at a certain point in time only depends on its position at the previous point in time and on a random variable that determines its direction and length of progress.

[36] For example: `set pcolor pxcor`, `set pcolor pxcor + pycor`, `set pcolor pxcor - pycor`, `set pcolor pxcor * pycor`, etc.

5. **Creating lists**:

   (a) The list of the first 100 natural numbers.

   (b) The list of the first 100 even numbers.

   (c) The list of the first 100 odd numbers.

   (d) The list of the powers of 2 as far as Netlogo will allow.

   (e) Starting with a fixed word, generate a list of letters chosen at random from those forming this word.

   (f) Generate a list of lists with all the possible permutations of the list `[0 1 2]`.

6. **Random numbers**:

   (a) Generate a random ticket for the National Lottery [37].

   (b) Generate a list of 1,000 random natural numbers chosen from between 0 and 1,000. Calculate the mean, the variance and the standard deviation of the list, as well as the maximum and minimum value of the list.

   (c) Generate a list of 10,000 binary numbers at random (0's and 1's). Calculate the mean, the variance and the standard deviation of the list.

7. **Angles and regular polygons**:

   (a) Calculate the list of the angles accumulated as you move round a regular polygon according to the number of sides of the polygon[38].

   (b) Generate the same list as above, but this time measuring the angle in radians[39].

   (c) Generate the list of sines and cosines of the angles from the above list.

8. **Square numbers**:

   (a) Calculate the list of square numbers from the first 1,000 natural numbers.

   (b) Is 516 a perfect square? And 2025?[40]

   (c) Check that the mathematical formula

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

   is valid for $n = 10, 100$ and 1000.

[37] Remember that, in its simple version, the National Lottery consists of six numbers chosen from between 1 and 49.

[38] In the case of a square, the resulting list should be `[0 90 180 270]`.

[39] NetLogo has the constant `pi` to represent the number π.

[40] You can use the command `member?` to answer this question.

9. **Logarithms**[41]:

   (a) Generate the list of logarithms to base 10 of the first 100 natural numbers.

   (b) What is the first natural number whose logarithm to base 10 is greater than 0.5?

   (c) Generate the list of logarithms to base k, with $k = 2, \ldots, 100$ of the number 100.

   (d) Calculate the sum of the logarithms to base 2 of the first 10 not null natural numbers and check that it is equal to the logarithm to base 2 of the product of all of them.

[41] In mathematics, the logarithm of a number, to a fixed base, is the exponent by which the base must be raised in order to obtain this number. For example, the logarithm of 1000 to base 10 is 3, because $1000 = 10^3 = 10 \times 10 \times 10$.

Juan Carlos García Vázquez
Fernando Sancho Caparrini

NetLogo

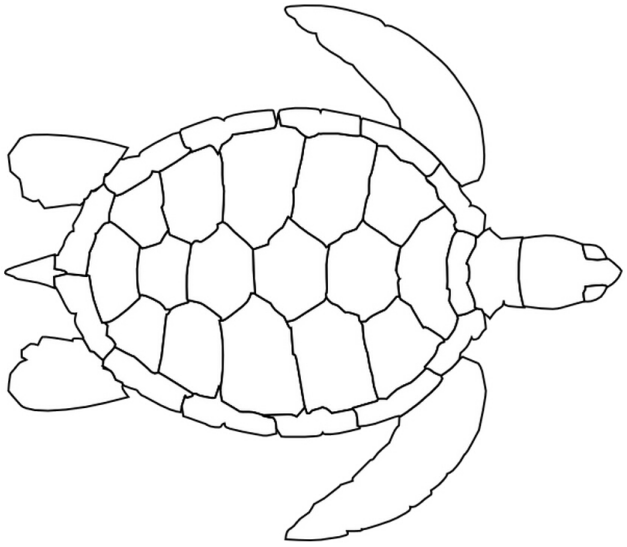
Una herramienta de modelado

A mí
El Observador

A los patches y los links, por ser nuestro soporte y romper nuestro aislamiento
Las Tortugas

A las tortugas, por alegrar nuestra parálisis con sus andanzas
Los Patches

A las tortugas, por ser nuestro principio y nuestro fin
Los Enlaces



A la tortuga que todos hemos tenido de niños

Prólogo

Cuando hace un tiempo nos planteamos usar NetLogo para completar el desarrollo académico de nuestros alumnos lo hicimos sin duda debido a los buenos resultados que a nosotros, en nuestra faceta de investigadores, nos estaba proporcionando.

Nuestra experiencia de uso de este software ha sido tan diversa como el propio NetLogo permite: lo estamos usando desde hace años en el proyecto ESTALMAT[1] para fomentar en los niños una forma de afrontar problemas fundamentada en las soluciones algorítmicas que un lenguaje de programación flexible puede ofrecer; en distintos cursos universitarios para introducir conceptos que cubren áreas tan aparentemente dispares como son los sistemas complejos, inteligencia artificial, modelado biológico, modelado cultural, experimentación matemática y física, etc. y tanto en niveles de pregrado como de máster; en el proceso de modelar comportamientos complejos en un ambiente de producción científica; para desarrollar herramientas de análisis y representación de información compleja en proyectos de humanidades digitales[2]; como prototipado de algoritmos para analizar y predecir comportamientos socioeconómicos, etc. Y en todas estas experiencias, los resultados han sido absolutamente satisfactorios.

El público que hemos tenido como alumnos en los diferentes cursos y como colaboradores en los trabajos de investigación cubre áreas de conocimiento supuestamente dispersas e incompatibles: desde humanistas que están interesados en el análisis de textos literarios, o matemáticos absortos en sus modelos abstractos, hasta ecólogos y biólogos que se interesan por la evolución de una determinada población de especies u organismos, o informáticos interesados en implementar un algoritmo distribuido para comprobar su validez. Y en todos ellos hemos podido constatar que, tras un momento inicial de excepticismo por el aspecto de "herramienta de juguete"que presenta, NetLogo les permitía entablar una conversación directa con su problema, mitigando en gran medida la dura realidad de que entre sus ideas y la proyección de éstas en el ordenador debe haber un aparato artificial que les obliga a formalizar su conocimiento de una forma completamente nueva y ajena a ellos.

[1] Este proyecto, que recibe el nombre del acrónimo de EStimulación del TALento MATemático, trata de detectar niños con capacidades para las matemáticas (que se corresponden normalmente con capacidades científicas en general) y estimularlos para que vayan aprendiendo los métodos, técnicas y estrategias matemáticas. Se puede encontrar más información en *http://thales.cica.es/estalmat/*

[2] Todas estas aplicaciones se han dado dentro del proyecto Hispanic Baroque (*http://www.hispanicbaroque.ca/*), financiado por el SSHRC de Canadá, y dirigido por Juan Luis Suárez, profesor de la Universidad de Western Ontario y director del laboratorio CulturePlex, un centro de estudios sobre complejidad cultural que hace uso de un enfoque que se encuadra dentro de las Humanidades Digitales (*http://cultureplex.ca/*).

Es quizás lo mejor que se puede decir de una herramienta de este tipo: NetLogo se convierte en una fina capa, casi transparente, entre el individuo y su materia de trabajo, y en poco tiempo puede pasar a formar parte de sus herramientas habituales de forma natural.

Nuestro punto de vista queda confirmado por la gran variedad de experiencias similares que encontramos en otros docentes e investigadores. Netlogo es una de las pocas plataformas de desarrollo con un rango tan amplio de usuarios, tanto por los niveles académicos que cubren como por los fines que persiguen.

A nivel educativo, gracias a su proximidad al lenguaje natural, se usó originalmente para enseñar conceptos básicos de programación a niños y jóvenes de edades muy variadas; aunque este uso se sigue dando en la actualidad podemos encontrar cada vez más cursos universitarios y módulos de formación de nivel superior en los que se hace uso de esta herramienta para introducir conceptos generales de modelado y resolución de problemas en diversas áreas del conocimiento.

En el contexto de la investigación académica, lo encontramos en el proceso de generar y verificar resultados de proyectos científicos, siendo cada vez más habitual encontrar referencias a su uso en artículos académicos especializados.

A un nivel más profesional, existen multitud de casos en los que NetLogo se ha usado como herramienta para validar experimentos sobre proyectos reales, muchos de ellos en el área del modelado ecológico y de dinámica de grupos.

Sin duda, esta diversidad de usos nos da una idea de su versatilidad y potencia, a la vez que de su sencillez y adecuada curva de aprendizaje.

A pesar de sus limitaciones frente a otros lenguajes más potentes y de más bajo nivel, el programador que use Netlogo como sistema de experimentación encontrará muchas ventajas durante el desarrollo y mantenimiento de las soluciones que prepare en la plataforma. Prechelt[3] mostró que el número de líneas de código que un programador puede escribir por unidad de tiempo es independiente del lenguaje utilizado. En consecuencia, puesto que los lenguajes de bajo nivel, más rápidos y potentes, requieren mayor número de líneas para realizar la misma tarea, la programación científica y orientada al descubrimiento debería hacer uso de lenguajes del más alto nivel posible que sean capaces de cubrir sus necesidades, y trasladarse a lenguajes más eficientes, pero de desarrollo más costoso (como C++ o Fortran), únicamente cuando exista la seguridad de que es imprescindible una mejora en el rendimiento, traduciendo el modelo de un lenguaje a otro. Esta elección inicial permite escribir más código, y comprobarlo, en la misma cantidad de tiempo, e incluso sabiendo que será necesario un lenguaje de bajo nivel para la obtención de resultados finales, el prototipado

[3] Lutz Prechelt. **Two Comparisons of Programming Languages**. Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*. O'Reilly, 2010.

rápido en un lenguaje de alto nivel como NetLogo puede ayudar a los modeladores a evaluar decisiones de diseño más rápidamente, reduciendo el tiempo total de desarrollo y mejorando el conocimiento del problema. Así pues, la elección de NetLogo como herramienta de trabajo y exploración para estudiantes e investigadores parece adecuada y sólidamente justificada.

Sin embargo, cuando decidimos escribir nuestras experiencias con NetLogo en forma de libro para ayudar a otros a introducirse en su uso, nos dimos cuenta de que no bastaba con proyectarlas como una sucesión de problemas y soluciones haciendo uso del lenguaje; apoyándonos en la experiencia que habíamos tenido en los años anteriores, decidimos que era necesario comenzar dando un paso atrás y endrentarnos a la creación de una guía de programación para usuarios que, en muchos casos, no tenían experiencia en otros lenguajes de programación[4].

[4] A pesar de que el tutorial que ofrecen los creadores de NetLogo en su web es muy completo y una ayuda imprescindible para todo usuario, nuestros alumnos y colaboradores demandaban un proceso más estructurado de aprendizaje.

De esta forma, hemos intentado escribir un ambicioso libro que pretende cubrir las necesidades del que se aproxima por primera vez a NetLogo, quizás sin ningún conocimiento previo de programación, quizás sin ningún conocimiento previo de modelado, y que a la vez sea capaz de profundizar en detalles menos evidentes para aquellos que ya han tenido contacto con esta herramienta. Esperamos haber cumplido con nuestro propósito y no defraudar a ninguno de ellos.

Cómo está distribuido el libro

El libro dedica los primeros capítulos, aproximadamente del capítulo 1 al 6, a una presentación introductoria de la herramienta, de los conceptos en los que se basa y de los fundamentos de programación necesarios para poder comenzar a desarrollar soluciones en ella.

Posteriormente, entre los capítulos 7 y 12, se introducen usos cada vez más elaborados de sus capacidades, cubriendo todo el espectro de funcionalidades que pueda requerir un proyecto de modelado habitual.

La tercera parte del libro, entre los capítulos 13 y 15, se dedica a detallar algunas características avanzadas de la herramienta, algunas de ellas que serán de utilidad únicamente en casos concretos, pero que es aconsejable conocer para expandir los límites de nuestra experiencia con la herramienta.

Por último, cerramos el libro con un capítulo en el que se proponen 8 proyectos completos que pueden ser desarrollados con NetLogo, similares a los que podemos encontrar en proyectos profesionales y académicos, y que pueden ser ampliados por distintas vías.

A lo largo de cada capítulo el lector encontrará insertadas una serie de actividades que tienen como objetivo ayudar a afianzar los concep-

tos y métodos explicados hasta el momento y facilitar la comprensión de los siguientes pasos del libro. Junto a estas actividades, en cada capítulo se propone un conjunto de ejercicios que pueden ser resueltos haciendo uso de los recursos explicados en él (hasta un total de más de 120 ejercicios).

Agradecimientos

Debemos iniciar los agradecimientos nombrando a Elena Varela, si hay algo en este libro que merezca la pena ser leído se debe a su meticuloso trabajo de revisar nuestra redacción. Todos los errores que se puedan encontrar se deben sin duda a nuestra incapacidad para expresar las ideas siguiendo sus consejos; su labor ha mejorado sin duda el libro que tienes entre las manos, pero también nos ha enseñado lo difícil y hermoso que puede llegar a ser escribir un texto en el que la claridad de las ideas se fundamente en el buen uso del lenguaje. Sin duda, este libro tendría una forma completamente distinta sin su inestimable ayuda.

Por supuesto, queremos agradecer a Juan Luis Suárez (y los proyectos que él dirige) por varias razones complementarias, porque es un amigo que siempre nos apoya y motiva con problemas que merece la pena abordar, y porque desde hace tiempo nos animaba a escribir algo parecido a lo que finalmente ha sido este libro. No olvidamos que quedamos en deuda con él (una deuda antigua que ahora crece) y le recordamos desde estas páginas que tenemos pendiente el proyecto, quizás continuación de éste, de mostrar las aplicaciones del modelado con NetLogo al análisis de la cultura.

Queremos agradecer la participación de Judy Kerry en la traducción inglesa del libro y las erratas (muchas) que ha ido encontrando en la versión española en el proceso de traducción. Su paciencia a la hora de enfrentarse al código fuente del libro (en LaTeX), a los textos en español y a nuestra forma de trabajar parece no tener límites.

No podemos olvidar a los distintos lectores que han ido revisando el libro en sus versiones preliminares, incluso cuando no era más que unas páginas mal redactadas y sin un objetivo bien definido. De entre ellos, aunque no olvidamos ninguno, queremos destacar los acertados comentarios y los apoyos de Antonio Pérez, de la Universidad de Sevilla, y de Marta Ginovart, de la Universidad Politécnica de Barcelona.

Índice de Contenidos

1

Un paseo por NetLogo. Antecedentes y entorno

Debemos iniciar este libro con una breve revisión de qué significa modelar, del papel que juega en el proceso de experimentación y representación de fenómenos, y de la diferencia que puede suponer el disponer de una herramienta adecuada para facilitar la transición del mundo real en el que encontramos el problema al mundo formal en el que podemos comprenderlo y, en el mejor de los escenarios, resolverlo.

Modelado y resolución de problemas

El conocimiento humano se basa principalmente en la observación y el análisis de fenómenos del mundo real en su sentido más amplio: desde las experiencias sensoriales del mundo natural, que abordan las áreas aplicadas, hasta las conceptualizaciones más abstractas, objeto de las áreas teóricas. Sorprende que aun cuando la distancia que separa algunos de los fenómenos es aparentemente insalvable, las metodologías empleadas para entender y predecir su comportamiento se asemejen. A raíz de tal constatación se formaliza durante el siglo XVII lo que conocemos como método científico; sus ventajas: asegura la robustez de las conclusiones obtenidas, puede ser aplicado a todas las áreas de conocimiento (sobre todo a las ciencias) y es independiente del tamaño del fenómeno observado. Aunque con variaciones según el momento histórico y los objetivos, el método científico se desarrolla en cuatro etapas:

- observación: etapa en la que se examinan atentamente los hechos

y fenómenos que tienen lugar en el mundo real y que pueden ser percibidos por los sentidos y medidos;

- formulación de hipótesis: en la que se elabora una explicación provisional de los fenómenos observados y de sus posibles causas;
- experimentación: en la que se reproduce y observa repetidas veces el fenómeno objeto de estudio, modificando las circunstancias que se consideren necesarias y confirmar así cuáles son las que provocan su aparición;
- tesis o generación de una teoría científica: en la que se interpretan los fenómenos observados de acuerdo con los datos experimentales.

Si la hipótesis se descubre falsa, deberá refutarse tras aportar los datos que la invalidan. Aumenta la probabilidad de éxito en buena medida cuanto mayor es el número de ejemplos o casos que permiten ilustrar el fenómeno estudiado. En muchas ocasiones, la experiencia se alimenta mediante la observación y manipulación del fenómeno real; en muchas otras, tal opción resulta físicamente imposible o desaconsejable, bien porque el fenómeno es inabarcable desde el punto de vista temporal o espacial, bien porque se basa en conceptos no manipulables, bien por razones éticas. Son precisamente los casos en que la manipulación de la realidad para reconstruir y probar hipótesis está vetada, los que han obligado al investigador a valerse de réplicas o modelos. Emplearemos en este libro la siguiente definición de modelo:

> *Un modelo constituye una representación abstracta de cierto aspecto de la realidad, formada esencialmente por los elementos que caracterizan dicho aspecto de la realidad y por las relaciones que mantienen esos elementos.*

De entre todas las opciones para crear modelos, destaca una por su eficacia: el modelado matemático. Se presenta bajo formas tan dispares como las teorías matemáticas (basadas en sistemas de axiomas y demostraciones, etc.), los modelos numéricos (basados en ecuaciones lineales, ecuaciones diferenciales, cálculo estocástico, etc.) o los modelos computacionales (basados en sistemas de agentes y sus interacciones, sistemas de partículas, algoritmos evolutivos, etc.) que se han desarrollado espectacularmente en los últimos años debido a la capacidad tecnológica emergente.

Bajo todas sus posibles formas, algunas de las ventajas que presenta la construcción de los modelos matemáticos son:

1. el proceso de construcción de modelos revela con frecuencia relaciones no evidentes a primera vista: el hecho de modelar suele ir acompañado de una mejor comprensión del fenómeno que se está representando;

2. una vez construido, es posible extraer propiedades y características de las relaciones entre los elementos del modelo que de otra forma permanecerían ocultas;

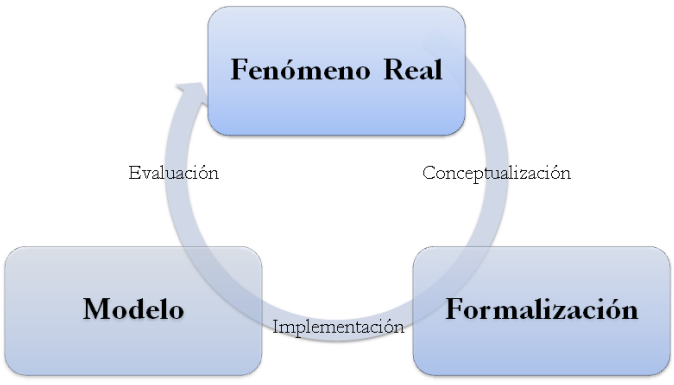
3. es posible simular situaciones complejas que no son representables en otros tipos de modelos;

4. a menudo proporcionan una resolución del problema, aunque no sea una solución analítica sino numérica, realizada por ordenador.

Los modelos se clasifican de muchas maneras en razón de la característica que se quiera destacar:

- según la forma en que se representan los elementos del modelo, podemos hablar de modelos icónicos, analógicos, simbólicos, etc.
- según su función, de modelos predictivos, evaluativos, de optimización[1], etc.;
- según el tipo de realidad que quieren modelar, podemos hablar de modelos deterministas frente a modelos estocásticos[2];
- según el modo en que se representa el tiempo, distinguimos entre modelos estáticos y modelos dinámicos[3].

En general, un buen modelo es aquel que se ajusta al fenómeno real de forma que nos permite comprender mejor sus propiedades y ampliar el conocimiento que tenemos de él. A lo largo de la experiencia acumulada en diversas disciplinas que hacen uso del modelado como herramienta esencial, se han destacado las siguientes fases para construir un buen modelo:



1. A partir del fenómeno real, tras una fase de interpretación de las observaciones pasamos a otra de formalización (haciendo uso de algún lenguaje formal).

[1] Los modelos predictivos informan del comportamiento de sus parámetros en un futuro. Los evaluativos intentan medir las diferentes alternativas, para poder comparar sus resultados. Y los modelos de optimización tratan de identificar un óptimo del problema, es decir, la mejor de las alternativas posibles.

[2] En los modelos deterministas la evolución futura del modelo está completamente determinada por el estado actual del mismo, mientras que en los estocásticos esta evolución puede depender de comportamientos aleatorios medibles por medios estadísticos.

[3] No hay evolución temporal en los modelos estáticos y sí en los dinámicos.

2. Si el modelo que queremos conseguir tiene propiedades dinámicas o admite un proceso de simulación que ayude a comprender la realidad, procedemos a implementar dicha formalización en un sistema que permita considerar este dinamismo.

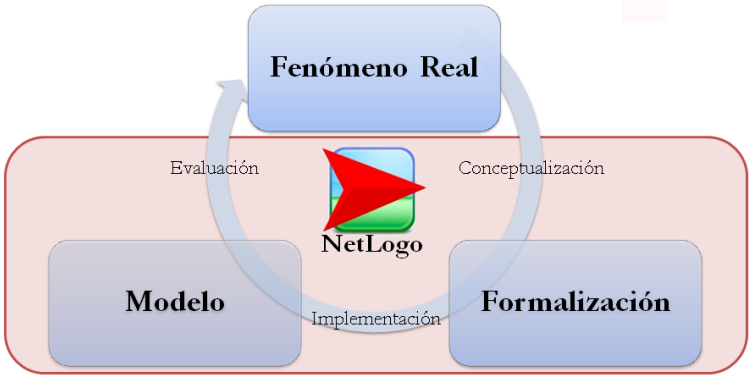
3. Por último, y tras haber ejecutado simulaciones en el modelo, evaluamos los resultados obtenidos confrontándolos con los que se pueden observar en el mundo real.

Debe tenerse en cuenta que lo normal no es realizar estos pasos siguiendo un proceso lineal y directo, sino que se forma un ciclo de vida en el que, por aproximaciones sucesivas, obtenemos una colección de modelos cada vez más fieles (al menos, en las características que son el foco de nuestra hipótesis) al fenómeno real.

Por su importancia como herramienta de investigación y teniendo en cuenta las fases necesarias para conseguir un modelo correcto, no cabe duda de que modelar es una actividad que se sitúa en el centro del proceso de generación y obtención de conocimiento, tanto a nivel educativo como a nivel de investigación[4]: ser capaz de modelar un problema es una muestra de haberlo comprendido correctamente, a la vez que la demostración de que se poseen las herramientas técnicas necesarias para su representación en un sistema formal.

El objetivo de este libro es proporcionar las bases necesarias para disfrutar de la autonomía que permite generar modelos que ayuden a plantear y verificar hipótesis plausibles. Veremos cómo Netlogo nos brinda la gran ventaja de ofrecer en un solo bloque una herramienta capaz de abordar prácticamente todo el proceso de modelado que hemos expuesto.

[4] Algo que no debe sorprender, ya que el proceso de investigación no es más que una continuación de un proceso educativo, basado en el descubrimiento, y en el que muchas veces la figura de maestro y discípulo recaen en la misma persona.



Al tratarse de un lenguaje de programación, el proceso de implementación es natural en NetLogo; aunque, gracias al tipo de elementos con los que trabaja, influirá también en el proceso de conceptualización y formalización del fenómeno real. Las características de la plataforma

hacen que la primera parte del proceso se simplifique considerablemente; pero va más allá y nos permite evaluar los resultados de su ejecución para poder compararlos con el mundo real por medio de la generación automática de experimentos y su ejecución.

¿Qué es NetLogo?

La mejor definición de lo que puede ser esta herramienta la aportan sus propios creadores[5] que, como dicen en el manual online de NetLogo:

- es un entorno de programación para la simulación de fenómenos naturales y sociales.
- es suficientemente simple para estudiantes y profesores, y suficientemente avanzado para que sirva de herramienta potente a investigadores de muchas áreas.
- permite a los usuarios cargar simulaciones y *jugar* con ellas, explorando sus comportamientos bajo condiciones cambiantes. Y también proporciona un entorno de desarrollo que permite a estudiantes, profesores y creadores de currículos crear sus propios modelos.
- es especialmente adecuado para el modelado de sistemas complejos que evolucionan en el tiempo. El programador puede dar instrucciones a cientos o miles de *agentes*[6] que actúan independientemente, lo que hace posible explorar las conexiones entre el comportamiento de los individuos a nivel local (nivel **micro**) y los patrones que emergen de sus interacciones a nivel global (nivel **macro**).
- representa a la siguiente generación de lenguajes de modelado *multiagente*[7], que incluyen, entre otros, a **StarLogo** y **StarLogo TNG**.
- tiene una extensa documentación y numerosos tutoriales. También viene con la **Biblioteca de Modelos**, una gran colección de simulaciones predefinidas que pueden ser usadas y modificadas. Estas simulaciones abordan contenido de áreas de las ciencias naturales y sociales, incluyendo biología, medicina, física, química, matemáticas, computación, economía y psicología social. Muchos currículos basados en modelos desarrollados con esta herramienta están disponibles en la biblioteca, y hay más en desarrollo.
- también puede ayudar a crear simulaciones participativas en clase con una herramienta llamada **HubNet**[8] que trae incorporada, de forma que cada estudiante puede controlar uno de los agentes en una simulación.

[5] Fue creado en 1999 por Uri Wilensky y ha estado en desarrollo ininterrumpido desde entonces en el *Center for Connected Learning and Computer-Based Modeling*, de la Universidad de Northwestern, Evanston, Illinois.

[6] Por ahora es suficiente con pensar en un *agente* como un individuo que es capaz de trabajar de forma autónoma y que interactúa con su entorno y otros agentes.

[7] *Multiagente* significa que no trabaja con un solo individuo, sino con un grupo de ellos.

[8] A través de una red de ordenadores o haciendo uso de dispositivos móviles. Actualmente únicamente soporta las calculadoras gráficas de Texas Instruments como ejemplo de dispositivo móvil, pero está en marcha su adaptación a tabletas y smartphones.

- se ejecuta en una máquina virtual Java (JVM), por tanto, funciona en todas las plataformas principales (Mac, Windows, Linux, y otras). Se puede ejecutar como una aplicación independiente, pero los modelos diseñados en él también pueden ejecutarse como applets de Java en un navegador web. También permite trabajar por medio de la línea de comandos y por manipulación directa desde otros lenguajes de programación y/o herramientas, ya que proporciona una interfaz de comunicación para su manejo a bajo nivel.

Además, se debe añadir que en la actualidad NetLogo es un proyecto de software libre que está siendo mantenido por una comunidad creciente, liderada por *Seth Tisue*, con un proceso de evolución continuo y en el que hay un gran trabajo colaborativo a todos los niveles (desde la creación de modelos para el resto de la comunidad hasta la creación de extensiones o actualizaciones del núcleo de la aplicación). En el momento de la creación de este libro su código y planes de trabajo se pueden encontrar en un repositorio de Github[9].

[9] https://github.com/NetLogo

Del Logo al NetLogo

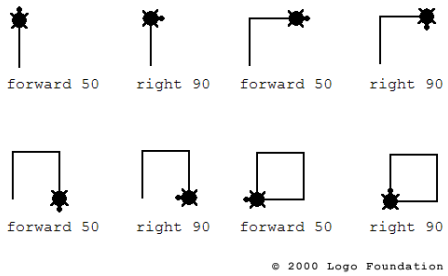
Figura 1.1: Implementación hardware de una tortuga para dibujar.

Logo es un lenguaje de programación de alto nivel que nació en los años 60, basado en **Lisp**, y de muy fácil aprendizaje, por lo que ha sido usado durante mucho tiempo como primera aproximación para mostrar los principios de programación a niños y jóvenes (figura 1.1).

Sin embargo, a pesar de estas intenciones didácticas, no debemos creer que es un lenguaje pobre, ya que dispone de todas las características necesarias para poder practicar e implementar todos los conceptos usuales de programación funcional avanzada[10].

[10] Como son: recursión, estructuras avanzadas de datos, funciones de orden superior (es decir, funciones que operan sobre funciones), etc.

Sin duda, una de las características más reconocibles de Logo es su capacidad para manejar unos pequeños *objetos gráficos* que permiten la realización de figuras geométricas por medio de instrucciones básicas, y que, como podemos ver en la figura 1.1, incluso a veces han tomado forma física.



Aunque realmente hablamos de Logo como *un lenguaje*, se corresponde con una familia de lenguajes similares entre sí y no hay una definición estándar del mismo. NetLogo comparte suficientes características con esta familia de lenguajes como para ser considerado uno

más de ellos, aunque con algunas diferencias importantes que detallamos en los siguientes puntos[11]:

- La precedencia de los operadores matemáticos es diferente. Los operadores infijos (aquellos que se escriben entre los operandos, como `+`, `*`, etc.) tienen menor precedencia que las funciones con nombre. Por ejemplo, `sin x + 1 = (sin x) + 1`.
- Las funciones `and` y `or` son funciones especiales que cortocircuitan el cálculo, lo que quiere decir que evalúan el segundo argumento únicamente si es necesario[12]. Esta característica hace que la evaluación de cadenas de condicionales sea más eficiente.
- Solo se pueden definir procedimientos en la pestaña de **Código**, y no de forma interactiva en el **Centro de Comandos**. En los siguientes capítulos profundizaremos en estos dos elementos.
- Las funciones que devuelven valores deben ser definidas con la palabra clave **`to-report`**, y en estos casos, el comando para devolver un valor ha de ser **`report`**.
- Al definir un procedimiento que admite parámetros de entrada, éstos deben estar encerrados entre corchetes.
- Los nombres de las variables siempre deben ser elegidos comenzando sin signos de puntuación, por lo que no son válidos los usos `:var` o `"var` tan comunes en otros Logos.
- Las variables locales y las entradas de los procedimientos se verifican léxicamente, no dinámicamente (es decir, si hay un error, se marca antes de la ejecución).
- NetLogo no tiene el tipo de dato `word` (también llamados `símbolos`), pero tiene el tipo `string` (cadena)[13].
- Las **`Tasks`** (o funciones *lambda*[14]) proporcionan una característica novedosa que ya es habitual en los lenguajes Lisp modernos, pero no en los Logos clásicos.
- Las estructuras de control, como **`if`** o **`while`**, son formas especiales, y no funciones ordinarias. Las formas especiales no se pueden definir, por lo que no se pueden crear nuevas estructuras de control[15].
- El comando **`run`** de NetLogo actúa sobre **`tasks`** y cadenas, pero no sobre listas, y no permite la definición ni redefinición de procedimientos.

Y, desde luego, NetLogo nos ofrece una capacidad fundamental que no se encuentra en la mayoría de los otros lenguajes de la familia Logo: trabaja directamente con agentes y es capaz de diferenciar *familias*

[11] Si no has sido usuario de Logo en alguna de sus versiones puedes saltar directamente a la sección siguiente, y esperar a más adelante, cuando sepas algo de NetLogo, para volver aquí.

[12] Con el operador `or`, en cuanto se encuentra un argumento cierto, no evalúa los demás porque el resultado global es cierto. Con el operador `and`, en cuanto se encuentra con un argumento falso, no evalúa los demás porque el resultado global es falso.

[13] Por ejemplo, lo que en Logo sería `[una frase]` (una lista de palabras), debería ser en NetLogo `"una frase"` (una cadena) o bien `["una" "frase"]` (una lista de cadenas).

[14] Una función *lambda* es una función anónima, a la que no se le pone nombre, y que puede ser tratada como cualquier otro dato del programa.

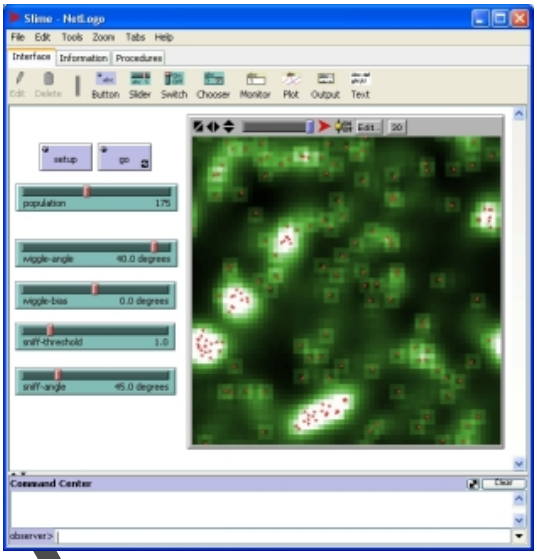
[15] Parte de esta funcionalidad se puede conseguir por medio de las **tasks**, como veremos más adelante.

de agentes para diseñar comportamientos individualizados para cada familia.

Instalación y ejecución de NetLogo

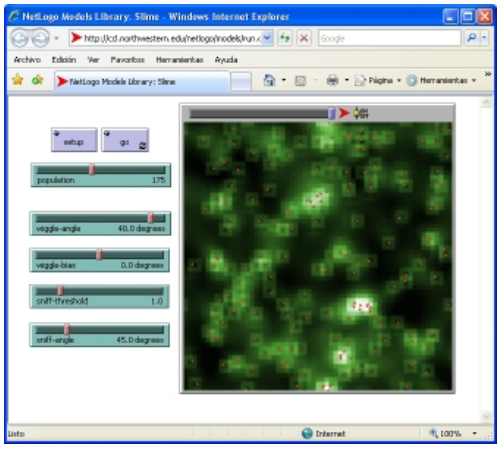
Gracias a que NetLogo se ejecuta en la máquina virtual de Java, y que admite la posibilidad de grabar los modelos como **applets** de Java (obsoleto), hay dos formas de ejecutarlos:

1. Tener el programa instalado y abrir el modelo desde la aplicación.



2. Por medio de un navegador web accediendo (en local o en un servidor en internet) al fichero asociado al applet. Esta modalidad presenta algunas limitaciones si se quiere hacer uso de funcionalidades especiales[16], pero admite la casi totalidad de ejemplos habituales.

[16] Además de las propias restricciones impuestas por medidas de seguridad en la ejecución de applets, algunas extensiones pueden no funcionar en modo applet.



A pesar de que ambas son válidas para la ejecución interactiva de los modelos y su uso del interfaz, hay notables diferencias entre ellas.

La más llamativa es que, mientras que en la ejecución del applet solo se puede acceder a la vista del interfaz del modelo (y, por tanto, no se pueden introducir modificaciones), en la ejecución desde el programa completo no sólo se accede a esta vista interactiva, sino que se puede acceder a la pestaña de código (que permite modificar el modelo) y a las herramientas adicionales que se proporcionan en la aplicación y que veremos brevemente en este mismo capítulo.

En realidad, existe una tercera forma de ejecutar los modelos de NetLogo, o incluso de operar con él, que es lo que se conoce como ejecución **headless**, que no carga el interfaz gráfico de NetLogo pero sí permite una ejecución completa de toda su funcionalidad como lenguaje de programación y motor de ejecución. Incluso, teniendo en cuenta que se ejecuta completamente en Java, se puede utilizar como librería para el desarrollo de aplicaciones que, cuando sea necesario, hagan uso del motor de NetLogo por medio de su API[17] de programación.

Para la mayor parte del contenido de este libro haremos uso de la aplicación completa, por lo que es aconsejable instalarla siguiendo los pasos que se especifican en la página oficial de NetLogo.

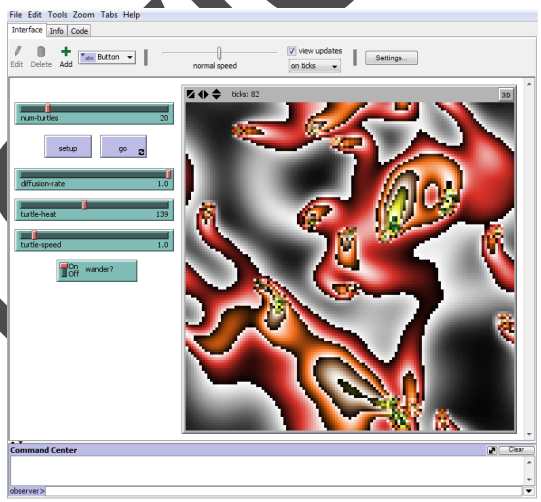
[17] API (del inglés **Application Programming Interface**) o **Interfaz de Programación de Aplicaciones**, es el conjunto de funciones y procedimientos que ofrecen algunas librerías para poder ser utilizadas por otro software.

Una visión rápida del entorno de trabajo

Tras el proceso de instalación, podremos encontrar varias aplicaciones en su carpeta junto a **NetLogo x.x.x**[18], que son **NetLogo 3D**, una versión que permite trabajar con modelos en mundos 3D, y **Hubnet**, que explicaremos brevemente al final de este capítulo y más extensamente en el capítulo 14.

[18] Los números dependen de la versión. La 5.2.0 es la versión estable en el momento de crear este documento.

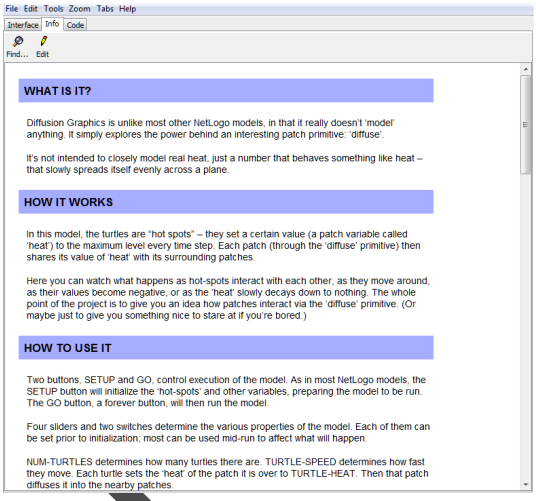
Al abrir la aplicación lo primero que debe llamar nuestra atención es la existencia de 3 pestañas que contienen las secciones principales sobre las que definiremos nuestros modelos:



1. **Pestaña de interfaz** (**Interface**): en la que se representa el mundo en el que se ejecutan las simulaciones, y donde podemos definir el interfaz que permite al usuario interactuar con el modelo.

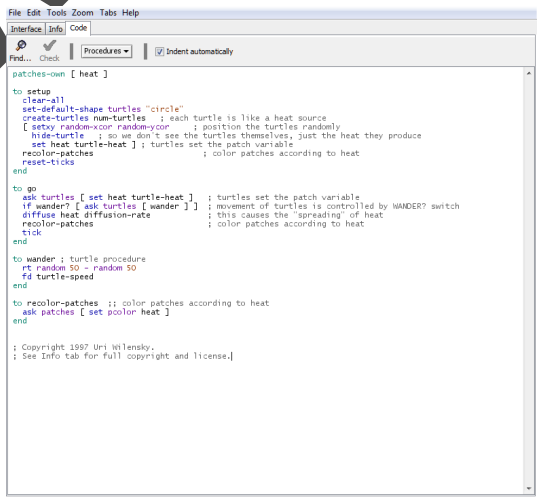
2. **Pestaña de Información** (**Info**): en la que podemos añadir información relacionada con nuestro modelo para aclarar conceptos de programación, explicar su funcionamiento, el modelo real que simula, etc.[19]

[19] El editor que proporciona es muy sencillo pero, usando un formato parecido al de las wikis, podemos conseguir resultados similares a los de cualquier página HTML.



3. **Pestaña de código** (**Code**): donde se escriben los procedimientos y declaraciones que definen el comportamiento del modelo[20].

[20] Proporciona las herramientas mínimas imprescindibles para una correcta edición del código.



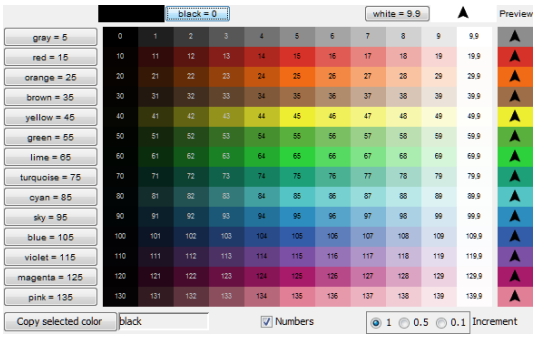
Como es habitual en todos los programas que trabajan con ventanas, en el menú superior encontraremos, además de las habituales opciones de **Ficheros**, **Edición** y **Ayuda**, el menú **Tools** (**Herramientas**) con el

que accederemos a las herramientas adicionales que complementan la experiencia con NetLogo.

Algunas de las más importantes son:

- **Color Swatches**: Permite navegar por el sistema de colores que usa NetLogo[21]. Además, cuando se accede a él desde algunas secciones del programa, puede usarse para seleccionar colores de forma interactiva.

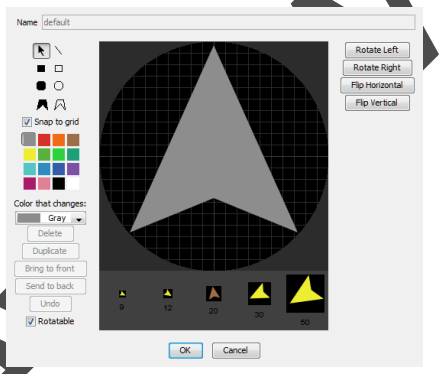
[21] Más adelante veremos las peculiaridades de este sistema.



ACTIVIDAD: Abre esta herramienta y acostúmbrate al sistema de colores de NetLogo. Cambia la resolución del muestreo por medio del control `Increment` y observa el contraste del color elegido sobre fondos negro y blanco.

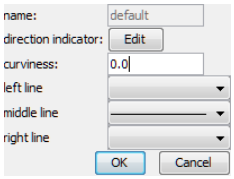
- **Turtle Shapes Editor**: Permite editar el aspecto[22] que muestran las tortugas en pantalla de forma que su representación facilite el reconocimiento del tipo de tortuga que es (y con él, su posible funcionamiento y comportamiento). Estas formas se almacenan en cada modelo, y además el sistema permite importar formas de otros modelos ya creados y de una librería que se precarga con el programa.

[22] Este aspecto se define por medio de formas vectoriales, de manera que puedan ser escalables sin perder resolución.



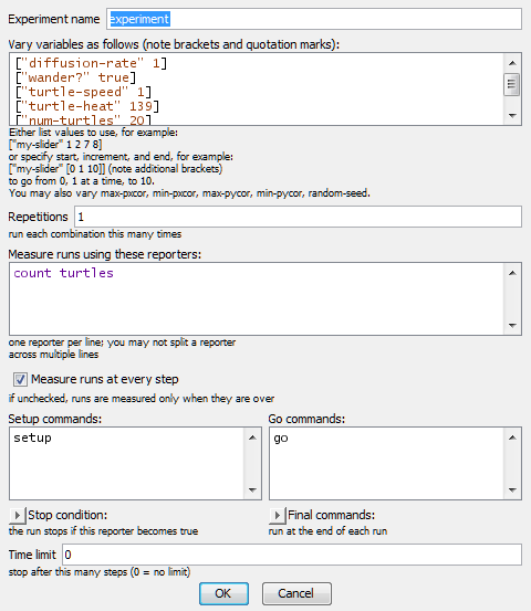
ACTIVIDAD: Carga algunas formas en el editor para ver cómo están hechas y crea las tuyas propias. Fíjate en las distintas formas simples que se pueden utilizar y el uso de la paleta básica de colores.

- **Link Shapes Editor**: Permite editar el aspecto de los enlaces que pueden conectar las tortugas entre sí, con parámetros para controlar su curvatura y las características visuales de las líneas e indicadores.



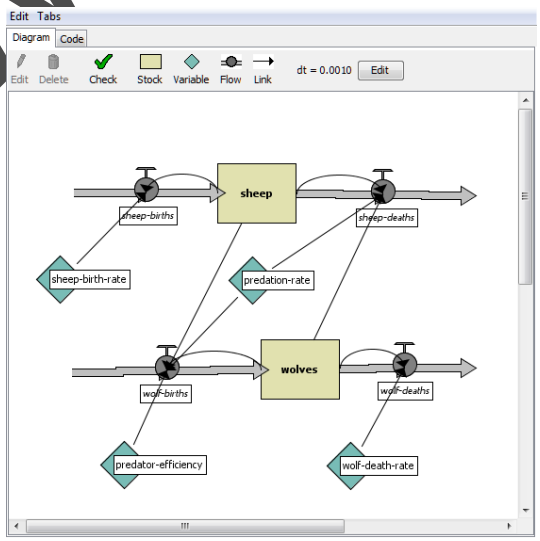
[23] Además, si disponemos de un ordenador con multiprocesador, permite hacer paralelismo real en su ejecución, lanzando experimentos independientes en cada uno de ellos.

- **BehaviorSpace**: Permite definir y lanzar conjuntos de experimentos sobre el modelo y grabar automáticamente todos sus resultados para ser analizados posteriormente con herramientas externas[23].

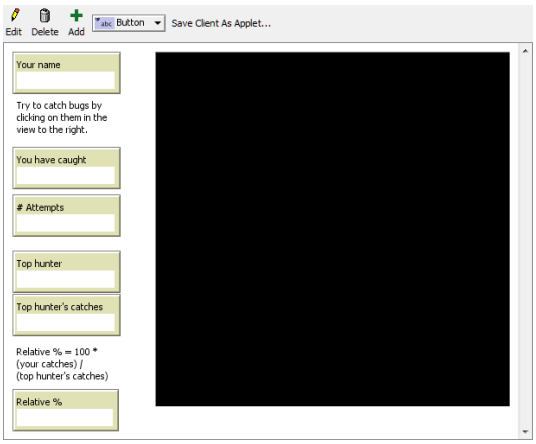


- **System Dynamics Modeler**: Abre una funcionalidad de NetLogo que permite diseñar un modelo de NetLogo por medio de técnicas habituales y estandarizadas de sistemas dinámicos[24].

[24] En la actualidad tiene muy poco uso, aunque es una funcionalidad que para ciertos tipos de problemas puede resultar muy cómoda.



- **HubNet Client Editor**: Abre el editor de interfaces de clientes para los experimentos colaborativos que se pueden ejecutar por medio de **HubNet**.



La Biblioteca de modelos

Como hemos comentado anteriormente, uno de las características más interesantes que tiene NetLogo es la cantidad de ejemplos completos que vienen por defecto en su instalación (todos ellos listos para ser ejecutados y modificados), y que recorren prácticamente cualquier área de conocimiento en el que el modelado de experimentos pueda tener sentido.

Por medio del menú de **Ficheros** (figura 1.2) podemos acceder al navegador de modelos (**Models Library**), que nos muestra un árbol de carpetas con los modelos agrupados según un criterio muy claro:

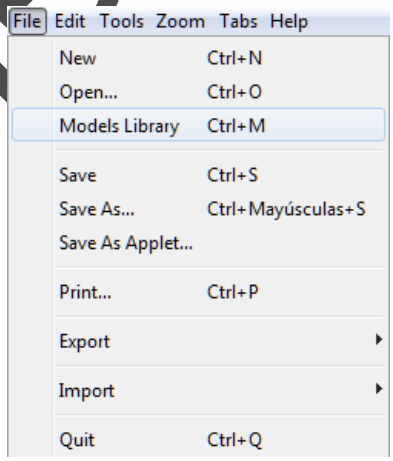
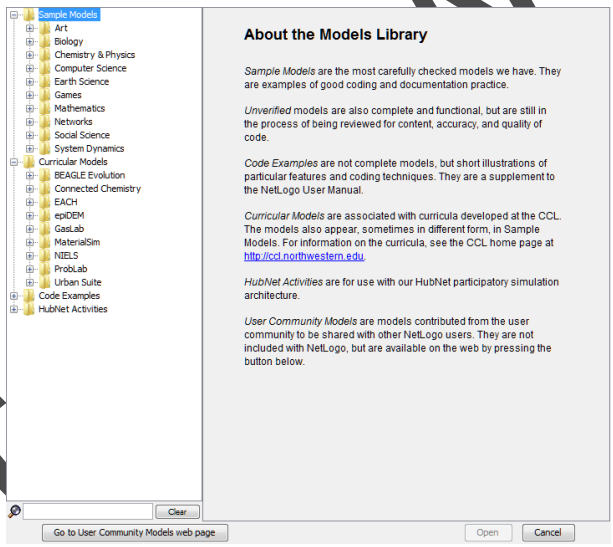


Figura 1.2: Menú de Ficheros.



- **Sample Models**: Contiene los modelos clasificados según el área de conocimiento al que pertenecen (Arte, Biología, Química y Física, Computación, etc.)

- **Curricular Models**: Agrupa aquellos modelos que se pueden encuadrar en un proyecto curricular determinado. Estos proyectos curriculares no tienen por qué englobarse dentro de un área de conocimiento concreta, sino que suelen ser transdisciplinares, y normalmente atienden a proyectos concretos que se están desarrollando en el *Center for Connected Learning and Computer-Based Modeling* (el mismo centro en el que se mantiene NetLogo)[25].

- **Code Samples**: presenta algunos modelos que tienen como objetivo destacar, y practicar, algunas de las características del lenguaje de programación, por lo que pueden ser usados como pequeñas muestras para profundizar en algunos conceptos clave del lenguaje.

- **HubNet Activities**: muestra algunos modelos que están preparados para ser usados como actividades colaborativas por medio de la herramienta HubNet de NetLogo.

[25] Algunos de los modelos que aparecen bajo esta clasificación pueden aparecer también en la clasificación anterior bajo alguna de las áreas de conocimiento.

Recomendamos que se empiece directamente por una exploración extensiva de los modelos que vienen en la carpeta **Sample Models**, tanto para ver la diversidad de posibles campos de aplicación como para hacerse una idea de la potencia que puede llegar a desarrollar este pequeño lenguaje de programación sin un esfuerzo excesivo.

ACTIVIDAD: Para ello, abre el modelo, lanza las simulaciones e interactúa con los controles que se han dispuesto en su interfaz, acude después a la pestaña de información para saber más acerca de cuál es el objetivo del modelo, conocer un poco acerca de cómo se ha programado y explorar las posibles extensiones que se proponen para profundizar, tanto en el modelo real que intenta explicar, como en las técnicas de programación que pueden ser útiles para abordarlo, pasa después a la pestaña de código para ver cómo se ha programado. Aunque todavía no sepas nada acerca del lenguaje de programación, intenta leerlo y sorpréndete con lo comprensible que resulta el código escrito en NetLogo.

Recursos en la web

Como hemos comentado, una de las grandes cualidades de NetLogo es su comunidad de usuarios (lo que ha llevado, tras un esfuerzo considerable de trabajo, a reprogramar completamente la herramienta para convertirla en una aplicación de código abierto), y que se traduce en la existencia de una gran cantidad de recursos disponibles en la web.

A continuación enumeramos solo aquellos que muestran más actividad, que se actualizan con más frecuencia, o que ofrecen un contenido

más cuidado[26]:

- El primer recurso fundamental es la propia página oficial de NetLogo[27], desde donde se puede acceder a una gran colección de recursos externos (algunos de los cuales enumeraremos aquí).

- Dentro de la página oficial destaca el manual web[28] que proporcionan (que también se instala localmente junto con la aplicación y al que se puede acceder por el menú de **Ayuda**) y que contiene un completo manual de referencia del lenguaje, introducción a la aplicación y algunos tutoriales básicos para adentrarse en su uso.

- Existe también un Grupo de Usuarios de NetLogo (alojado en Yahoo Grupos[29]) donde se pueden compartir dudas e ideas con otros usuarios de la comunidad, así como un Grupo de usuarios de NetLogo en el ámbito educativo[30].

- La página oficial también sirve de repositorio de modelos generados por la comunidad, de forma que se puedan descargar o ejecutar como applets en la misma página (aquellos para los que sea posible). Recientemente, se ha inaugurado un sitio en el que se pueden compartir modelos de NetLogo de forma colaborativa, su nombre es *NetLogo Modeling Commons* [31]

- En Yutzu[32] se puede encontrar un paquete que contiene información variada sobre la herramienta, así como un conjunto de videos (en inglés) que fueron diseñados a modo de tutorial por Carl Boettiger[33].

- Como ya dijimos, en GitHub se encuentra el repositorio asociado al código fuente de la aplicación[34] (sólo para usuarios avanzados con conocimientos en Java y/o Scala).

- *James Steiner*, uno de los mejores programadores de NetLogo, tiene una interesante biblioteca propia de modelos de NetLogo llamada TurtleZero[35], junto con una wiki que profundiza en algunas características del lenguaje que no se abordan en el manual oficial.

- INSISOC (Centro de Ingeniería de Sistemas Sociales) ofrece un pequeño manual/curso acerca de los fundamentos de NetLogo[36].

[26] Debe tenerse en cuenta que casi todo el material disponible en la actualidad se encuentra en inglés.

[27] *http://ccl.northwestern.edu/netlogo/*

[28] *http://ccl.northwestern.edu/netlogo/docs/*

[29] *http://groups.yahoo.com/group/netlogo-users/*

[30] *http://groups.yahoo.com/group/netlogo-educators/*

[31] *http://modelingcommons.org*. Desde la versión 5.0.4, NetLogo añade una opción en el menú de Ficheros para facilitar la subida directa de modelos a este repositorio.

[32] *http://www.yutzu.com/*

[33] Fueron diseñados para la versión 4 de NetLogo, pero siguen siendo válidos en un prcentaje muy amplio para la versión actual.

[34] *https://github.com/NetLogo*

[35] *http://www.turtlezero.com*

[36] *http://www.insisoc.org*. Es uno de los pocos recursos en español que se pueden encontrar.

Ejercicios

1. Abre la herramienta de creación de formas de tortugas y crea alguna que puedas usar posteriormente en los modelos que haremos a lo largo del libro.

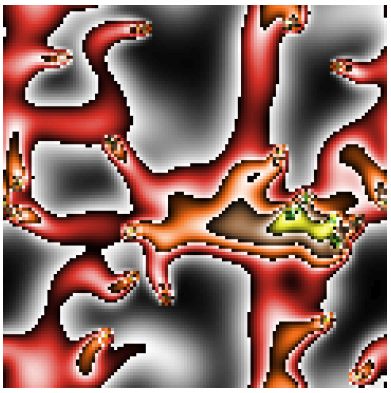


Figura 1.3: Diffusion Graphics.

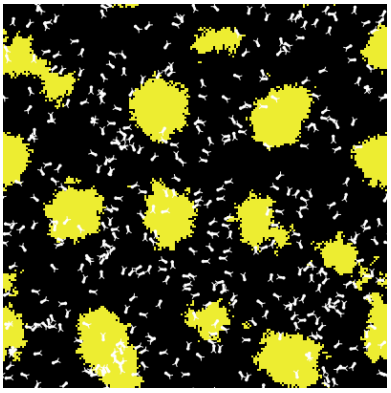


Figura 1.4: Termites.

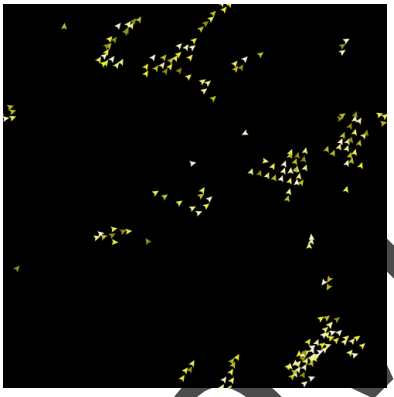


Figura 1.5: Flocking.

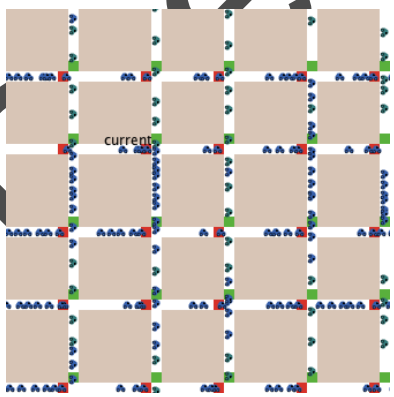


Figura 1.6: Traffic grid.

2. Visita el repositorio de NetLogo en GitHub y busca información acerca del proyecto y su comunidad de desarrolladores.

3. Biblioteca de modelos:

   a) **Diffusion Graphics** (figura 1.3):

      1) Localiza el modelo **Diffusion Graphics** que viene bajo la clasificación **Art** y cárgalo en NetLogo.
      2) Ejecuta varios experimentos cambiando el *número de tortugas* iniciales, el *coeficiente de difusión* y la *velocidad* de las tortugas.
      3) Abre la **pestaña de información** y lee las explicaciones y comentarios del modelo.
      4) Abre la **pestaña de Código** y échale un vistazo a los procedimientos y las instrucciones que aparecen en ellos.

   b) **Termites** (figura 1.4):

      1) Localiza el modelo **Termites** en el apartado **Biology** y cárgalo en NetLogo.
      2) Lee el apartado de información para hacerte una idea de lo que pretende el modelo y el papel que juegan los parámetros.
      3) Cuando una termita recoge madera ¿de qué color se muestra? Busca en la **pestaña de código** el lugar donde se asigna este color y haz que cambie a verde (`green`).

   c) **Flocking** (figura 1.5):

      1) Localiza el modelo **Flocking** en el apartado **Biology** y cárgalo en NetLogo.
      2) Lee el apartado de información para hacerte una idea de lo que pretende el modelo.
      3) ¿Cuál es el papel de cada uno de los seis parámetros?
      4) Comprueba que los valores de los parámetros cargados por defecto producen un comportamiento de organización en las tortugas y en un periodo de tiempo relativamente corto todas se orientan en la misma dirección.
      5) ¿Cómo varía el tiempo que tarda el grupo en ordenarse completamente respecto de los parámetros *visión* (`vision`) y *separación mínima* (`minimum-separation`)? ¿y respecto del *número de tortugas*?
      6) Una vez consensuada la orientación del vuelo, ¿qué ocurre si variamos el valor de los parámetros anteriores?

   d) **Traffic grid** (figura 1.6):

      1) Localiza el modelo **Traffic grid** que viene bajo la clasificación **Social Science** y cárgalo en NetLogo.

2) Inspecciona la **pestaña de información** para saber lo que hace el modelo y el significado de sus parámetros, así como del papel que juegan los diversos controles de su interfaz (**botones**, **sliders** y **switches**).

3) Observa las diferencias en la simulación dependiendo de si los semáforos están encendidos o no.

e) **Fire** (figura 1.7):

1) Localiza el modelo **Fire** en el apartado **Earth Science** y cárgalo. Este modelo simula la propagación de un fuego por un bosque.

2) Ejecuta el modelo para diferentes valores del **slider** `density` y observa la proporción del bosque que es quemado en cada caso.

3) Comprueba que existe un valor de *densidad crítica* (alrededor del 60 %) de modo que para densidades superiores a éste el incendio se propaga casi por todo el bosque, mientras que si la densidad permanece por debajo de él la proporción de bosque quemado es relativamente pequeña.

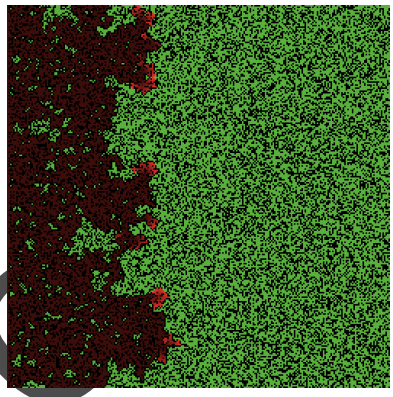


Figura 1.7: Fire.

4. Apúntate al grupo de usuarios de NetLogo y revisa las dudas y soluciones que se han producido en las últimas fechas.

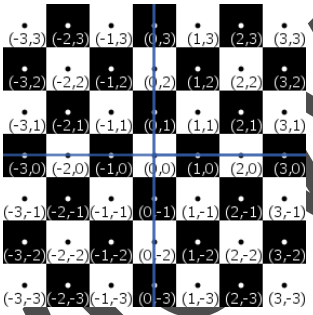
2
Familiarizándonos con NetLogo

El mundo de NetLogo

Si en Logo el elemento principal con el que se podía trabajar era la **tortuga**, en NetLogo esta interacción se extiende para poder operar con lo que se denominan genéricamente agentes[1].

NetLogo proporciona tres tipos de agentes:

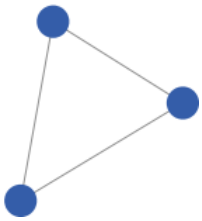
1. Los **patches** (**parcelas**) forman el terreno que define el mundo. Se distribuyen como una cuadrícula rellenando el mundo y son inmóviles.



2. Las **turtles** (o **tortugas**) son una extensión de la tortuga de Logo y pueden desplazarse por el mundo e interactuar con los patches del terreno y entre sí.

3. Los **links** (**enlaces**) definen relaciones entre agentes móviles (tortugas) a modo de aristas de un grafo. No son móviles, en el sentido de que no podemos desplazarlos individualmente por el mundo, ya que su posición depende exclusivamente de la posición de los agentes que relacionan.

[1] Llamamos **agente** a un individuo artificial, autónomo y dotado de reglas o características que gobiernan su comportamiento y su capacidad de tomar decisiones. Los agentes interaccionan entre sí y con el medio ambiente obedeciendo un conjunto de reglas. Son flexibles y tienen capacidad de aprender y adaptar su comportamiento basándose en la experiencia. Esta capacidad requiere alguna forma de memoria. Los agentes incluso pueden tener reglas para modificar sus reglas de comportamiento.



Por el momento nos centraremos en los patches y las tortugas, en un capítulo posterior profundizaremos en el uso de los links.

El mundo de los patches

Aunque hay instrucciones del lenguaje que permiten modificar las características del mundo (su tamaño y forma, en número de patches), por ahora vamos a acceder a sus propiedades por medio de la herramienta visual que proporciona el entorno.

Para ello basta pulsar sobre el botón **Settings...**, que se encuentra en la barra de herramientas de la pestaña **Interface**, o bien pulsar en el mundo con el botón derecho y seleccionar **Edit...**. Aquí podemos definir las dimensiones del mundo, la posición del origen de coordenadas, el tamaño visual del mundo (indicando cuántos pixels ocupará cada patch en la pantalla), la topología del mundo, el tamaño de letra de las etiquetas que usarán los agentes y algunas otras opciones en las que no entraremos por ahora (figura 2.1).

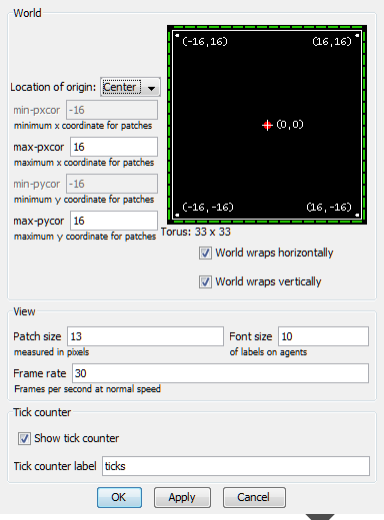
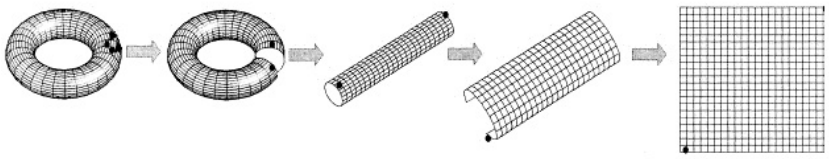


Figura 2.1: Ventana de edición de las características del mundo.

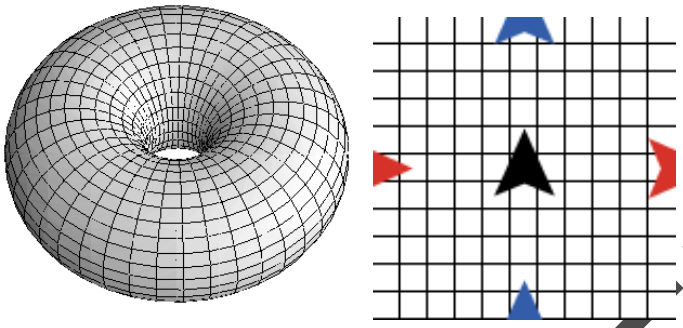
Respecto a la **topología** del mundo hemos de tener en cuenta lo siguiente: *los mundos que se pueden definir en NetLogo son siempre finitos, es decir, tienen un tamaño finito y un número finito de patches en su composición, pero no por ello tienen por qué ser limitados*. Es decir, podemos dar una configuración topológica al mundo de forma que no haya fronteras. Esto se consigue por medio de la identificación de sus extremos laterales: si identificamos los extremos izquierdo y derecho, obtenemos un cilindro vertical; si identificamos los extremos superior e inferior, obtenemos un cilindro horizontal; y si identificamos simultáneamente izquierdo con derecho y superior con inferior, obtenemos una figura que matemáticamente se conoce como un **toro**.



Una vez establecida alguna de estas identificaciones en los extremos, los agentes móviles que se desplacen más allá de alguno de ellos,

aparecerán por el lado opuesto, y los patches de cualquiera de los extremos tendrán como vecinos los patches del otro extremo identificado.

En las figuras siguientes se muestra una representación 3D de un toro, así como la conexión entre sus extremos en el mundo 2D. Se puede observar cómo la tortuga roja atraviesa el lado derecho y aparece por el izquierdo (identificación en el eje horizontal), mientras que la azul atraviesa el extremo superior y aparece por el inferior (identificación en el eje vertical).



Cada patch del mundo (como cualquier otro agente) tiene un conjunto de propiedades básicas. Para acceder a ellas por medio del interfaz, basta hacer click con el botón derecho sobre el patch que se desea inspeccionar, y seleccionar **Inspect patch...** (figura 2.2).

La ventana emergente nos muestra una visión local y ampliada del patch seleccionado y sus propiedades básicas: coordenadas en el mundo (**`pxcor`**, **`pycor`**), color (**`pcolor`**), etiqueta que muestra (**`plabel`**) y el color que utiliza esta etiqueta (**`plabel-color`**)[2]. Usando esta misma ventana podemos modificar algunas de las propiedades de los patches, aunque al ser agentes estáticos, no nos permitirá modificar sus coordenadas. De hecho, debido a que las coordenadas del patch son únicas (no hay otro patch en el mundo con las mismas coordenadas que él), éstas sirven para identificarlo y hacer referencia a él.

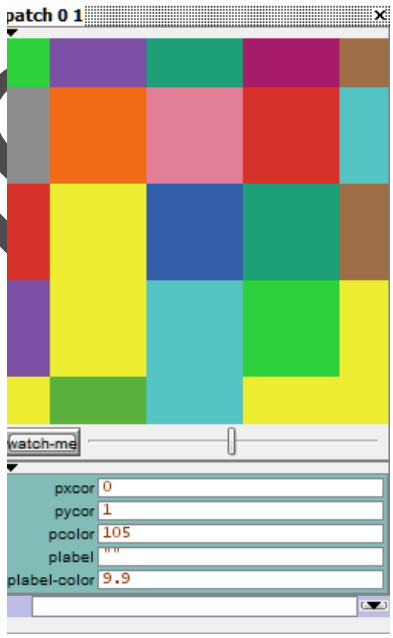


Figura 2.2: Ventana de inspección de patches.

Cuando se crea el mundo, los patches son de color negro (aunque en algunas de las representaciones que usaremos en este libro serán de color blanco); si queremos modificar su color basta asignarle un valor adecuado a la propiedad `pcolor` del patch.

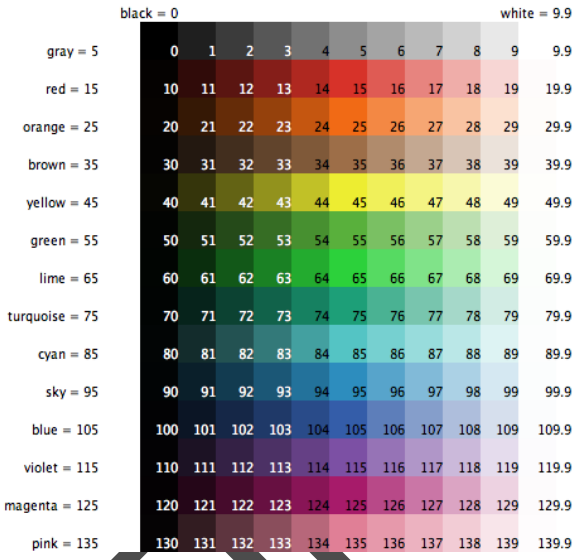
Aunque NetLogo es capaz de trabajar con la representación RGB de los colores[3], por defecto usa un sistema que es más simple y solo precisa de un número para identificarlos. Este sistema, aunque más limitado que RGB, ofrece algunas ventajas porque agrupa, cada 10 unidades, gamas del mismo color. La tabla siguiente muestra los valores de cada uno de los colores asociados (se muestran solo los valores enteros, pero admite decimales para colores intermedios). Además, para

[2] Más adelante veremos cómo definir nuevas propiedades, por ahora tenemos suficientes con las que el sistema trae por defecto.

[3] **RGB** es una representación habitual para colores cuando se trabaja con sistemas digitales, e indica la proporción de cada color primario por medio de tres números: **R**ed-Rojo, **G**reen-Verde y **B**lue-Azul, normalmente con valores entre 0 y 255. Así, una tonalidad de amarillo es $(255, 255, 0)$, el blanco es $(255, 255, 255)$ y el negro $(0, 0, 0)$.

los 14 colores básicos, NetLogo tiene constantes definidas para poder nombrarlos. Por ejemplo, a todos los efectos decir `orange` y `25` es lo mismo[4]:

[4] Si queremos obtener un color de la gama del verde más clara, basta escribir algo del tipo `(green + 2)`, un rojo oscuro puede ser `(red - 2)`. Se trata de un método directo para trabajar con colores al que le sacaremos mucho provecho.



Presentando a las tortugas

Pasemos ahora a conocer un poco mejor a las tortugas. A diferencia de los patches, que son automáticamente creados al iniciarse el mundo, las tortugas hay que crearlas manualmente para que aparezcan en el mundo, y se pueden crear en cualquier momento. Antes de ver cómo se crean y se manipulan, vamos a suponer que tenemos una tortuga en el mundo y analicemos sus características por defecto (figura 2.3).

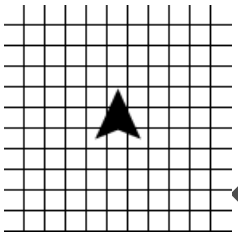
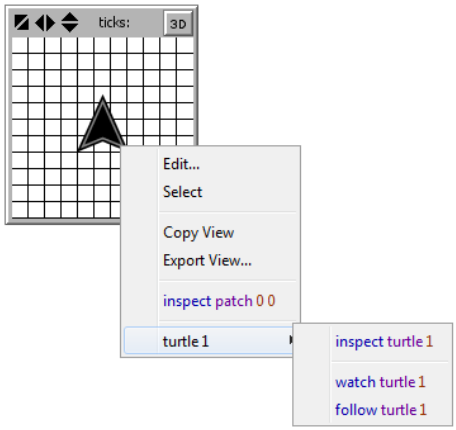


Figura 2.3: Tortuga recién creada en el centro del mundo.

Al hacer click con el botón derecho sobre la tortuga accedemos al menú contextual que nos muestra las tortugas que hay bajo el ratón (también aparecerá el patch sobre el que está la tortuga) para que seleccionemos la tortuga que corresponda.



Como queremos acceder a las propiedades de la tortuga, pulsamos sobre "**inspect turtle x**" (más adelante veremos para qué sirven las demás opciones), que abre una ventana similar a la que vimos para mostrar las propiedades del patch (figura 2.4).

Las propiedades que por defecto tiene una tortuga (algunas más que las que tienen los patches) son:

- **`who`**: a diferencia de los patches, que se identifican por su posición, las tortugas no tienen ninguna propiedad que las diferencie a priori de las demás. Por tanto, NetLogo proporciona para cada tortuga del mundo un identificador único por medio de este número, que se incrementa secuencialmente y sin repetición. Por tanto, la primera tortuga tendrá como identificador `0`, la siguiente que se cree tendrá `1`, y así sucesivamente.
- **`color`**: el color de la tortuga según el código de colores que sigue NetLogo. Similar al color visto para los patches.
- **`heading`**: la orientación de la tortuga en el plano. Se mide en grados, y se debe tener en cuenta que no sigue los mismos criterios que la trigonometría habitual, sino que comienza a contar desde el eje positivo de la Y y crece en sentido horario (figura 2.5).
- **`xcor`**, **`ycor`**: coordenadas de la posición que ocupa en el espacio. Se mide en unidades de patches pero, a diferencia de estos, puede tomar valores decimales, que representan todos los valores intermedios entre las coordenadas de dos patches consecutivos (figura 2.6).
- **`shape`**: forma de la tortuga. Por defecto es la que se muestra en las imágenes, pero los modelos vienen con diversas formas que pueden utilizarse y con un editor de formas para crearlas.

- **`label`**: etiqueta que la tortuga mostrará en pantalla (si no es vacía)[5].
- **`label-color`**: color de la etiqueta.

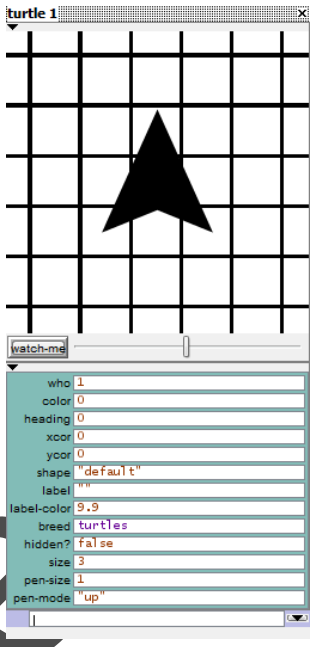


Figura 2.4: Ventana de inspección de tortugas.

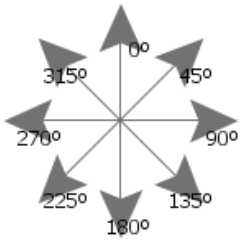


Figura 2.5: Sistema de orientación de NetLogo.

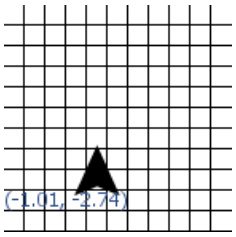


Figura 2.6: Coordenadas de la tortuga.

[5] En la figura 2.6 se muestran las coordenadas de la tortuga por medio de su etiqueta.

- **breed**: familia de tortugas a la que pertenece. Esta característica de NetLogo es uno de sus puntos fuertes, pues nos permite definir familias de agentes que pueden tener distintos conjuntos de propiedades y comportamientos. Lo veremos en un capítulo posterior.

- **hidden?**: indica si la tortuga está oculta o visible. Por defecto las tortugas son visibles, pero pueden ocultarse individualmente en la representación del mundo.

- **size**: determina el tamaño de la tortuga (en patches). Su valor por defecto es 1.

- **pen-mode**: como en otras versiones de Logo, las tortugas pueden dejar un rastro en el suelo mientras se mueven. Los posibles valores de este parámetro controlan esta opción: `up` (valor por defecto), el lápiz de dibujo de la tortuga está en alto y no deja rastro; `down`, el lápiz de dibujo de la tortuga está bajado y sí deja rastro; `erase`, la tortuga baja la goma de borrar y elimina, en su camino, los posibles rastros que pueda cruzarse.

- **pen-size**: Este parámetro determina la anchura, en pixels, del lápiz que se usa para dibujar el rastro. Su valor por defecto es 1 (figura 2.7).

Figura 2.7: Una figura realizada con variaciones de tamaño de lápiz.

Al igual que en el caso de los patches, podemos cambiar los valores de sus propiedades por medio de esta ventana, aunque lo normal será que lo hagamos por medio de instrucciones del lenguaje de programación. Como una tortuga es un agente móvil, podremos cambiar sus coordenadas espaciales (lo que provocará que la tortuga aparezca instantáneamente en la nueva posición), pero la propiedad `who` no admite modificación ya que es su identificador.

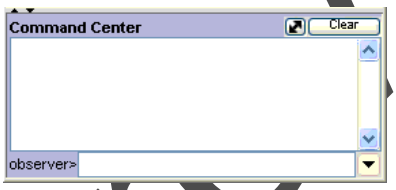
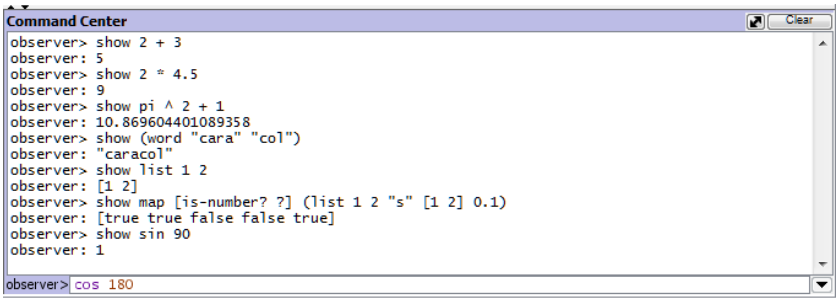


Figura 2.8: El centro de comandos ocupa, por defecto, la zona horizontal inferior de la pestaña de Interfaz, aunque podemos cambiarla a la zona vertical derecha. Su tamaño se puede cambiar arrastrando la barra que lo separa del interfaz, y su contenido se puede borrar por medio del botón **Clear** que proporciona.

Centro de Comandos

En esta sección vamos a interactuar con NetLogo por medio del **Centro de Comandos**, lo que nos permitirá ir adentrándonos en el lenguaje de programación de una forma interactiva, sin necesidad de escribir procedimientos completos ni conocer demasiados secretos del lenguaje (figura 2.8).

Por medio de este centro podemos comunicarnos con NetLogo escribiendo instrucciones que se ejecutarán al pulsar la tecla de retorno de carro. Comenzaremos con unos ejemplos sencillos, usándolos para mostrar el tipo de datos que es capaz de manejar NetLogo, es decir, a modo de calculadora extendida:



El Centro de Comandos se compone de una barra inferior, donde se introducen los comandos, y un área donde se muestran los resultados de aquellas instrucciones que producen alguna devolución. A todos los efectos se comporta como una consola de texto similar a las de otros sistemas.

Como muestra la linea inferior, el sistema colorea automáticamente las instrucciones en función de si reconoce el texto como una instrucción válida, como una constante, etc. Además, aunque no es necesario añadir el comando **`show`**, que hace que se muestre el resultado, el intérprete lo añade automáticamente a nuestra petición en caso de que de la instrucción devuelva un resultado y no sea una acción[6].

[6] Veremos que las instrucciones de NetLogo se dividen en dos grandes grupos: las **acciones**, que no devuelven nada pero producen cambios en el mundo, y las funciones, (**reportes**) que devuelven valores.

Trabajando con números. Números aleatorios

Aprovecharemos algunas funciones matemáticas de NetLogo que serán útiles más adelante para practicar la sintaxis del lenguaje en el intérprete.

Importante: **Todos los operadores en NetLogo deben ir separados con espacios. Lo que quiere decir que** `sin(x+1)` **debe escribirse:** `sin (x + 1)`.

NetLogo no diferencia internamente entre números enteros y decimales, sino que todos ellos se almacenan internamente como números de coma flotante de doble precisión[7]. Muchas veces, si una función espera como dato de entrada un entero y le damos un número con parte decimal, simplemente ignorará la parte decimal.

El rango de números con los que puede trabajar es, aproximadamente, de -2^{53} a 2^{53}, es decir, `+/-9007199254740992`. Si un cálculo excede esta cantidad no devuelve un error, pero el resultado puede no ser preciso. Por ejemplo:

[7] En consecuencia, un número entero para NetLogo es, simplemente, un número que no tiene parte decimal, de forma que `1` y `1.0` se consideran iguales, algo que no es así en otros muchos lenguajes. Lo que sí ocurre es que NetLogo oculta las terminaciones `.0` para facilitar la identificación de los enteros.

```
observer> 2 ^ 52 + 1 = 2 ^ 52
observer: false
observer> 2 ^ 53 + 1 = 2 ^ 53
observer: true
```

También puede haber problemas de precisión con cantidades fraccionarias[8]. Por ejemplo:

[8] Este es un problema general en todas las herramientas basadas en Java.

```
observer> 1/6 + 1/6 + 1/6 + 1/6 + 1/6 + 1/6
observer: 0.9999999999999999
```

Y, como es habitual en casi todos los lenguajes de programación, una división por `0` devuelve un error de ejecución.

Respecto a los números aleatorios, y al igual que el resto de lenguajes de programación, NetLogo trabaja con números **pseudo-aleatorios**[9]. La forma de conseguir que no siempre se obtengan los mismos valores es hacer depender esta función generadora de un valor inicial (*semilla*, *seed*), lo que significa que si se empieza siempre con la misma semilla, se obtiene siempre el mismo número[10]. Para resolver este problema normalmente se oculta al usuario la necesidad de dar una semilla haciendo uso, por ejemplo, de la hora del ordenador (que es una cantidad que cambia continuamente). La secuencia de instrucciones siguiente puede servir para entender este fenómeno:

```
observer> random 100              => 9
observer> random 100              => 1
observer> random 100              => 44
observer> random-seed 1
observer> random 100              => 81
observer> random-seed 1
observer> random 100              => 81
```

Como se puede deducir de las líneas anteriores, la función **`random`** recibe como dato de entrada un número, `n`, y devuelve un número entero pseudo-aleatorio en `[0,n)`[11] (o en `(n,0]` si `n < 0`) siguiendo una probabilidad uniformemente distribuida. Existe una versión no entera (de coma flotante), llamada **`random-float`**, que opera sobre números decimales. La función **`random-seed`** permite fijar la semilla del generador de números pseudo-aleatorios.

Debido a que el último valor no se devuelve nunca, la instrucción que devuelve un número entero al azar entre `1` y `10` sería `(1 + random 10)`. La fórmula general para obtener un número entero al azar entre `a` y `b` sería:

```
a + random (b - a + 1)
```

Por ejemplo, para generar un número entero entre -5 y 5 la instrucción sería: `(random 11) - 5`

¿Y qué ocurre si queremos que el número devuelto sea decimal? Por ejemplo, si queremos obtener al azar un número decimal entre `1` y `10`, la instrucción sería: `1 + random-float 9`. Y la fórmula general para devolver un número decimal al azar entre `a` y `b` sería[12]:

```
a + random-float (b - a)
```

Por lo que la generación de un número decimal al azar entre -5 y 5 quedaría: `(random-float 10) - 5`.

Además de la distribución uniforme que proporcionan las funciones anteriores, NetLogo ofrece generadores de números pseudo-aleatorios que siguen otras distribuciones habituales[13]. Pero no son las únicas que hacen uso de números pseudo-aleatorios, ya veremos a lo largo

[9] Los números pseudo-aleatorios, aunque parecen aleatorios, se generan por medio de una función determinista que devuelve números dentro de un rango con probabilidad uniforme.

[10] Aunque esta característica pueda parecer contraproducente, cuando estamos realizando modelado científico se convierte en una cualidad que podemos aprovechar, ya que hace que todos los experimentos sean reproducibles almacenando las semillas con las que se han realizado.

En lo que sigue, y con el fin de dar una representación más compacta de las interacciones, usaremos la notación => para mostrar las salidas del Centro de Comandos.

[11] Es decir, uno de: `0,1,2,...,n-1`.

[12] ¿Puedes explicar la razón por la que las expresiones son distintas dependiendo de si quieres decimales o no?

[13] Algunas de estas funciones son: **`random-poisson`**, **`random-exponential`**, **`random-normal`** y **`random-gamma`**.

de este libro funciones que realizan una selección *al azar* de entre un conjunto de objetos, y todas ellas usan internamente el mismo generador de números aleatorios a partir de una semilla, por lo que su efecto también puede ser reproducible.

ACTIVIDAD: Utiliza el Centro de Comandos como calculadora para realizar algunas operaciones numéricas y acostumbrarte a las peculiaridades de la sintaxis de NetLogo.

Trabajando con cadenas de texto

El segundo gran tipo de dato con el que puede trabajar NetLogo son las *cadenas de texto*. Una **cadena** (**string**) es simplemente un conjunto de caracteres encerrados entre comillas[14]. Por ejemplo `"esto es una cadena"`[15]. Para representar la cadena vacía se usa `""`[16].

A continuación mostramos algunas de las funciones más habituales para trabajar con cadenas:

```
observer> length "Perro chino"              => 11
observer> first "Perro chino"               => "P"
observer> but-first "Perro chino"           => "erro chino"
observer> last "Perro chino"                => "o"
observer> but-last "Perro chino"            => "Perro chin"
observer> item 4 "Perro chino"              => "o"
observer> item 0 "Perro chino"              => "P"
observer> substring "Perro chino" 3 5       => "ro"
observer> member? "i" "Perro chino"         => true
observer> member? "mi" "Perro chino"        => false
observer> position "e" "Perro chino"        => 1
observer> remove " " "Perro chino"          => "Perrochino"
observer> remove "s" "Perro chino"          => "Perro chino"
observer> replace-item 5 "Perro chino" "-"=> "Perro-chino"
observer> reverse "Perro chino"             => "onihc orreP"
observer> word "Perro" "chino"              => "Perrochino"
```

Algunas cosas que conviene resaltar:

1. La primera posición de los caracteres en las cadenas es `0`.
2. Cuando se trabaja con cadenas y caracteres, se diferencia entre mayúsculas y minúsculas.
3. La forma de concatenar cadenas es por medio de la instrucción **`word`**.

ACTIVIDAD: Toma una frase cualquiera, exprésala como una cadena de texto, y ejecuta las instrucciones necesarias para poder separarla por palabras.

[14] Desde la versión 5.0, NetLogo puede trabajar con caracteres *Unicode*.

[15] NetLogo no diferencia entre cadenas, palabras o caracteres. Únicamente la longitud de la cadena y la existencia o no de espacios hace que nosotros lo interpretemos de una forma u otra.

[16] La cadena vacía ya la hemos encontrado en las etiquetas de patches y tortugas para generar etiquetas vacías (o lo que sería equivalente, no usarlas.

Si necesitas insertar algún carácter especial (tabulación, salto de línea, etc.) en una cadena haz uso de una **secuencia de escape**:

1. `\n` = salto de línea
2. `\t` = tabulación
3. `\"` = comillas dobles
4. `\\` = backslash

Trabajando con listas. Datos compuestos

Cuando trabajemos con bloques de datos sencillos puede ser suficiente el uso de números y cadenas, pero muchas veces será necesario almacenar o manipular muchos valores simples de una vez. En este caso la mejor solución es trabajar con listas.

Una **lista** es una estructura ordenada que almacena muchas unidades de información. Por ejemplo, `[0 2 4 6 8]`[17] es la lista de los pares menores que `10`. Una característica interesante de las listas es que no tienen que ser homogéneas, sino que pueden contener distintos tipos de datos dentro, incluso otras listas. Por ejemplo:

```
[0 2 “cara” 6 [1 3 5] “col”]
```

Veremos que las listas se convierten en aliados muy eficientes para trabajar de manera muy cómoda con información estructurada [18].

Las listas constantes (aquellas que no varían) se consiguen escribiéndolas directamente entre corchetes[19]:

```
observer> [0 2 4 6 8]
observer> [0 2 "cara" 6 [1 3 5] "col"]
observer> []
```

Pero si quieres formar una lista usando en alguno de sus términos el resultado de una función, o un valor no constante, necesitamos un constructor para poder formarla. En NetLogo este constructor es **`list`**[20]:

```
observer> list (random 10) (random-float 10)
observer: [2 2.04452253035499]
observer> (list (random 10) (random-float 10) 5)
observer: [8 0.27387597179620027 5]
observer> (list (random 10))
observer: [4]
```

Una forma muy útil de construir listas es por medio de la función **`n-values`**, que permite generar una lista de una determinada longitud por medio de una función. Por ejemplo:

```
observer> n-values 10 [2 * ?]
observer: [0 2 4 6 8 10 12 14 16 18]
```

construye una lista de tamaño `10` y para rellenarla usa la función $2x$[21]. El formato general es:

```
n-values k [ f(?) ]
```

que genera la lista `[f(0) f(1) ... f(k-1)]`.

Algunos ejemplos de su uso son:

```
observer> n-values 5 [0]
observer: [0 0 0 0 0]
observer> n-values 5 [?]
observer: [0 1 2 3 4]
```

[17] En otros lenguajes de programación se usan los paréntesis para delimitar las listas. NetLogo usa corchetes.

[18] También veremos que son el formato natural para manipular la información proveniente de agentes o grupos de agentes.

[19] Observa que la última de ellas es la lista que no tiene elementos, que se llama **lista vacía**.

[20] Observa que si quieres formar una lista con más de dos datos (o solo con uno) debes incluir la instrucción completa entre paréntesis. Veremos que hay algunas otras instrucciones de NetLogo que admiten una cantidad indeterminada de entradas, en todas ellas será necesario incluir la instrucción completa entre paréntesis en caso de que el número de entradas sea distinto de 2.

[21] Observa el uso de la variable especial, `?`, como variable de la función que se usa.

```
observer> n-values 3 [sin (? * 360 / 3)]
observer: [0 0.8660254037844387 -0.8660254037844385]
```

La mayoría de las instrucciones que vimos para manipular cadenas funcionan con listas:

```
observer> length [0 1 2 3 4 5]              => 6
observer> first [0 1 2 3 4 5]               => 0
observer> but-first [0 1 2 3 4 5]           => [1 2 3 4 5]
observer> last [0 1 2 3 4 5]                => 5
observer> but-last [0 1 2 3 4 5]            => [0 1 2 3 4]
observer> item 4 [0 1 2 3 4 5]              => 4
observer> sublist [0 1 2 3 4] 2 4           => [2 3]
observer> member? 1 [0 1 2 3 4 5]           => true
observer> position 1 [0 1 2 3 4 5]          => 1
observer> remove 1 [0 1 2 3 4 5]            => [0 2 3 4 5]
observer> replace-item 1 [0 1 2 3 4] "a"    => [0 "a" 2 3 4]
observer> reverse [0 1 2 3 4 5]             => [5 4 3 2 1 0]
```

Las posiciones de las listas también comienzan con `0`.

Pero, además, tenemos un conjunto de funciones más amplio que se pueden aplicar, por ejemplo, a listas numéricas:

```
observer> max [1 2 3 4 5 4 3 2 1]       => 5
observer> min [1 2 3 4 5 4 3 2 1]       => 1
observer> sum [1 2 3 4 5 4 3 2 1]       => 25
observer> variance [1 2 3 4 5 4 3 2 1]  => 1.9444444444446
observer> mean [1 2 3 4 5 4 3 2 1]      => 2.77777777777777
observer> modes [1 2 3 4 5 4 3 2 1]     => [1 2 3 4]
observer> sum [1 2 3 4 5 4 3 2 1 "a"]   => 25
```

Según el último ejemplo, las funciones que esperan listas numéricas ignoran los elementos que no sean números.

ACTIVIDAD: Prueba los siguientes ejemplos y razona por qué cada uno funciona de forma distinta.

```
observer> sum ["a" "b"]
observer> max ["a" "b"]
```

Además, podemos ampliar las listas añadiendo elementos a cualquiera de sus dos extremos[22]:

```
observer> lput "a" [1 2 3]                  => [1 2 3 "a"]
observer> fput "a" [1 2 3]                  => ["a" 1 2 3]
```

[22] Estas primitivas serán muy importantes, pues junto con `first`, `but-first`, `last` y `but-last` nos permiten construir y deconstruir listas de forma sistemática.

La forma de concatenar listas es haciendo uso de la instrucción **`sentence`**.

ACTIVIDAD: Deduce a partir de los siguientes ejemplos cómo funciona y en qué se diferencia de `list`.

```
observer> list [1 2 3] [4 5]           => [[1 2 3] [4 5]]
observer> sentence [1 2 3] [4 5 6]     => [1 2 3 4 5 6]
observer> list 1 [4 5 6]               => [1 [4 5 6]]
observer> sentence 1 [4 5 6]           => [1 4 5 6]
observer> (sentence 1 [4 5 6] 7)       => [1 4 5 6 7]
observer> (sentence [1 2 3] [4 5 6] 7) => [1 2 3 4 5 6 7]
observer> (sentence [1 2] [4 [5] 6] 7) => [1 2 4 [5] 6 7]
```

Veremos en un capítulo posterior cómo sacarle potencia a las listas por medio de programación, viendo cómo podemos diseñar procedimientos recursivos para resolver problemas complejos[23].

[23] Algunos de los cuales podrán ser escritos de forma muy concisa gracias a algunas instrucciones adicionales que NetLogo proporciona.

Trabajando con lógica. Tipo Booleano

Al igual que el resto de lenguajes de programación, NetLogo también está preparado para trabajar con los valores booleanos **`true`**/**`false`** que se obtienen por operaciones condicionales, como son las comparaciones[24]:

[24] También se pueden realizar comparaciones entre cadenas, donde automáticamente se aplica el orden lexicográfico carácter a carácter.

```
observer> 3 > 2                  => true
observer> "a" > "b"              => false
observer> "a" < "b"              => true
observer> n-values 4 [? < 3] => [true true true false]
```

ACTIVIDAD: Escribe la instrucción que sea capaz de decidir qué términos de la lista `[0 1 2 3 4 5 6]` son pares y cuáles no.

Reconociendo el tipo de dato

Para poder saber si un dato es de un tipo u otro de forma automática, NetLogo proporciona algunas funciones booleanas que reconocen el tipo[25]:

[25] Veremos que este tipo de reconocedores existirán incluso para poder saber si un objeto es un agente de tipo tortuga, un patch, un link, etc.

```
observer> is-number? 3     => true
observer> is-number? "a"   => false
observer> is-string? "a"   => true
observer> is-string? 3     => false
observer> is-list? 3       => false
observer> is-list? [3]     => true
observer> is-boolean? 3    => false
observer> is-boolean? 0    => false
observer> is-boolean? true => true
```

ACTIVIDAD: Comprueba de qué tipo son los siguientes datos: `[true false]`, `[[1]]`, `1 < 2`.

Contextos de ejecución

ACTIVIDAD: Si hasta ahora no te lo has preguntado, es hora de hacerlo, ¿qué función tiene la aparición de `observer>` junto a la barra de introducción de comandos? Si pulsamos sobre él nos resultará más fácil entender su significado (figura 2.9).

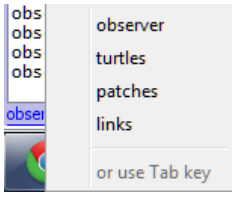


Figura 2.9: Contextos del Centro de Comandos.

Fijémonos que de las cuatro opciones que aparecen, tres de ellas son los diversos tipos de agentes que tiene NetLogo, mientras que el cuarto es **`observer`**.

En NetLogo la ejecución está orientada a los agentes, es decir, las diferentes instrucciones del lenguaje, así como los procedimientos que nosotros vayamos definiendo, las pueden ejecutar cualquiera de los agentes... o podemos ejecutarlas nosotros. El papel del *observador*, que solo hay uno, es el que jugamos nosotros, externos al mundo artificial que se ejecuta pero con la capacidad de controlarlos a todos.

Lógicamente, no todos los tipos de agentes pueden ejecutar todas las instrucciones del lenguaje ya que puede depender, por ejemplo, de si la ejecución afecta a propiedades que no le corresponden. Por ejemplo, si ejecutamos `set color blue`, que establece el color a azul, esta instrucción sólo tiene sentido en el contexto de los agentes móviles, puesto que son ellos los que tienen esta propiedad, pero ni los patches ni el observador podrán ejecutarla[26].

La figura 2.10 puede aclarar la diferencia. ¿Qué ha ocurrido al ejecutar "`1 + 1`" en el contexto de los `patches`? Si lo hubiéramos hecho en el contexto `observer` el sistema hubiera respondido simplemente `observer: 2`, pero al haberlo ejecutado en el contexto de los patches lo que realmente se ha hecho es que cada patch ha ejecutado la operación y ha devuelto el resultado.

[26] Aunque ya mostraremos las formas que existen de que un agente/observador le pida a otros agentes realicen operaciones.

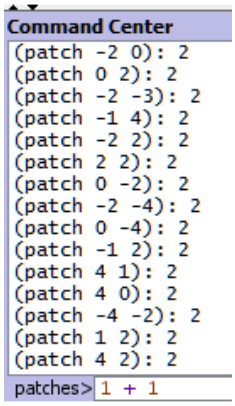


Figura 2.10: Ejecución de un comando por los patches. En la ventana se muestran los últimos patches en su devolución, por ejemplo, el patch que está situado en las coordenadas `(4, 2)` devuelve: `(patch 4 2): 2`, al igual que el patch de coordenadas `(1,2)`. No te preocupes si al ejecutarlo has obtenido otro orden en la lista de patches, más adelante sabremos porqué.

ACTIVIDAD: Vamos a aprovechar esta característica para hacer algunas pruebas sobre patches. Para ello, ejecuta las siguientes instrucciones y justifica los resultados obtenidos (aunque todavía no hayamos visto algunas de las instrucciones usadas, es fácil suponer qué hacen por su significado en inglés).

```
patches> set pcolor blue
patches> set pcolor one-of base-colors
patches> set plabel (word "(" pxcor "," pycor ")")
patches> set pcolor [pcolor] of one-of neighbors
patches> set plabel pxcor + world-width * pycor
```

A continuación vamos a crear una tortuga para jugar un poco con ella en el mundo[27]:

[27] Ten la precaución de cambiar el contexto del intérprete a `observer`, ya que las instrucciones siguientes no las pueden ejecutar los agentes

- `observer> clear-all`: Comenzamos borrando todo para partir de un mundo limpio[28].

- `observer> crt 1`: Después creamos una tortuga (**`create-turtles`**) donde le indicamos el número de tortugas a crear. El efecto de esta instrucción es la creación de una tortuga que, por defecto, se sitúa en el origen de coordenadas pero con orientación (**`heading`**) inicial y color aleatorios.

[28] Aprovechamos este momento para explicar que muchas de las instrucciones de NetLogo (sobre todo las más habituales) admiten una versión resumida, por ejemplo, **clear-all** y **ca** tienen el mismo efecto.

Como ya existe una tortuga en el mundo, podemos pasar al contexto `turtles` para manejar la tortuga por medio de las instrucciones básicas de movimiento (que son similares a las existentes en otros lenguajes Logo). La figura siguiente muestra una ejecución interactiva en la que cada línea se ha introducido en orden (cambiando adecuadamente entre contextos tal y como se ha indicado).

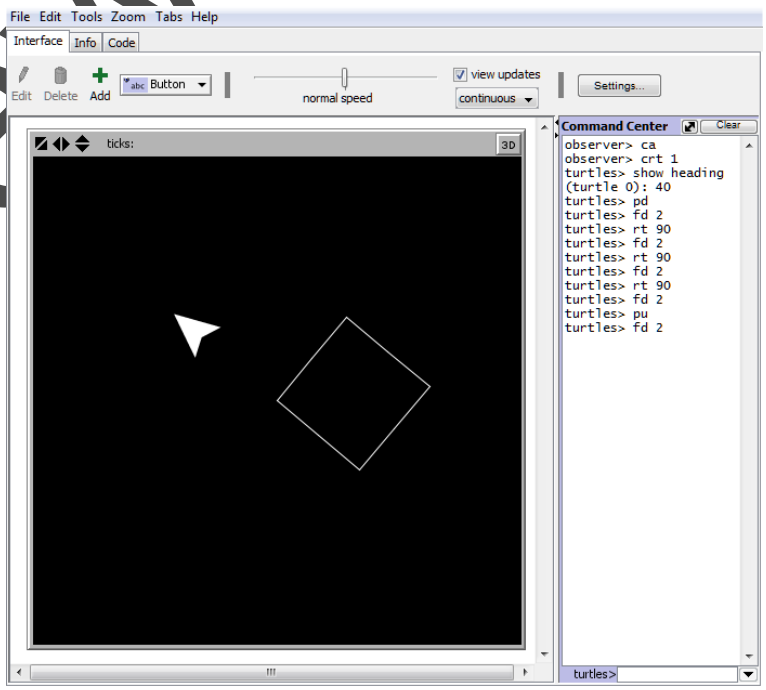
El significado de las instrucciones desconocidas es:

1. **`fd n`** : (**`forward`**) hace que la tortuga avance `n` unidades (en patches).

2. **`rt n`** : (**`right`**) hace que la tortuga gire a la derecha (*su* derecha) `n` grados.

3. **`pd`** : (**`pen-down`**) baja el lápiz de la tortuga para que deje rastro en su movimiento.

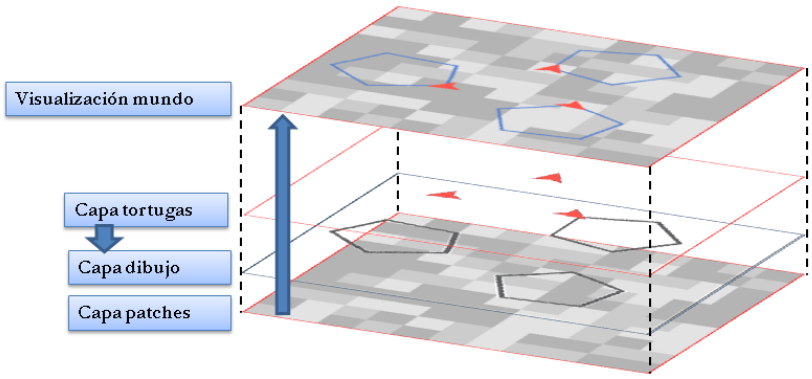
4. **`pu`** : (**`pen-up`**) sube el lápiz de la tortuga para que no deje rastro en su movimiento.



Observa que al ejecutar `heading` la tortuga ejecuta realmente la instrucción `show heading` que da como resultado que muestre en la ventana el valor de su propiedad heading.

Es importante hacer notar que los dibujos que puedan realizar las tortugas cuando tienen el lápiz bajado se almacenan en una capa distinta, la capa de dibujo, con la que no pueden interactuar los agentes[29].

[29] Esta capa puede borrarse por medio de **`clear-drawing`**, o **`cd`** en su versión abreviada.



A diferencia de otros Logos, en NetLogo no existen instrucciones de rellenado (fill) de áreas, y la forma de fijar un color de fondo es por medio del coloreado de los patches.

Otras instrucciones que pueden ser útiles para el manejo de la tortuga son:

1. **`bk n`** : (**`back`**) hace que la tortuga retroceda (camine hacia atrás) `n` patches.
2. **`ht`** : (**`hide-turtle`**) oculta la tortuga (pero si se mueve y tiene el lápiz bajado, sigue dejando un rastro).
3. **`home`** : lleva la tortuga al origen de coordenadas, manteniendo su orientación y dejando rastro en caso de que tuviera el lápiz bajado.
4. **`lt n`** : (**`left`**) gira `n` grados hacia *su* izquierda.
5. **`pe`** : (**`pen-erase`**) baja la goma de borrar de la tortuga, por lo que su movimiento borrará los rastros dejados con los que se cruce.
6. **`st`** : (**`show-turtle`**) muestra la tortuga (útil si antes ha sido ocultada).
7. **`stamp`** : hace que la tortuga deje una marca en el mundo con su forma. Esa marca es únicamente un dibujo en el suelo, no una tortuga real.
8. **`move-to patch x y`** : hace que la tortuga se desplace al patch con coordenadas `(x,y)`[30].
9. **`die`** : elimina a la tortuga del mundo.

[30] Hay usos más generales del comando `move-to` para situar a la tortuga en la posición que ocupa cualquier otro agente, por ejemplo: `move-to one-of patches`.

10. **`hatch n`** : hace `n` copias exactas de la tortuga (todas ellas con exactamente los mismos valores en todas las propiedades).

Ejercicios

A pesar de que hasta ahora solo hemos acariciado la superficie de las capacidades de NetLogo ya podemos hacer algunas cosas interesantes[31]:

1. **Diferentes topologías**[32]:

   a) Crea una tortuga y deselecciona en la configuración del mundo la opciones **mundo sin límite horizontal** y **mundo sin límite vertical**.

   b) Establece los valores `max-pxcor` y `max-pycor` para que el mundo tenga tamaño 21×21.

   c) Orienta la tortuga hacia al norte, y desplázala en esa dirección. ¿Qué ocurre con la tortuga cuando la desplazamos una cantidad mayor que el tamaño del mundo?

   d) Devuelve la tortuga al origen de coordenadas, oriéntala hacia el este y comprueba que ocurre lo mismo que en el caso anterior.

   e) Haz lo mismo pero marcando esta vez las opciones **mundo sin límite horizontal** y/o **mundo sin límite vertical**[33].

   f) Teniendo en cuenta las tres topologías que proporciona NetLogo (*plano euclídeo*, *cilindro* y *toro*), intenta prever cuál sería el camino más corto entre dos puntos en cualquiera de las topologías.

   g) Para dimensiones fijas del mundo, si supones un toro, ¿eres capaz de deducir alguna regla que te permita que una tortuga que se mueve en línea recta vuelva alguna vez al punto de origen?

   h) Después de haber dibujado algunas figuras sobre el cilindro o el toro, ¿puedes imaginar el aspecto que tendrían sobre la representación 3D de la superficie?

2. **Dibujo de un polígono**[34]:

   a) Crea un tortuga desde el centro de comandos.

   b) Usa el inspector de tortugas para cambiar el color, tamaño y forma de la tortuga[35].

   c) Dibuja con la tortuga un pentágono de lado 4, de color azul, y anchura de trazo igual a 2 pixels. Es posible que tengas que ampliar el tamaño del mundo.

   d) Calcula el ángulo de giro para dibujar un polígono regular de n lados cualquiera.

[31] Si nunca has trabajado con tortugas, la mejor forma de familiarizarte con su forma de trabajo es interactuando con ellas y viendo cómo responden a las distintas órdenes en tiempo real.

[32] La *Topología* es la rama de las matemáticas dedicada al estudio de las propiedades de los cuerpos geométricos que permanecen inalteradas por transformaciones continuas. Es decir, aquellas que siguen siendo ciertas aunque el mundo lo estires y deformes, siempre y cuando no lo rompas ni pegues dos partes separadas, por eso a veces se le conoce como *Geometría de la lámina elástica*.

[33] Hazlo con todas las posibles combinaciones.

[34] La **geometría intrínseca** de la tortuga, algo en lo que profundizaremos más adelante, hace que su manipulación permita entender las figuras geométricas independientemente de un sistema de referencia rígido, por lo que su uso, decidiendo el mejor sistema de referencia para cada figura que queramos hacer, puede simplificar el proceso de generación de las mismas.

[35] Recuerda que si quieres saber las formas disponibles en tu modelo, puedes abrir la herramienta que NetLogo trae para trabajar con formas. Intenta importar una nueva forma de la biblioteca.

3. **Caminos aleatorios**[36]:

   a) Crea una tortuga, activa el rastro y elige un color y un ancho de trazo.

   b) Utiliza el comando `random` para orientar a la tortuga con un ángulo aleatorio entre -90 y 90 grados, y desplázala un paso hacia delante.

   c) Repite la acción varias veces y observa el camino aleatorio descrito.

   d) Haz lo mismo pero restringiendo el ángulo de giro de otro modo y avanzando una longitud aleatoria, pongamos entre 0 y 10 unidades.

4. **Contexto patches**:

   a) Selecciona el contexto `patches` en el centro de comandos.

   b) Consigue diferentes patrones de colores en los patches usando las coordenadas de los mismos y la instrucción `set`. Prueba con diferentes fórmulas[37].

   c) Utiliza `random` para asignar colores aleatorios a los patches en la gama de los verdes.

   d) Utiliza la operación `x mod 2`, que devuelve el resto de x entre 2, para dibujar un damero de dos colores asignando a cada patch un color según el valor de `(pxcor + pycor) mod 2`. Investiga los efectos si cambiamos 2 por otro número natural.

5. **Creando listas**: escribe instrucciones para crear las siguientes listas:

   a) La lista de los 100 primeros números naturales.

   b) La lista de los 100 primeros números pares.

   c) La lista de los 100 primeros números impares.

   d) La lista de las potencias de 2 hasta donde permita Netlogo.

   e) A partir de una palabra fija, genera una lista de letras elegidas al azar de entre las que forman esa palabra.

   f) Genera una lista de listas con todas las posibles permutaciones de la lista `[0 1 2]`.

6. **Números aleatorios**:

   a) Genera un boleto al azar de la Lotería Primitiva [38].

   b) Genera una lista de 1000 números naturales aleatorios elegidos entre 0 y 1000. Calcula la media, la varianza y la desviación estandard de la lista, así como el máximo y el mínimo valor de la lista.

[36] Matemáticamente, los caminos aleatorios son cualquier proceso aleatorio donde la posición del objeto en cierto instante depende sólo de su posición en el instante anterior y de una variable aleatoria que determina su dirección y longitud de avance en cada momento.

[37] Por ejemplo: `set pcolor pxcor`, `set pcolor pxcor + pycor`, `set pcolor pxcor - pycor`, `set pcolor pxcor * pycor`, etc.

[38] Recuerda que, en su versión simple, la Lotería Primitiva consta de seis números elegidos entre el 1 y el 49.

c) Genera una lista de 10.000 números binarios al azar (0's y 1's). Calcula la media, la varianza y la desviación estandard de la lista.

7. **Ángulos y polígonos regulares**:

   a) Calcula la lista de los ángulos acumulados al recorrer un polígono regular en función del número de lados del polígono[39].

   b) Genera la misma lista del apartado anterior, pero esta vez midiendo el ángulo en radianes[40].

   c) Genera la lista de los cosenos y senos de los ángulos de la lista del primer apartado.

8. **Números cuadrados**:

   a) Calcula la lista de los cuadrados de los primeros 1000 números naturales.

   b) ¿Es 516 un cuadrado perfecto? ¿y 2025?[41]

   c) Comprueba que la fórmula

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

   es válida para $n = 10, 100$ y 1000.

9. **Logaritmos**[42]:

   a) Genera la lista de los logaritmos en base 10 de los primeros 100 números naturales.

   b) ¿Cuál es el primer número natural cuyo logaritmo en base 10 es mayor que 0,5?

   c) Genera la lista de los logaritmos de 100 en base k, con $k = 2, \ldots 100$.

   d) Calcula la suma de los logaritmos en base 2 de los primeros 10 números naturales no nulos y comprueba que es igual al logaritmo en base 2 del producto de todos ellos.

[39] En el caso de un cuadrado la lista resultante debería ser `[0 90 180 270]`.

[40] NetLogo tiene la constante `pi` para representar el número π.

[41] Puedes hacer uso del comando `member?` para responder a esta pregunta.

[42] En matemáticas, el logaritmo de un número, en una base determinada, es el exponente al cual hay que elevar la base para obtener dicho número. Por ejemplo, el logaritmo de 1000 en base 10 es 3, porque $1000 = 10^3 = 10 \times 10 \times 10$.