Building knowledge with turtle geometry

Exploring the land of powerful scientific ideas with Logo's Turtle

Andreas R. Formiconi

Version 1.0

June 2022

Cover picture

The miniature in the cover picture comes from a "moralised bible". Moralised Bibles were publications of some paraphrased biblical texts, accompanied by moral comments and illustrated. This one dates back to 1220-1230 and is currently in the Austrian National Library. It is also famous for being used by mathematician Benoit Mandelbrot in his book "The fractal geometry of nature" (1983, p. 276). Mandelbrot observed that, around 1200, science and philosophy were languishing, but the architecture was flourishing - it was the era of Gothic cathedrals. No wonder then that God, intent on creating the world, used architect's tools. The legend of the miniature reads: "Here God creates the sky and the earth, the sun and the moon and all the other elements".

Mandelbrot in this illustration saw three geometric shapes, circles, waves and "curls", pointing out that while the circles and waves had been given enormous attention over the centuries, the curls had remained intact. Curls found a mathematical home thanks to Mandelbrot's fractals. Wonderful the consonance, I would say visionary, between the intuition of the medieval miniaturist and the mathematical scenario envisaged by Mandelbrot. The coexistence of regular and irregular geometric shapes, that is to say, certain and uncertain, which struck Mandelbrot in this miniature, is appropriate to represent the spirit of this book.

I'm reproducing this image faithfully as it is available in the Public Domain.

Fractals represent one of the invasions of the so-called exact sciences of the twentieth century by complexity, i.e. uncertainty. In section 6.4, "The impossible task of measuring the length of coastlines", I tried to expose the reasoning made by Mandelbrot in the article in which he introduced the concept of fractal dimension, which is distinctive of fractal objects.

Contents

1 Learning, emotions, complexity: not just what, but how **5**
- 1.1 Introduction — Seymour Papert's Logo language and Turtle Geometry . . . 5
- 1.2 Papert's thought and last neuroscience findings . . . 6
  - 1.2.1 The four pillars of learning . . . 7
  - 1.2.2 Emotions . . . 9
- 1.3 Fighting the "School of Mourning" . . . 11

2 Software languages **17**
- 2.1 The Logo culture . . . 17
- 2.2 The ABZ approach . . . 22
- 2.3 A free software environment: LibreLogo . . . 24
  - 2.3.1 Why free software . . . 24
  - 2.3.2 LibreLogo . . . 25
- 2.4 Differences among the languages . . . 26

3 Do not "teach": be maieutic! **29**
- 3.1 Before talking to the turtle... . . . 29
  - 3.1.1 First steps . . . 31
  - 3.1.2 Towards first geometrical shapes . . . 33
  - 3.1.3 Making things easier... . . . 37
- 3.2 Making things smaller and larger . . . 44
  - 3.2.1 The Turtle is able to learn symbolic names . . . 44
  - 3.2.2 The Turtle is even smarter: passing parameters to new commands . . . 45
- 3.3 Doing round things . . . 48
  - 3.3.1 Syntonic learning . . . 48
  - 3.3.2 Spira mirabilis . . . 53
  - 3.3.3 Golden spiral . . . 56
  - 3.3.4 POLY . . . 58
  - 3.3.5 Successive approximations . . . 66

4 Simulation: physics dynamic **71**
- 4.1 Free-falling body . . . 71
- 4.2 Free-falling body with constant acceleration and horizontal velocity component . . . 75
- 4.3 Free-falling body with air resistance . . . 76
- 4.4 Body hanging from spring . . . 77

4.5 Pendulum 78
4.6 Body free-falling from space 80
4.7 Calculating the orbit of Halley's comet... 80

5 Simulation: behaviour **89**
5.1 Is there a place for randomness in computers? The Turtle plays the turtle... 89
5.2 Shaping the environment 92
5.3 Modeling smell 96
5.4 Modeling sight 103
5.4.1 Facing light 104
5.4.2 Two-eye vision 106
5.4.3 Two-eye vision with intensity perception 108
5.5 Modeling interactions 112
5.5.1 Multitasking 112
5.5.2 Interacting turtles 115

6 Simulation: fractals growth **121**
6.1 Recursion 121
6.2 Fractals 123
6.2.1 L-systems: the turtle language for growing plants 129
6.2.2 The Cantor powder 142
6.3 Fractals and randomness 144
6.4 Mathematical and natural fractals: the impossible task of measuring the length of coastlines 146

7 Appendices **165**
7.1 Appendix 1: Index of powerful ideas and relevant reflections . . . 165
7.2 Appendix 2: How to manage graphics in Writer 167
7.3 Appendix 3: Programs listings 169
7.3.1 Modeling smell 169
7.3.2 Facing light 172
7.3.3 Two-eye vision 174
7.3.4 Two-eyes vision with intensity perception 176
7.3.5 Managing two turtles simultaneously in LibreLogo 179
7.3.6 Approximating the circle perimeter 181
7.3.7 Fractal L-system stick tree 181
7.3.8 The impossible task of measuring the length of coastlines 182

Chapter 1

Learning, emotions, complexity: not just what, but how

1.1 Introduction — Seymour Papert's Logo language and Turtle Geometry

These notes are about the use of turtle geometry at the primary and secondary school. The text is largely inspired to the work and vision of Seymour Papert, a mathematician and computer scientist who also got a PhD with Jean Piaget. Nowadays, Papert is known mainly for being the inventor of the educational programming language Logo. The Logo programming language was aimed at fostering a better way to teach sciences and to induce self-reflection: learning while reflecting on its own learning. We live in times of very rapid change, too rapid. Papert's profound thought has been largely forgotten today. Many of current "coding" practices seem to have much to do with a sort of instructive game, but the connection between such practices and sound scientific concepts seems to be largely absent. This text represents an effort to fill this gap. In particular, by showing a series of exercises, spanning from the primary school level to the last years of secondary school, we try to recover and discuss Papert's concepts of *syntonic learning* and *powerful ideas* [Papert, 1993]. Papert was used to say that the *powerful idea* concept is so important that the most powerful idea is that of *powerful idea* itself! Throughout the article the reader will find gray text boxes intended to comment on powerful ideas that are evoked by the exercises. In appendix 7.1 (pag. 165) an index of all gray boxes is provided. But with this book I'm trying to fill also another gap. At the time of the development of Logo, Papert was not alone. The project was backed up by a consistent group of outstanding scientists. Among them, Harold Abelson and Andrea diSessa developed a wide exploration of amazing studies which can be done with Logo, kind of dizzying journey, which, among other things, from the study of polygonal shapes leads to the exploration of feedback, to the simulation of growth dynamics, ending with a simulation of general relativity! The textbook they wrote, Turtle Geometry [Abelson and diSessa, 1980], is very

rich in ideas for an educator with a reasonable mathematical background, but too difficult to be used at large in school contexts — the appearance of theorems and lemmas right in the first chapter might be daunting for many readers. Thus, in this work, I would like to bring some of their studies closer to the school world, by revisiting them in a more affordable way and providing ready-to-use code to start with, encouraging further hands on explorations. However, I also added other topics so as to cover a wide range of age levels.

Interestingly, having proposed this matter to several hundreds of teacher students, I realized how teaching Logo to kids is very similar to teaching it to grown up students. And, in average, during the first approach kids perform even better! We often discuss this point with students that claim having had major difficulties "talking to the turtle"[1]. The majority of them experience a kind of "resurrection" after some initial struggling: "At the beginning I didn't understand anything but then... how cool!" But a few of them never get rid of the bad mood, at least within the two and a half weeks span of the lab. Talking with these students is always very interesting because they are amazed when I tell them that, most of the times, nine year old kids get along with the turtle quite well. When trying to dig into this paradox, it appears that kids grasp quite naturally the playful, even if thoughtful, aspect of the situation. On the other side, adults easily get trapped in their own prejudices: I don't have a head for numbers, computer and me are very far away, I never went along with technologies... Especially the last one is a statement I heard lots of times, and, amazingly, from the older digital natives. That is, people having thousands of Facebook contacts, candidly claim of not getting along with technologies!

Having stressed that, please, do face the turtle with childish curiosity, starting from scratch. Since I suppose you know how kids tackle something new, try doing just the same. Let's be clear though: playful doesn't mean easy. Learning is very rewarding, but it takes hard work.

1.2 Papert's thought and last neuroscience findings

In cognitive science, learning is said to consist of forming an internal model of the outside world. The latest scientific findings show that the process of knowledge construction is also a biophysical construction process: new dendrites grow, the number and strength of synapses change rapidly. In other words, the *connectome* — the "wiring diagram of the brain" — is constantly evolving. One might be tempted to think of a sort of magic mirror of reality, but it would be a misleading metaphor. The brain constantly monitors all external and internal sensory stimuli, constantly negotiating the content received with the internal representation of the world. This is an endless learning process, but it is not merely a linear process of accumulation of new information. Making sense of the world is a matter of recovering the hidden causes of events in order to reconstruct some general theory that can account for them. It seems that our brain is extremely good at this and at a very early age. Scientific studies have been done that show how 8-month-old babies behave like budding scientists who think like statisticians [Xu and Vashti, 2008].

[1] The turtle will be the protagonist of our activities. We'll clarify it in the following.

This scientist attitude should be exploited better at school. Several authors had the intuition that exploration should be an important ingredient in school activities — Dewey, Montessori, Freinet, Papert, just to mention few of them. The feeling that we should exploit the natural budding scientist attitude of the young brain much better is strong. The point is particularly striking when dealing with scientific subjects: the fact that STEM disciplines are taught, more often than not, paying very little attention to the basic processes of scientific knowledge creation is an amazing oxymoron. Teaching science subjects should be a fantastic opportunity to put such a natural talent of the brain to work. Unfortunately, this is not the case. The STEM disciplines are usually taught like everything else, according to a model of passive transmission of knowledge.

This was exactly the idea of Seymour Papert. His intention was to provide contexts — "microworlds" he called them — which students could explore, posing themselves goals and, eventually, making discoveries.

Papert is often associated with "pure discovering learning methods", that is, learning without a teacher's mediation. But that is a misinterpretation of his thinking. Rather, he meant a kind of guided discovery:

> Of course, the teacher plays a role, the teacher guides the discovery and reinforces it, by talking about afterwards. [Papert, 1986, video at 6'20"]

It is one thing to teach while taking advantage of children's propensity to explore, it is another to leave them to themselves. Simply exposing students to any educational artefact or context does not create any magic.

1.2.1 The four pillars of learning

The considerations developed by Papert, on the other hand, are particularly interesting in the light of current neuroscientific knowledge. For example, it is easy to find in Papert's words, the first three of what Stanislas Dehaene calls the four pillars of learning: *attention*, *active engagement*, *error feedback* and *consolidation*. To begin with, however, it is important to stress that only by listening to the students first, can one hope to activate their attention. Something that Dehaene himself tells very effectively [Dehaene, 2020, pag. xxii]:

> I am deeply convinced that one cannot properly teach without possessing, implicitly or explicitly, a mental model of what is going on in the minds of the learners. What sort of intuitions do they start with? What steps do they have to take in order to move forward? What factors can help them develop their skills?

In part, it is the experience that makes it possible to answer these questions. However, when teaching a new course, the experience may be lacking. Moreover, it is also important to recognize individual problems, which can be caused by shortcomings, but also by specific talents. In order to get feedback from students it is necessary to activate their attention and stimulate their active involvement. This is the only way to achieve listening. The right mood can be created by immediately starting to offer the first small challenges, with something fun if possible. The idea is to encourage exploration, but as mentioned earlier, this does not mean that teachers should be absent, leaving students to the mere

display of any educational artifact or context. On the contrary, teachers must be more engaged because they must encourage initiative and personal exploration, but without failing to guide students towards meaningful objectives.:

> The Logo teacher will answer questions, provide help if asked, and sometimes sit down next to a student and say: "Let me show you something." [Papert, 1993, pag. 83]

And coming to the third pillar of learning — error feedback — Dehaene's idea that "*error feedback* is not synonimous with punishment" [Dehaene, 2020, pag. 208] resonates with Papert's perspective, as well:

> In the Logo environment, children learn that the teacher too is a learner, and that everyone learns from mistakes. [Papert, 1993, pag. 52]

And when talking about a school programming activity:

> ...bugs, are not seen as mistakes to be avoided like the plague, but as an intrinsic part of the learning process. [Papert, 1993, pag. 71]

As far as *consolidation*, the fourth pillar, is considered, the Logo feature that Papert described by the expression "low floor and high ceiling" lends itself very well to the realization of Bruner's spiral curriculum [Bruner, 1960]. In this book the reader will find several cues to resume specific topics at later stages of the training course, with increasing degrees of depth. The case of the circle and other circular forms is the one described in greater detail and with wider temporal extension, starting from the exercises to draw circles with one's own body, at the elementary school level, arriving at the calculation of the orbits of celestial bodies, at the end of high school or the beginning of tertiary education.

The guiding idea, therefore, is to provide examples that can be used according to the concept of Bruner's spiral curriculum, intended as a form of long-term consolidation, enriched by the deepening of the knowledge acquired in previous years. Obviously, the consolidation intended by Dehaene also involves short and medium term strategies and the resumption of topics in subsequent years is only the last stage of the entire consolidation process that curricula should take on.

Deahene's four pillars are all essential for learning, but it all starts with the first one, *attention*: capturing students' attention is the main challenge a teacher has to face, failure in this task means defeating all subsequent efforts. And it is not trivial at all to trigger and, above all, to maintain the attention of the students.

Usually, it is given for granted that students, of any age, are paying attention to the lessons, just because they are at school. Of course, they may appear to be listening, but this does not imply true and fruitful attention. Conscious attention is a precious resource which is not so easy to activate and keep for a long time. In this case, listening means being aware of the actual state of attention of its students and being able to evaluate how long attention can be truly sustained. Everybody knows how powerful video games are in stimulating attention. Video games work, as players are very motivated. The art of teaching involve the creation of contexts which triggers such a motivation as much as possible.

Attention is very much related to filtering: our brains see what they expect to see. The listening attitude of teachers means also trying to remember what it means to be ignorant: we all tend to think that what we see everyone can see. The famous "invisible gorilla" experiment and other similar cases, such as unseen large objects in front of a landing plane simulated, remind us that inattention to the unexpected may be incredibly strong. If students are distracted, the teacher's words will be completely lost. Attention is also related to the myth of multitasking. The idea that we can perform two (or even more) tasks simultaneously is illusory, unless one of two tasks was learned so well to be executed in a highly automatized fashion. If both tasks require conscious attention, one of them would necessarily slow down. Tasks different from the most important one will act as *distractions*. It has been proved, for instance, that excessively decorated classrooms distract students and reduces their concentration [Fisher et al., 2014].

It is also because of these considerations about attention that we have preferred text-based instead of blocks based programming environments for the scientific explorations described in this book: the blank page as a free space for maximising concentration and unleashing creativity. Too many distractions instead, dilute concentration and dissipate creativity.

1.2.2 Emotions

Nowadays neuroscience has widely documented the role of emotions in learning:

> Negative emotions crush our brain's learning potential, whereas providing the brain with a fear-free environment may reopen the gates of neuronal plasticity [Dehaene, 2020].

The crucial role of emotions in our rational thinking is being deeply studied for more than a couple of decades. Damasio, LeDoux and Dehanene are among the most prominent neuroscientists, which have studied the influence of emotions on human thinking from different perspectives. There are evolutionary reasons that account for the role of emotions in learning. The chances of survival of an individual, of any animal species, are heavily influenced by the ability to remember. But not all memories are equivalent. The close relationship with the chances of survival determines a scale of priorities, which we could classify as urgent, highly convenient, marginal or irrelevant. Nature does not waste, because the resources of each individual are limited. The greatest urgency occurs when life itself is at stake. The shape or smell of a predator must be reacted to as quickly as possible. In the face of such dangers, any other brain activity takes second place: no matter how promising a courtship or the smell of delicious food may be, the appearance of a predator will blow any plan away. Similar priorities condition the persistence of memory associated with the various situations, since circumstances relating to serious emergency threats require the most persistent, long-lasting memory.

A long evolutionary process has shaped the brain in its fundamental functions. The persistence of memories is closely tied to emotions. Memories that have been acquired in conditions of stress or fear always elicit the same negative sensations, each time they are called back to consciousness. Such associations are extremely strong and difficult to eradicate. It is the amygdala, which is

located deep in our "reptilian brain", the structure in charge of sending alert signals to the other parts of the brain. In particular, those signals are sent to the nearby hippocampus, which stores the most relevant episodes of our existence. This connection between the hippocampus and the amygdala hardwire the most negative experiences. However, a math problem is not as dangerous as a saber-toothed tiger! Unfortunately, the mechanism activated is the same in both cases, once the first encounters with formal mathematics have been stored together with traumatic emotions.

> The effects of school-induced stress have been particularly studied in the field of mathematics, the school subject most famous for the all-too-well-known anxiety it induces in so many students. In math class, some children suffer from a genuine form of math-induced depression because they know that, whatever they do, they will be punished with failure. Mathematics anxiety is a well-recognized, measured, and quantified syndrome. Children who suffer from it show activation in the pain and fear circuits, including the amygdala, which is located deep in the brain and is involved in negative emotions. These students are not necessarily less intelligent than others, but the emotional tsunami that they experience destroys their abilities for calculation, short-term memory, and especially learning [Dehaene, 2020, pag. 212].

Similarly, good experiences have a positive impact on learning, albeit with a lower priority than negative memories. Many studies have shown that making children laugh improves certain cognitive abilities such as attention, motivation, perception and/or memory, which in turn improve learning [Esseily et al., 2016].

Positive feedbacks are good not only for children but also for older students. This is one of the letters received from primary school teacher students.

> Twenty-three years, eighteen of which spent studying. Last year of university. Only one goal: graduating. As soon as possible. Getting rid of lectures full of slides, read in a semi-deserted dark classroom. Escaping the layout of my head, set to study pages and pages, for five years now. Getting rid of cold relationships, bureaucracy.
>
> "You're not kids anymore," they tell us. Is that the plan? An adult's life, pretending to know everything, but unable to really do anything? Lost in everything?
>
> Then another lab: "Laboratory of teaching technologies", says my study plan.
>
> I'm floundering!
>
> Then the first emails from the teacher. He says he doesn't want people to sleep in class (a famous phrase in school literature...). — He talks about a laboratory that we can do at home too, by Internet. A lab about thinking and thinking about thinking...
>
> What a funny fella...
>
> So, intrigued and suspicious, to lesson one. The professor arranging things on the desk, like electronics, toys, apparently. I've never seen anything like it but three juggling balls[2] and a laptop.

[2]Papert spent a dozen pages of Mindstorm [Papert, 1993, pag. 105] digging through the

> What's going on here? Course program? Books list? Slides?
>
> None of that. Just a question: "Do you know what's this?" No, I do not know. Because out of three hundred books I have studied, there is no paragraph dedicated to this turtle! A robot, but an imaginary mathematical turtle as well. Interesting, I want to try it... Like a schoolgirl....
>
> As soon as I'm back, I'll try...
>
> And I'm caught right away. I'm in love with that turtle. I need to understand, I need to figure out how to move this turtle, how to change direction, how to color the trace. I have to do this. There's the textbook we got from the professor. I'm not starting to go through the first page. Instead, I go there to seek answers to my questions, when I'm stuck. And what I find there are just hints, then it's up to me. It's not easy, but slowly I'm better off. I feel like a kid...
>
> And I discover a world, a world full of clues and insights: my mind travels at the speed of light, creating a thousand connections. No categories, no mental predisposition. My mind builds bridges and breaks them down, creates synapses, ideas that shine in my head. I feel my brain digress, but positively: I learn from practice, not only from theory. I let my thoughts go like a river, like a shower of meteorites. I'm thinking like a kid...
>
> But I'm not alone. The forum is full of life. Ask questions, seek answers, explore and try to assist classmates. I don't always succeed, but at least I try. I try to help them with my experience, and they help me with theirs. It's good to struggle together. Like kids...."

1.3 Fighting the "School of Mourning"

At the beginning of his Method, Edgar Morin talks about the "School of Mourning" [Morin, 1977, Pag. 8]:

> The school of research is a school of Mourning. Every neophyte who dedicates himself to research is induced to the fundamental renunciation of knowledge. They are convinced that the age of the Pico della Mirandola belonged to a past of three centuries ago, that it is now impossible to build a vision of man and the world together. It is shown to them that the growth of information and the increasing heterogeneity of knowledge exceeds any capacity for storage and processing of the human brain. They are assured not to be complained about it, but to be happy about it. They must therefore devote all their intelligence to increasing *that determined knowledge*. They are included in a specialised team and in this expression the term underlined is specialised, not team. Once specialists, researchers are offered the exclusive possession of a fragment of the puzzle, but the global picture eludes everyone.

---

debugging concept by teaching cascade juggling. That's why we needed the juggling balls in the lab...

In "Seven Complex Lessons in Education for the Future" Morin points out the *principle of pertinent knowledge*. This means grasping general, fundamental problems and inserting partial, circumscribed knowledge within them. The predominance of fragmented learning, divided into disciplines, often makes us unable to connect the parts to the whole; instead we need a way of learning that allows us to frame the subjects in their context. Omitting connections and intersections between disciplines deprives them and prevents one from grasping the emergence of the new.

Another key aspect of Morin's thinking is the role of *uncertainty* implicit in complexity. Little or nothing is taught in school about the most disruptive and pervasive outcome of twentieth-century scientific exploration: the intimate interweaving of certainty and uncertainty that permeates all areas of science. The most spectacular presence of uncertainty on the scientific scene is probably that of quantum mechanics, whose laws cannot be 'understood' in the sense we understand in ordinary life. We understand them in that we accept them, and for a very simple reason: because they are working, and as long as they are working. But even scientists like Einstein never really got used to it. The mind tries to remove the inconsistencies between its own vision and the world: "After all, I do not live in the microscopic world, physicists will deal with it!" Not at all, uncertainty has triumphantly entered other very different areas, even much closer to our direct experience. Circumstances in which one cannot help but face the unpredictable, which we have called chaos.

Chaos occurs in a wide variety of contexts, from the simplest, such as the double pendulum, to the movement of celestial bodies[3]. Or, in more complex milieux, such as weather events — the famous *butterfly effect*[4] — population dynamics[5] and the myriad omnipresent fractal forms of nature [Peitgen et al., 1992,Mandelbrot, 1967]. Indeed, today we know that chaos is more like the rule in nature, whereas order is more like the exception.

The uncertain has contaminated even fields that seemed absolutely immune, such as that of mathematical logic. As in the case of Kurt Gödel's theorems about limits of provability in formal axiomatic theories; or that of theoretical computer science, with Alan Turing's demonstration of the impossibility of solving the so-called 'halting problem', i.e. the impossibility of writing an algorithm to determine whether a program will stop or continue indefinitely - a problem with substantial practical implications.

At the turn of the 19th and 20th centuries, Jacques Hadamard (1865-1963) defined the category of ill-posed inverse problems. Something unimaginable a few years earlier. Inverse problems require to trace the causal factors that gave rise to a series of observations. Ill-posed problems are those where very

[3]there are moons, like Saturn's Hiperion, or Pluto's Nix and Hydra, whose rotational axes have a chaotic orientation in time, this means that it is essentially impossible to predict how these moons will spin in the future: a hypothetical inhabitant could never know exactly when and where the sun will rise next.

[4]In 1972 Edward Lorenz, a metereology professor at MIT, presented a paper entitled: "Predictability: Does the Flap of a Butterfly's Wings in Brazil Set Off a Tornado in Texas?" [Lorenz, 1972]. While performing numerical simulations of meteorological dynamics, Lorenz found that in such complex systems, even tiny errors of initial conditions diverge wildly making prediction impossible. His findings definitively changed the course of science.

[5]The logistic equation is a famous example of how chaotic behavior can emerge solving even very simple models. In the logistic model the population increases at a rate proportional to the current population until the population size is small and decreases if close to the maximum possible value allowed by the context.

little perturbations of data cause huge variations in the solutions. Tomographic medical images, such as CT or PET, are the solution of inverse problems which are ill-posed. The PET problem is more ill-posed than CT's one, for instance — the consequence is that PET images are worse because the ill-posedness and the quality of data do not allow to get more spatial resolution.

Many of the above examples may be difficult to propose in school. But there are options. In chapter 6 (Section 6.4) we deal with the problem of measuring coastline length, which is tricky because of the fractal nature of coasts. It is a multidisciplinary exploration that secondary school students could face, with the help of their teachers. The startling difficulty of such an apparently insignificant task could be a good way to set sail into the ocean of uncertainty.

For the same reason that Morin criticizes, many scientists may consider his considerations as those of a philosopher. That is, something relevant to "another domain", since the fragmentation of knowledge is generally considered inevitable. There are exceptions, however. A notable voice is that of Hromkovič, who is leading an outstanding work about the teaching of computer science at school. Recently, in a paper about the role of computer science in school curricula, Hromkovič and Lacher wrote [Hromkovič J., 2017] :

> Do not teach computer science as an isolated subject, but teach computer science as a part of science and technology offering a deep contextual view. Take care on contributing to knowledge transfer to other disciplines. Build and use bridges to the development of languages and mathematics.

This thought fits neatly into Morin's general vision. At the same time, they provide an excellent framework for the ideas and the activities I'm proposing in this book. This is not a book about computer science, strictly speaking. It is a book about science that can be written because computer science exists. It is a book for looking at some specific aspects of science, using a computer and an adequate programming languages, through personal engagement, learning by doing and discovery. In the following we are going to describe briefly the contents of each chapter.

In the next chapter (2) we discuss the software language and the development environments the reader is supposed to use while reading the book. It is not just an instrumental chapter, but an opportunity to do some important remarks about languages and development environments. They play an important role in this book. Of course, it can just be read, in order to get a general idea. However, in order to be able to propose some of these activities to the students, you have to engage in these activities yourself. So, this is the first point: to make full use of these resources, you have to read and try by yourself, hands on. The second point is related to the kind of instruments I'm proposing, that is free software or tools made available freely by academic groups. The access to education is a key pillar of modern democracies. All western constitutions state the importance to let all citizens access the educational resources to achieve an adequate cultural level. A key point, especially as far the lower grades of instructions are concerned. Not in every country, not in every region and not in all families the accessibility to expensive educational tools can be given for granted. The use of free or low cost tools plays a crucial role in such a broad perspective. Not only, the use of the most widespread open standards allows for a long lasting learning investment.

The beginning of Chapter 3 is about the first steps, and how to guide kids along them, starting with physical activities. This part is inspired to Papert's concept of *syntonic learning*, with which the teacher exploits the link between phisical and intellectual activities. Then it is shown how a fairly complex idea may hide even behind a simple physical activity. Through the rest of the chapter variations about the circle are explored, namely spirals, polygonal approximations, orbits. This chapter can also be read from the perspective of Bruner's spiral curricula, the circle being revisited in increasingly complex ways, starting with children's play.

The following chapters fall all within the world of simulations. In all the examples the computer is used as a tool to explore scientific concepts, hands on. Simulations by themselves are an important ingredient in scientific education, since they are a crucial tool of contemporary science. Nowadays, numerical approaches play a crucial role in a wide range of problems in all fields of science. The speed of computers and their memory allows to simulate phenomena which would be far too complex to tackle in other ways. In biology there is even a specific expression to describe experiments conducted by means of computer simulations: *in silico* instead of in *in vitro* experiments.

In chapter 4 we explore the computational way of studying and solving classical dymanics problems. This writing was inspired largely by Sherin's work on the comparison of programming languages and algebraic notation as expressive languages for physics [Sherin, 2001]. The codes have been written in the simplest possible way so as not to frighten readers and encourage them to push themself into further explorations. The examples are variations on the problem of the free fall of a body, where various additional forces are considered, such as air resistance, the elastic force of a spring, the constraint of a pendulum, or different initial conditions, such as non-zero horizontal initial velocity or a large distance from earth surface, or a combination of both to calculate the orbits of celestial bodies around the sun.

While chapter 4 sticks to the classical deterministic description of phenomena, in the following two we introduce *uncertainty* in terms of Morin's big picture. In Chapter 5, we introduce randomness for simulating animal behaviour. The behaviour models used in the exercises are pretty naive but perfectly adequate to highlight basic features of living systems, such as the power of feedbacks and the deep entanglement between randomness and deterministic relationships in complex systems. The chapter starts by introducing random choices in the turtle's behaviour. Then three kinds of models are discussed: a simple smelling model, some variations on a two-eyes vision model and a multiple-turtles interaction model for simulating phenomena such as the spread of a disease in a population. The multiple turtles study gives also the opportunity to delve a little into the concept of multitasking by means of a simple exercise.

Uncertainty plays a different role in chapter 6, apart some cases where also randomness has been added. We enter this new context through the concept of recursion to explore a variety of fractal shapes. Different fractals may serve to highlight various aspects. The basic "sticky tree" opens the door to a reflection on infinity and infinitesimal. The Koch fractal allows to explore the impossible task of measuring a coastline. This is achieved by expanding in a reasonably simple way the seminal paper of Benoit Mandelbrot about the concept of *fractional dimension* [Mandelbrot, 1967], the distinguishing geometrical feature of fractal shapes. The blurring of integer dimensions which characterizes fractals

is the sort of uncertainty we explore in this chapter. The last examples show how amazing the fractals reproduction of plant shapes may be.

The examples proposed in chapters 5 and 6 show how an adequate mixing of simple random choices and few trivial deterministic rules may give rise to realistic natural manifestations, both dynamic and geometric. The whole is intended to give a tangible idea of Morin's saying that human condition is about *navigating in an ocean of uncertainty through archipelagos of certainty* [Morin, 1999].

Chapter 2

Software languages

2.1 The Logo culture

Where computer programming is concerned, programming languages should be distinguished from development environments. That seems self-evident, but a lot of people confuse the two. In this book we used one language, Logo, but three development environments: LibreLogo, XLogoOnline and WebTigerJython[1] . The criteria for choosing one system or another are: scientific and cultural context, availability of a reasonably shared standard and presumed longevity.

In this chapter we are going to clarify the choices we made about languages and environments.

Logo started in the 1960s, when computer scientists Cynthia Solomon, Wally Feurzeig, and Seymour Papert joined forces in a project to make mathematics and computer science more accessible to children. The project started in 1966 at 'Bolt, Beranek and Newman Inc.', but in 1969 the three scientists gathered at the MIT Artificial Intelligence Lab where they formed the 'Logo Group'. Seymour Papert was the leader of the project, because of its peculiar background. He was born in South Africa in 1928, began studying mathematics at Johannesburg and then at Cambridge. He then obtained a PhD in psychology, working from 1958 to 1964 with Jean Piaget at the University of Geneva.

From the beginning, Logo was related to the turtle, a kind of robot able to move and, possibly, to draw lines on the ground according to programmed commands received from a computer.

Given the shape of the robot and its slow motion, the association with a turtle seemed natural. However, Papert also referred to an earlier turtle, that designed by Williams Greg Walter (1910-1977), a prominent neurophysiologist and cybernetic scientist. In those days, around 1950, the robots built by Greg Walter were known as cybernetic machines. Cybernetics was a multidisciplinary scientific thought movement promoted by a group of eminent scientists such as Norbert Wiener, John von Neumann, Claude Shannon, Alan Turing and many others, including William Greg Walter. The fundamental idea behind cybernetics was the *feedback loop*[2]. This has been recognized as the basic mechanism

[1] WebTigerjython is actually an IDE for programming in Python. However, in this book, we mainly use the instructions from the turtle library, so we basically use it as a form of Logo.

[2] At page 99 we will see an example of feedback.

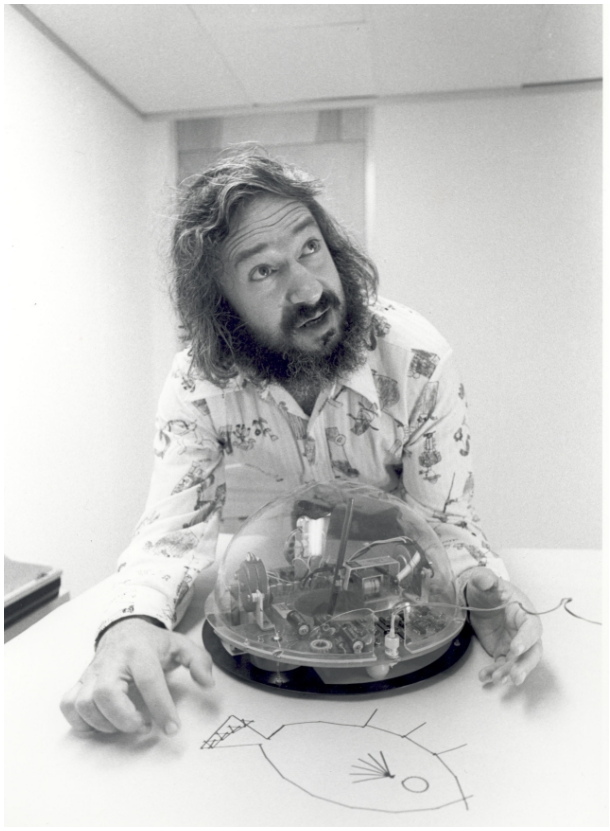


Figure 2.1: Seymour Papert shows one of the early versions of Logo, when it was some sort of robot for drawing.

for self-regulation in machines as well as in living systems. We will explore feedbacks in several contexts, such as in the simulation of the smelling sense, at page 99. Walter's turtle was an electro-mechanical device which "imitates life" [Walter, 1950]:

> An imitation of life concerning the author's instructive genus of mechanical tortoises. Although they possess only two sensory organs and two electronic nerve cells, they exhibit "free will".

The study of feedback mechanisms in these simple contexts enabled an exploration of their crucial role in living systems. The turtle we simulated on page 108, with two-eyed vision and the perception of light intensity, is a reproduction of Walter's turtle, basically.

The concept of a turtle, or any other hypothetical creature, constrained to move in a given space, two- or n-dimensional, and able to create figures while moving, gives rise to the notion of *turtle geometry*. In fact, turtle geometry can be thought of as a geometry, like euclidens or cartesians. The use of one geometry rather than another is a question of convenience in the specific case. They are all "true" but focus on different entities. The circle, for example, in Euclid geometry is defined by the notion of point and line [Byrne, 1847, pag. 22]:

> A circle is a plane figure bounded by one continued line, called its circumference or periphery; and having a certain point within it, from which all straight lines drawn to its circumference are equal.

In Cartesian geometry the circle is given by an equation, i.e. a circle is a plane figure bounded by a line which is composed by all points of coordinates (x, y) such as

$$x^2 + y^2 = r^2 \tag{2.1}$$

where r is the radius.

And in turtle geometry? In this case, the circle is a block of code:

```
REPEAT
     FORWARD 1
     RIGHT 1
```

Listing 2.1: LibreLogo

Puzzling? Mathematicians are by no means racist: they do not care how strange a creature seems, as long as it obeys the given rules of the environment to which it claims to belong. The previous code *is* a circle, in turtle geometry. Harold Abelson and Andrea diSessa wrote their "Turtle Geometry" using blocks of code as mathematical objects. For instance, their derivation of the Total Trip Theorem[3] [Abelson and diSessa, 1980, pag. 24] concerns the properties of a generalization of the previous code, the so-called POLY procedure:

[3]The Total Trip Theorem, or Closed-Path Theorem says that the total turning along any closed path is an integer multiple of 360°. We used this theorem at page 64.

```
TO POLY
     REPEAT
          FORWARD SIDE
          RIGHT ANGLE
```

Listing 2.2: LibreLogo

The language used by Abelson and diSessa in "Turtle Geometry" is Logo, but since it is used as a language for creating mathematical objects, it is a sort of proto-Logo, where syntactical elements are reduced to a minimum, in favor of readability. For example, instruction blocks are identified by indentation — as in today's Python — the number of cycles is absent and variables are assumed to be globally accessible.

Turtle geometry is not confined to the world of education, but is used even in other contexts, when it is appropriate to create geometric shapes by "drawing" them. For example, Prusinkiewicz et al used a Logo-type turtle to generate 3D motions according to an L-system formalism to model herbaceous plant development [Prusinkiewicz et al., 1988][4].

But beyond this, in the 1970s and 1980s, expressions such as "Logo movement", "Logo culture", "Logo as a philosophy of education" or even "Logo as a laboratory for lifelong learning about learning" were commonplace:

> Logo is a language for learning. That sentence, one of the slogans of the Logo movement, contains a subtle pun. The obvious meaning is thatLogo is a language for learning programming; it is designed to make computer programming as easy as possible to understand. But Logo is also a language for learning in general. To put it somewhat grandly, Logo is a language for learning how to think. Its history is rooted strongly in computer-science research, especially in artificial intelligence. But it is also rooted in Jean Piaget's research into how children develop thinking skills. [Harvey, 1982, pag. 163]

Everyone agrees that Seymour Papert was the guiding influence in developing such a computer culture. However, it is important to realize that, as such, this culture was intimately shared by a rather large group of scholars involved in education or research about artificial intelligence. See, for instance, the perspective shared by Cynthia Solomon:

> As computers continue to enter schools and homes, parents, teachers, and children face the problem of integrating the machines into their lives. For many, computers serve as powerful instruments for personal use and intellectual development. Many Logo researchers see the potential of computers to serve as personal instruments for everyone and have been working toward that goal. In the process, they have focused on developing not only the Logo language, but things to do with the language and ways of thinking and talking about these activities. How people talk about what they are doing, the way they interact with one another, and the way they interact with the computer give rise to a new kind of culture, a computer culture. [Solomon, 1982, pag. 195]

[4] In chapter 6 we shall see examples of this kind.

Or the "Cultural Glossary by E. Paul Goldenberg [Goldenberg, 1982, pag. 210]:

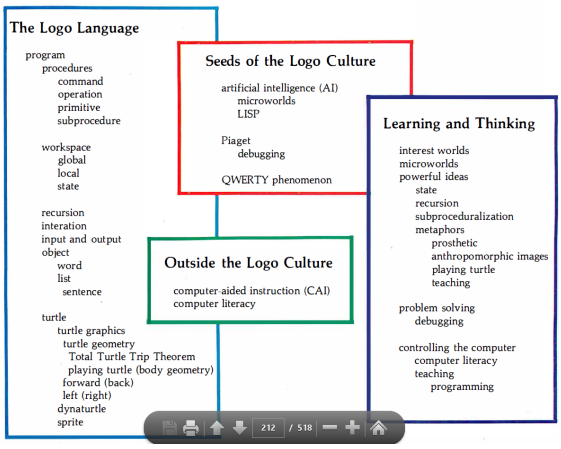


Figure 2.2: Concepts at stake around Logo in 1982.

The relevant fact is that all these people, and many others, although from a quite different background, shared a *broad vision*. A vision that led them, starting from the Lisp artificial intelligence language, to create Logo, as "a LISP-like language, and a laboratory for loose, lifelong learning about learning" [Harvey, 1982, pag. 163].

But has Logo's culture permeated the educational system? Not quite, if we're talking about the generalized use of Logo in schools. On the other hand, one cannot speak of failure, since 309 versions of Logo have been counted up to now [LogoTree, 2020], and of all these versions, 75 are still active.

Even programming systems based on visual blocks may be seen as a legacy of Logo. Scratch [Resnick et al., 2009], the best known, was developed at MIT Media Lab under the direction of Mitchel Resnick, who obtained his doctorate with Seymour Papert and Hal Abelson. However, this branch of education languages came not only from the academy, as was the case with Logo. It also drew on the social and customary changes brought about by mass access to the Internet. The success of Scratch came with the emergence of several similar systems, including Blockly [Fraser, 2015] and Snap! [Garcia et al., 2015].

These systems, in various shades, are very powerful and enable the development of multimedia projects. However the emphasis shifted from expressions like"Logo culture", "Logo as a philosophy of education" or "Logo as a laboratory for lifelong learning about learning" to slogans such as "designing, creating, remixing" or "Imagine, Program, Share". In a nutshell more social less cultural. Indeed, there are studies comparing text-based languages and block-based languages within education. The results show that there is a small benefit of block languages for learning basic software constructions in the short term, but not so much in the long term. On the other hand, some of these studies did not show any significant differences in the in-depth understanding of what an algorithm does [Weintrop and Wilensky, 2015]. In addition, students demonstrated higher levels of self-esteem with Logo (logo) [Lewis, 2015] and greater confidence and pleasure in programming [Weintrop and Wilensky, 2019]. A fairly obvious result

of these studies is that, as students become more skilled, they take more advantage of text programming. Consequently, new mixed systems have emerged that make it easy to switch from text to block language versions and vice versa, such as Pencil [Bau et al., 2017].

However, this whole line of research is mainly focused on programming skills in the narrow sense, programming in Java or Javascript being the ultimate goal. In fact, in block language culture, the focus is on programming, which is widely referred to as "coding". A very popular word these days, on the Internet as well as in focus groups on technology and school. The hype is considerable, but the widespread impact in the school world is far from general and homogeneous. And, usually, it is a matter of adding "coding hours" to already saturated curricula. The widespread "to code or not to code" controversy is pointless because it does not address the fundamental issue, which is essentially cultural. What we're concerned about here is the impact of computer science across all disciplines. The significant goal is to help the students to explore various fields of science in the new ways made possible by computer science, leveraging kids' natural budding scientist attitude. The issue is finally cultural because it concerns the "teaching in the box" tendency, which prevails in all disciplines, even in teacher training. In such a wide sense, the problem of using computers to promote scientific thought remains unchanged over time: in the era of the Logo movement as well as today.The problem entails teachers' training: in-the-box educated teachers who teach in a box-free world. The Logo movement had a sound cultural goal but the school world was not ready to take advantage of it. The visual programming hype is a successful Internet fact but the school world is disoriented, missing the meaningful point.

There is another problem. I began this chapter writing that programming languages need to be distinguished from development environments. With visual programming systems this is no more true. Each system is offered through its own web interface and each one has its own features, its own export format. For instance, one cannot port a program from a system to another by means of some simple text editing, as in the case of Logo dialects. Thus, talking about a visual language means talking about a specific environemnt as well. How long a web environment may last? Will it be possible to recover and recycle all the scripts accumulated in the system if one day the system is shut down? Abelson and diSessa wrote 'Turtle Geometry' some forty years ago. Today, the wealth of logo examples in the book can still be easily copied, even by speech typing and a few touch-ups. It would hardly be possible to do the same thing with a collection of scripts made of block code, unless there is a system for exporting them to a standard text format, such as Logo.

2.2 The ABZ approach

The Training and Advisory Centre for Computer Science Education ABZ[5] does an outstanding work. In short, ABZ is one of only a few to have inherited the Logo culture developed at MIT in the 1970s.

We have already mentioned (page 13) the principles stated by Hromkovič

[5]ABZ stands for Ausbildungs- und BeratungsZentrum für Informatikunterricht. It is directed by Prof. Juraj Hromkovič at the ETH (Eidgenössische Technische Hochschule) in Zürich.

and Lacher [Hromkovič J., 2017] about teaching the computer science way of thinking. However, to understand the breadth of their vision it is worthwhile to quote the following statement:

> Do not teach the final products of science, technology, and humanities, and do not consider it the highest goal to train to successfully apply them. For the latest knowledge may be found outdated with time. Teach the process of discovering new knowledge, teach the need to search for new solutions, teach the ways of collecting experience and formulating hypotheses, teach the ways of verifying hypotheses, teach how others can be convinced about the truth discovered, teach the constructive way of thinking in order to create new products and finally new technology, and teach the processes of testing and improving the products of our work.

This statement resonates strongly with the writings of the Logo's cultural protagonists. And as far as the teaching of computer science subjects they summarize as follows:

> Teaching computer science offers more than algorithmic thinking (or more general and as recently presented: computational thinking). To understand this claim, one has to have a more careful look at the development of human culture, science, and technology. This helps not only to recognize that the computer science way of thinking was crucial for the development of human society since anyone can remember, but it helps to make a good choice of topics for sustainable computer science education in the context of science and humanities. This leads to the creation of textbooks that do not focus on particular knowledge for specialists, but offer serious contributions in the very general framework of education.

The work of ABZ is sustained by a sound cultural vision but entails also the production of a great deal of educational resources, form the kindergarten to the secondary school level[6]. Interestingly, the vision also emerges from the choice made in developing the digital environments: XLogoOnline[7] and WebTigerJython[8]. The XLogoOnline environment has three sections for different age groups: two block-based for kids up to second grade and between third and fourth grade, one Logo-based for 5th and 6th grade (we'll come back to this later). Web TigerJython is a Python programming environment. It is suitable

[6]The resources include books for students and their teachers, but also software environments where the exercises and studies offered in the manuals can be done. The environments are different, considering the large target ages, ranging from 3 to 17 years. Resources include digital environments, but also unplugged exercises, available on the ABZ website or through an edition of Bebras cards, the game related to the Beaver Computing Challenge. Some of these resources are available in several languages besides German.

[7]XLogo is a programming environment for pre-school and elementary school children. It exists both online — https://xlogo.inf.ethz.ch/ — and downloadable (for Windows, OS X and Linux (32-bit and 64-bit), available at https://www.abz.inf.ethz.ch/primarschulen-stufe-sek-1/unterrichtsmaterialien/. On the same page, a logo programming booklet is also available in multiple languages.

[8]WebTigerJython is a Python programming environment, which is available both as a web service — https://webtigerjython.ethz.ch/ — and as downloadable software — https://www.tjgroup.ch/engl/index.php

for introducing high school students to Python as well as general concepts of programming. However, by loading the turtle module, it can be used as a Logo environment, with the benefits of a general language like Python. All these environments may sound different, and they are, but they have one thing in common: the fact that they are used around the logic of Logo. The first two sections of XLogoOnline use blocks instead of text-based instructions, but they are limited to conventional basic logo instructions. They are intended as an entry into the Logo language, in two successive steps. The first one allows to move the turtle by fixed amount of space or rotation while in the last one, one can use numeric parameters. WebTigerJython is a Python programming environment, but it is used with the Turtle library, which basically replicates a classic Logo environment, although integrated with all other Python capabilities.

2.3 A free software environment: LibreLogo

2.3.1 Why free software

The other environment we used is LibreLogo, a standard component of Writer, the word processor available in the LibreOffice suite. LibreOffice is free software and is widely used all over the world. Free software is defined by four types of freedom[9]:

0. The freedom to run the program as you wish, for any purpose (*Freedom 0*).

1. The freedom to study how the program works, and change it so it does your computing as you wish (*Freedom 1*). Access to the source code is a precondition for this.

2. The freedom to redistribute copies so *you can help your neighbour* (*Freedom 2*).

3. The freedom to distribute copies of your modified versions to others (*Freedom 3*). By doing this *you can give the whole community a chance to benefit from your changes*. Access to the source code is a precondition for this.

I highlighted two sentences, respectively, in *Freedom 2* and *Freedom 3*, because they express a more ethical than technical concept. That is what free software is all about. It should be noted — this is often misunderstood — that the *open source* concept is different, since it does not entail the ethical aspect: a software is said to be open source if its source code is made available, together with the executable version of the program. However any mention about ethical aspects is made. Free software is developed by individuals, who often join loose communities of developers, or non-profit organizations, sharing a common vision. Instead, open source may by developed by private economic actors who adhere to the shared development paradigm because it fits well into their marketing strategies. There are companies that develop open source projects

[9]This is the free software definition given by the Free Software Foundation: https://www.gnu.org/philosophy/free-sw.en.html

alongside traditional proprietary products because they find it convenient for their marketing strategies.

Free software is good in educational contexts because one is implicitly teaching that:

- to collaborate benefits all
- to crack proprietary software is bad because it is against the law
- it is possible to save money for the public benefit in a perfectly legal way
- free software is something for everybody and not only for those that can afford it

The last point is important in the context of public education because most national constitutions have articles establishing the right of access to education for all citizens, with no exceptions. Their statements are framed in article 26 of the Universal Declaration of Human Rights. It's worthwhile to remind it:

> *Article 26 of the Universal Declaration of Human Rights*
>
> 1. Everyone has the right to education. Education shall be free, at least in the elementary and fundamental stages. Elementary education shall be compulsory. Technical and professional education shall be made generally available and higher education shall be equally accessible to all on the basis of merit.
> 2. Education shall be directed to the full development of the human personality and to the strengthening of respect for human rights and fundamental freedoms. It shall promote understanding, tolerance and friendship among all nations, racial or religious groups, and shall further the activities of the United Nations for the maintenance of peace.
> 3. Parents have a prior right to choose the kind of education that shall be given to their children.

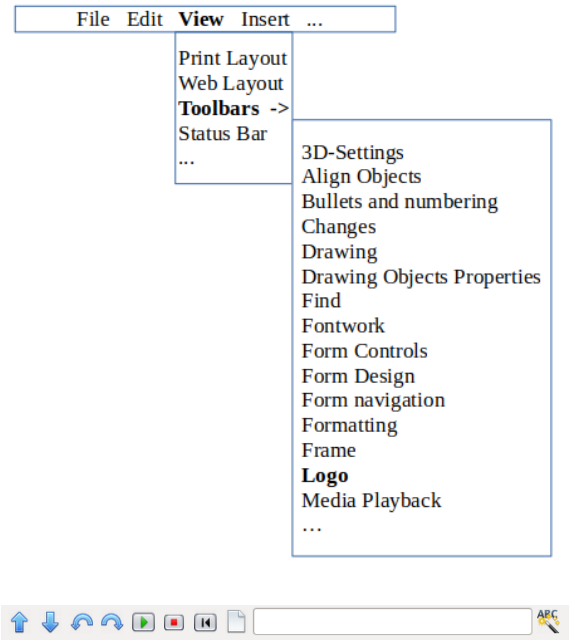
Thus, if we understand education not only as a mere question of skills, but also as a fundamental tool for growing citizens aware of the fundamental principles of democratic coexistence in an increasingly complex world, then the choice of tools and technologies used at school also plays an important role.

2.3.2 LibreLogo

LibreLogo is an option available in Writer, the word processor of the LibreOffice suite, since version 4.0. It can be activated in the available toolbars. LibreOffice is widely used, due to the wide range of functionality, compared to those of the most common commercial competitors. It also has an unprecedented level of internationalization, thanks to the contribution of developers from around the world, including ethical and linguistic minorities.

The first time you launch LibreOffice the LibreLogo toolbar is not active. Therefore you need to activate it, with the appropriate menu command: **View → Toolbars → Logo**:

Once this is done, you must close the program and relaunch it to see, among the other toolbars, also the LibreLogo one:



2.4 Differences among the languages

XLogo and LibreLogo are pretty much the same. The most significant difference is how variables are named and how they must be defined. In LibreLogo, variables are used just like most current languages, whereas XLogo adheres to the original Logo style, i.e. prepending to the variable name the " character when defining them and the : character when using them. For instance, the following codes create variables A and B, assigning values 1 and 2 respectively, then create variable C, assigning them the sum of the previous values.

```
A = 1
B = 2
C = A + B
```

Listing 2.3: LibreLogo

```
make "A  1
make "B  2
make "C = :A + :B
```

Listing 2.4: XLogo

In chapter 3 the scripts are written in Logo. Most of these have been listed in the LibreLogo version. However the translation in the XLogo version is easy. The player is encouraged to try both versions: translating a program is a good exercise. Other minor differences may be dependent on the version used, as these environments can be released in new versions. There are a few additional comments along the way.

Starting with chapter 4, scripts are listed in WebTigerJthon version, with some exception. It's better to code slightly complex programs in that environment. This is not to say, however, that most (not all) of the scripts can also be written in XLogo or LibreLogo. For instance, the example of orbits calculation (section 4.7) is listed in Python but we also tried a LibreLogo version.

Finally, in programs listing appendix 7.3 there is one exception. The last listing, about the calculation of coastline length, is written in R language. R was born as a language for statistical analysis that grew in a very powerful system

for analysing big data with the added values of performing complex statistical elaborations using vectorial optimization and for producing sophisticated graphical analysis. The use of R is not within the scope of the book. We have reported it to foster curiosity. Even R is free software: anyone can download it, copy the script in the editor and begin tinkering. [10]

[10] R can be downloaded from its site (https://www.r-project.org/). There is also a very powerful IDE for developing complex projects, RStudio, which can be downloaded from https://rstudio.com/.

Chapter 3

Do not "teach": be maieutic!

3.1 Before talking to the turtle...

Reflection — Syntonic learning

Always keep in mind that our brain evolved to optimize a multisensorial, olistic kind of learning. We evolved to learn through our bodies. Not as Sir Ken Robinson used to joke about: — Professors use their bodies mainly to bring their heads to congresses! —

And this sort of holistic learning is particular important for kids. Thus, various kind of "unplugged activities", involving physical games or practices are appropriate before facing the cold world of touchscreens or other electronic devices. Seymour Papert talked about *syntonic learning*:

> This term [syntonic learning] is borrowed from clinical psychology and can be contrasted to the dissociated learning already discussed. Sometimes the term is used with qualifiers that refer to kinds of syntonicity. For example, the turtle circle is body syntonic in that the circle is firmly related to children's sense and knowledge about their own bodies.

The following pictures show some examples of possible practices.

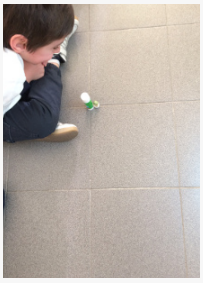


Figure 3.1: If I would be the puppet...

Figure 3.2: Pay attention to the path...

Figure 3.3: One small step, turn a little bit, again and again...

Figure 3.4: "Drawing in the flour"...

These pictures were shot by maestra Antonella Colombo, a former student of mine, now teaching in the primary school of Paderno d'Adda. She made great work in her classroom, based on our discussions of the concept of syntonic learning.

Very powerful mathematical idea — Isomorphism

We go back here to Seymour Papert for this insight: syntonicity evokes the concept of *isomorphism*. It is a very general and fundamental concept that appears in several areas of mathematics. In simple words, an isomorphism is a correspondence between two different worlds that preserves sets and relations among elements. The fact that one sees the correspondence between a square "drawn with one's own body" — for instance by walking along some square figure visible on the floor — and the square drawn by the turtle on the computer screen, is a significant step towards abstraction, a step towards the concept of *isomorphism*.

3.1.1 First steps

In any environment, students face a blank sheet first. Of course, you have to help them to begin with. But refrain from giving instructions, as far as possible — Today we are going to make something new... let's open this program (LibreOffice) — Then, everyone is faced with a blank page with pretty familiar text formatting commands at the top. Give them time (always give time...) to realize where they have landed. Do not explain that this is a word processor, so and so. Just suggest to try something, for instance typing something and let emerge and realize that this is is nothing else that one of those programs for writing text, possibly assignments or similar stuff — So what? What are we here for? — Try creating a sense of mystery. At a certain point, you may propose to type a specific word: for instance[1]: FORWARD. Discuss the meaning of this word in English, recalling the translation into their native language. Then, give this small instruction — Go over there, to the left, and press the green arrow... — Two things will happen: a green object appears, could it be an animal, with head and legs? A sort of turtle? Yes, a turtle actually, the protagonist of our story and we will call it "the turtle" in the following. But also a small window appears, telling that there is an error. It is good to stumble upon an error immediately. It's an expedient to begin creating a feeling of confidence with the error: the error as an opportunity of getting precious information. — Well — you may admit — I forgot something! — and tell them to add a number, FORWARD 100, and press the green play command again...

```
FORWARD 100
```

Listing 3.1: The first move...

There is a drawing: the turtle at the top of a vertical segment. When working with kids, it is good to present the turtle as a friend, who may help them to do something new. Thus, the turtle is a kind of living being, not so much because it resembles a turtle but because it's able to talk, even if in a peculiar way. We write FORWARD 100 and the turtle draws a segment, i.e.

[1] I used uppercase characters just for clarity, LibreLogo is "case insensitive". Also XLogo is case insensitive but it's better to use lowercase because in this way the environment provides highlighted syntax. Instead, in TigerYjthon lowercase must be used because the environment is case sensitive.

we write a command and the turtle answers by doing something related to the command — a dialogue is going on. What should be done is to let children have the feeling of playing with the turtle, following their innate aptitude for setting their own goals and exploring what amazes them. Of course this needs time but true learning deserves time - no tricks are available in this respect. And, of course again, to achieve this simple drawing you have to tell kids the command to write but you do not need to say something like - "Now I tell you how to draw a segment". Just tell the command and wait...

Probably someone will propose something: —- let's change the number... what else can we do... — Wait for ideas or questions like these ones, always keep waiting till explorations are going on. Then add something new, RIGHT 90 for instance. However, at this point, you may introduce two handy commands for deleting previous drawings and sending the turtle home at any new try:

```
CLEARSCREEN
HOME
FORWARD 100
RIGHT 90
```

Listing 3.2: The turn command

We got the same drawing but the turtle turned right by an angle of 90°. However, do not explain what 90 means, let them find out for themselves.

Powerful idea — *Divide et impera* This is one of the commonly quoted elements of the so-called "computational thinking". There is a lively debate about the exact definition but, to tell the truth, several of its aspects have always been sound habits of a good scientist. This is the case of "breaking the problem down into smaller parts". I'm quoting it right here, because of how the issue may emerge when teachers are "doing coding" with devices such as the Bee-Bot or similar ones. Sometimes, when planning the coded path with text or cards, they interpret turning commands as steps, thus confusing two different acts: turning should not involve movement, as well as stepping should not involve turning. This will be clearer when we will comment on the "state" powerful idea, later on. Here it's worthwhile to point out the importance to break down a problem or a process in smaller clearly defined sub-parts.

The reason for I'm insisting so much about what children can do is that I had several opportunities to play turtle with them. Really a few say it's too difficult, most just play and get excited to let the small animal create funny things. Too often we forget how kids crave hard. Once, trying to draw a house a girl came out with kind of a bizarre castle. I praised her, commenting on how marvellous was this drawing, she was so proud. Then, when I came back to her, after having considered the works of other kids, I found her in tears, sobbing desperately: she had accidentally erased the commands in a way that I was no more able to recover. I felt guilty for not having taught her how to save the work once in a while.

By the way, therefore, if you start creating some more complex stuff you like, don't forget to save the document once in a while. It's so easy.

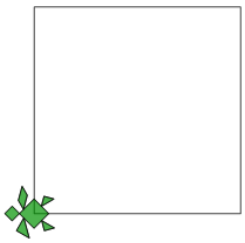
3.1.2 Towards first geometrical shapes

Most probably someone will come out saying: — Let's do a square! — or — Let's do a triangle... — or something else. Just go and follow their proposals, even if here we are going to show the example of a square. If nobody comes out with an idea, you may help: — Why don't we try to make a ...?

Let's go on with the square. Often kids achieve this easily, landing to something like:

```
CLEARSCREEN
HOME
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
```

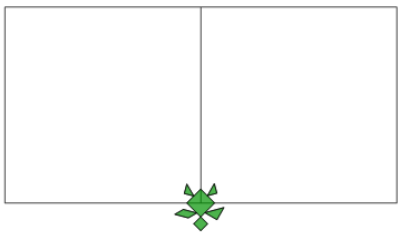
Listing 3.3: The first figure



The first of your kids who will achieve this result will rejoice, of course, but is the task really finished? At first glance yes, since the square is there, however, the point is more intriguing than that. In order to work properly, that is, in order to use the principle of dividing a job in smaller parts, the turtle should end always in the initial state, after having drawn a geometrical figure. What we mean is that if we are invoking two times the drawing of a square, the turtle should draw the same square, superimposing the second to the first one. Instead, in this case, see what happens if we try to repeat two times the same code:

```
CLEARSCREEN
HOME
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100

FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
```

Listing 3.4: What does it really mean that a square is "finished"?



This is quite different from the expected results. The problem is that our code for drawing a square is difficult to use repeatedly, since we have to reflect on the landing position of the turtle to foresee where the next square will be drawn. These considerations drive us towards the definition of the important concept of *state* of a system. When a process — for instance, for drawing a square — has the property to leave the turtle in the initial state, is said to be *state-transparent*.

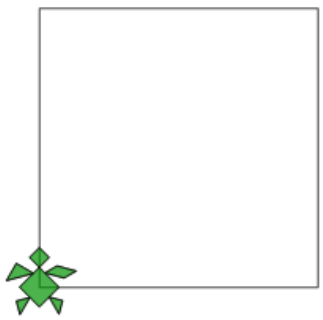
Powerful idea — State of a system

It's very natural to think about the "state of the turtle" since by knowing its position we do not know everything: we also need to know where the turtle is looking. That is, in order to know everything about the turtle we need to know both its position and direction. Therefore, when drawing a square, instead of just talking about its initial position, we should specify position and direction, i.e. three numbers: two spatial coordinates in the plane and a direction angle. Thus, the turtle allows us to employ, in a practical way, the concept of "state" of a system, which is a crucial concept in all basic sciences. For instance, in classical mechanics, the state of system is given by the knowledge of spatial coordinates and speed components of all its particles. In the world of turtle geometry (we'll talk about later on) the turtle state is given by its two spatial coordinates and the direction angle it is pointing to.

Returning to our square, it is easy to see that what is missing is a last turn right — I added numbers at the beginning of lines, in order to refer them more easily:

```
1  CLEARSCREEN
2  HOME
3  FORWARD 100
4  RIGHT 90
5  FORWARD 100
6  RIGHT 90
7  FORWARD 100
8  RIGHT 90
9  FORWARD 100
10 RIGHT 90
11
12 FORWARD 100
13 RIGHT 90
14 FORWARD 100
15 RIGHT 90
16 FORWARD 100
17 RIGHT 90
18 FORWARD 100
19 RIGHT 90
```

Listing 3.5: We added RIGHT 90 instructions at lines 10 and 19



Now, if we we repeat this piece of code a second time, the turtle will draw a second square, perfectly superimposed to the first one.

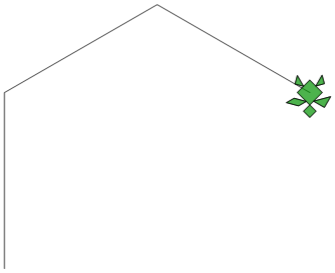
And why not a triangle?

This is a typical goal, often finalized to build the roof of a house, on top of the previous square. Usually, both kids and grownups are pretty confident to achieve it after having been successful with the square experience. The idea is to work on the square code by changing the number of sides — pretty easy — and, of course, the angle, to obtain an equilateral triangle. In the majority of cases, without distinction of age, people decide to substitute 90°with 60°, because the internal angles of the equilateral triangle are equal to 60 degree. So, what's the result?

Try to think about this before moving on to the next page. Maybe you could use this page to make a pencil drawing...

```
CLEARSCREEN
HOME
FORWARD 100
RIGHT 60
FORWARD 100
RIGHT 60
FORWARD 100
RIGHT 60
```

Listing 3.6: We wanted to draw a triangle but...



Quite disappointing, isn't it?[2]

Reflection — Scientific knowledge process

The process of building scientific knowledge is iterative: first imagine a theoretical description, then try to devise an experimental confirmation, finally go back to refine the theory if necessary. The triangle example is an example of how naturally the process is activated with Logo. In this case we know the theory, which says that the internal angles of an equilateral triangle are equal to 60°. This is correct of course. What's wrong is how we applied this knowledge: the angle we specify in the RIGHT instruction is the *deviation angle* from the current direction of the turtle, not the internal angle of the figure we are trying to build. The deviation angle is supplementary of the internal angle we want to produce, that is $180° - 60° = 120°$. Once they see the disappointing result, people realize the error immediately and go to write the correct code.

3.1.3 Making things easier...

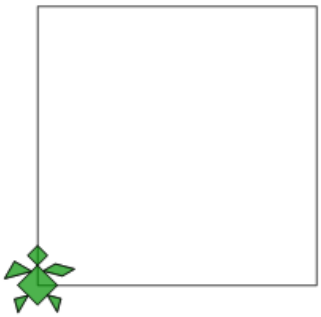
Repetitions

In order for the turtle to draw a square, some typing was necessary. In particular, we had to repeat a couple of commands four times. This isn't terrible, but imagine if you had some more complex piece of code and wanted to repeat it a lot, say hundreds or thousands of times - this can happen when you are more familiar with coding. It would be painful to type all this stuff, or even simply impossible. Fortunately, turtle is able to understand some repetition commands. The most common is the following:

[2] This error is so common that even Abelson and diSessa quoted it in their Turtle Geometry [Abelson and diSessa, 1980, pag. 7]

```
CLEARSCREEN
HOME
REPEAT 4 [
   FORWARD 100
   RIGHT 90
]
```

Listing 3.7: In order to execute a sequence of instructions more than once we use the REPEAT command: all instructions within square brackets are repeated.



When the turtle finds, say, a REPEAT 100 instruction, it will execute all the commands listed among the two square brackets, [and], for one hundred times. Instead of typing one hundred times the same sequence of instructions, it is sufficient to enclose them within the square brackets specifying the number of repetitions.

Creating new commands

As far as we have seen so far, the turtle executes the instructions while "reading" it or, if you prefer, while "listening" to it. However, the turtle not only executes the instructions one after the other, like an automaton, but it is also able to memorize many commands at once. Let's explain this with an example.

```
1 CLEARSCREEN
2 HOME
3
4 TO SQ
5 REPEAT 4 [
6    FORWARD 100
7    RIGHT 90
8 ]
9 END
```

Listing 3.8: Defining the "SQ" new command

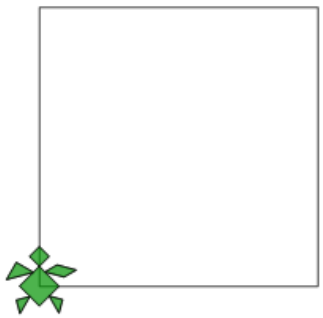
When the turtle finds an instruction beginning with the word TO, followed by a name, it stops executing the successive instructions, as usual. Instead, it "sits and listens to the commands", learning them one by one without doing nothing else, until it meets the END command. What's going on when you try to execute this code in LibreLogo? Nothing! However, if we think about

it, there is nothing to execute in this code, actually, except for the first two commands, CLEARSCREEN and HOME.

The following code shows how to exploit this ability of the turtle.

```
1  TO SQ
2     REPEAT 4 [
3        FORWARD 100
4        RIGHT 90
5     ]
6  END
7
8
9  CLEARSCREEN
10 HOME
11
12 SQ
```

Listing 3.9: Defining and applying the new "SQ" command. We colored in blue our new command.

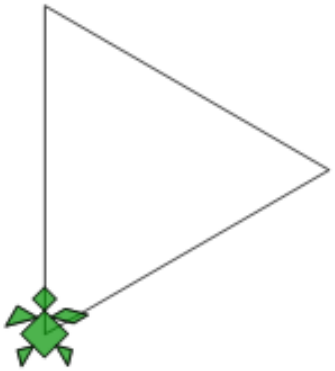


Instruction number 1 is composed by two parts: TO tells the turtle to sit down and listen and SQ is the name we can use as a new command. This means that, if the turtle finds this new command, it will remember all the instructions memorized between TO and END, executing them in sequence. That's why we have put the instruction SQ at the end of the code. More precisely, between instructions 1 and 6 the turtle sits and listens - nothing happens. Starting with instruction 7 the turtle will execute all instructions, one at a time.

Since we have put the code at page 38 between the TO and END commands, we got a square. Probably, sooner or later someone will try to define other new commands, for instance the triangle, again.

```
TO TR
   REPEAT 3 [
      FORWARD 100
      RIGHT 120
]
END


CLEARSCREEN
HOME

TR
```

Listing 3.10: The "TR" new command



Reflection — Encapsulating functionality in new commands: modular thinking

As soon as students grasp the meaning of the TO...END construct, they feel a sense of empowerment. First because one can encapsulate even a very complex sequence of instructions in just one command and, second, because one can extend and customize the Logo commands set limitlessly. This perceptions lead easily to a modular way of thinking, which means trying to simplify a large process in a limited number of blocks that may be combined easily and, possibly, even reused in other contexts. This is an extremely useful step towards a scientific way of thinking, by means of which more complex tasks can be simplified, reducing the occurrencies of errors and improving communication.

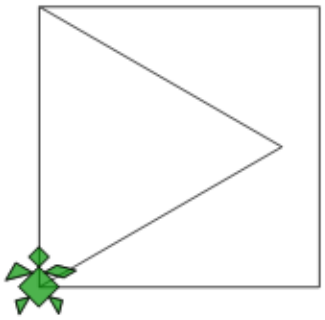
As a consequence of this achievement, the level of self-determined goals is raised for instance for drawing new figure by combining previously made pieces. A typical example is that of a house composed by a square and a triangle.

Another classic — drawing the house

So, let's try to make a house. It should be easy: first draw the square and then put the triangle at the top. Very often people write a piece of code of the following kind...

```
TO SQ
   REPEAT 4 [
      FORWARD 100
      RIGHT 90
   ]
END

TO TR
   REPEAT 3 [
      FORWARD 100
      RIGHT 120
   ]
END


CLEARSCREEN
HOME

SQ
TR
```

Listing 3.11: The "usually wrong" first house...

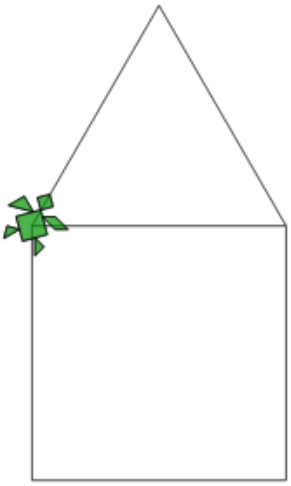


But this is not what it was intended. One realizes that some more reflection is needed and that something has to be done between the two new commands, SQ and TR. In order to solve the problem you must put yourself in turtle's feet: where is it and in which direction is pointing it, after having drawn the square? And from this *state*, which is the fist move when tackling the triangle? When reflecting on these questions, it is clear, that before starting to draw the triangle, the turtle should move to the upper left corner of the square, by means of a FORWARD 100 instruction. By doing so, the drawing of the triangular roof will begin at the top of the square, which is good. However, in this way we achieve just a translation of the triangle and our house will look like if it would be opened like a can. For this reason, before starting to draw the roof we should let the turtle rotate right by an angle of... well you should discover it by yourself!

This is the result you should have achieved:

```
1  TO SQ
2     REPEAT 4 [
3        FORWARD 100
4        RIGHT 90
5     ]
6  END
7
8  TO TR
9     REPEAT 3 [
10       FORWARD 100
11       RIGHT 120
12    ]
13 END
14
15
16 CLEARSCREEN
17 HOME
18
19 SQ
20
21 FORWARD 100
22 RIGHT 30
23
24 TR
```

Listing 3.12: The correct first house...

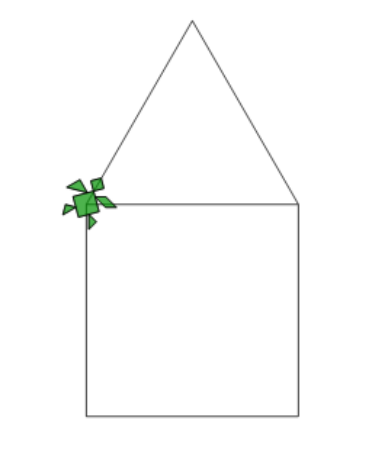


In instruction 19 the turtle draws the square, then with 21 and 22 it achieves the correct status (position and orientation) to start drawing the triangle by means of instruction 24.

Reflection — Again on modular thinking while drawing the house

When you find out you can draw an object like that, you feel a sense of empowerment: this new language is extensible, at your will. You can build new blocks of functionalities and combine them freely. Once the house is there, it comes spontaneous to take one more step: perhaps we could put everything in a new command, HOUSE for instance. Try to reflect on the following code, which draws the same house:

```
TO SQ
   REPEAT 4 [
      FORWARD 100
      RIGHT 90
   ]
END

TO TR
   REPEAT 3 [
      FORWARD 100
      RIGHT 120
   ]
END

TO HOUSE
   SQ
   FORWARD 100
   RIGHT 30
   TR
END


CLEARSCREEN
HOME

HOUSE
```



Listing 3.13: The "HOUSE" COMMAND

Where the names of new commands are emphasized: **SQ**, **TR** and **HOUSE**. Now we can send the turtle around, disseminating houses everywhere!

3.2 Making things smaller and larger

3.2.1 The Turtle is able to learn symbolic names

It's time to introduce another magic of the Turtle. It was fun to draw many houses so easily, but what if we wanted to create smaller and larger houses? Here you have the new trick: the Turtle is able to use *symbolic names*. Try this:

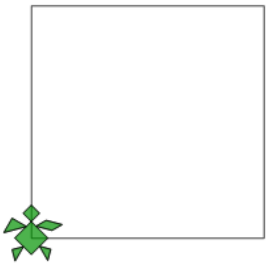
```
SIDE = 100
FORWARD SIDE
```

Listing 3.14: "SIDE" is a parameter that has value 100

We used a new name here, SIDE, assigning a number to it. In this way the Turtle knows that, every time it encounters the name SIDE, it must use the number 100 instead. This is very useful when you have to use many times the number 100 in different instructions. For instance, if we go back to our first way to draw a square, we can write the code in the following way:

```
CLEARSCREEN
HOME
SIDE = 100
FORWARD SIDE
RIGHT 90
FORWARD SIDE
RIGHT 90
FORWARD SIDE
RIGHT 90
FORWARD SIDE
RIGHT 90
```

Listing 3.15: Using the SIDE parameter



Thus, if we want to make a smaller square we have to change just the value of SIDE.

In case you are using XLogo

In case you use XLogo the code for assigning symbolic names is somewhat different. For instance, in order to assign the value of 100 to SIDE one has to type make "SIDE 100:

```
home
make "SIDE 100
forward :SIDE
right 90
forward :SIDE
right 90
forward :SIDE
right 90
forward :SIDE
right 90
```

Listing 3.16: How one should write the previous code in the XLogo environment

Therefore, in order to run the same code in XLogo one has to change accordingly the assignment instruction and substitute :SIDE in place of SIDE, everywhere.

Reflection — The door to algebra

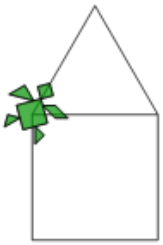
As it is well known, algebra is based on symbolic representation of numbers. Algebra is about generalization. The use of symbolic names in place of numbers in Logo creates the mental device for manipulating symbolic representations of numbers. A crucial step towards mathematical thinking. We used the expression "symbolic names" so far, in the following, depending on the context, we will call them *variables* or *parameters*.

3.2.2 The Turtle is even smarter: passing parameters to new commands

We have seen how to create new commands, for instance the HOUSE one. Let's suppose we would like to draw houses of different sizes. See here how we can do that:

```
TO SQ SIZE
   REPEAT 4 [
      FORWARD SIZE
      RIGHT 90
   ]
END

TO TR SIZE
   REPEAT 3 [
      FORWARD SIZE
      RIGHT 120
   ]
END

TO HOUSE SIZE
   SQ SIZE
   FORWARD SIZE
   RIGHT 30
   TR SIZE
END


CLEARSCREEN
HOME

SIZE = 50
HOUSE SIZE
```

Listing 3.17: How to draw houses of different sizes with the same command using the SIZE parameter



The possibilities are unlimited. Look what a primary teacher student was able to do by exploiting the use of parameters in new commands:

The possibilities are endless, especially if we consider that we may give as

many parameters as we want to a new command. Try to play with this:

```
TO RECT A B
REPEAT 2 [
   FORWARD A
   RIGHT 90
   FORWARD B
   RIGHT 90
]
END

CLEARSCREEN
HOME

A = 100
B = 200
RECT A B
```

Listing 3.18: A new command with two parameters

3.3 Doing round things

3.3.1 Syntonic learning

Sooner or later someone will come out saying: — Why not making something round, like a circle? Remember figures 3.1, 3.2, 3.3 and 3.4 on page 30? The playful exercises shown in these figures were followed by lively discussions about letting the turtle produce the same geometrical shapes. Maestra Antonella reported a piece of these discussions among her eight year old kids:

> ...
> Girl 1 — With always LEFT the turtle turns on itself but it doesn't move!"
> Boy 1 — We must change the numbers...
> Boy 2 — We must bend!
> Girl 1 — Yes, I said it, we have to keep turning left!
> Girl 2 — ... but if we keep just turning left we get just a small point...
> Girl 1 — I got it: we have to make FORWARD 1 LEFT 1!
> Girl 2 — ... and keep doing it...
> Boys — Wow! It's turning in circle.
> Girl 1 — But it's not a true circle, just a piece...

Teacher Antonella took the opportunity to introduce the REPEAT command.

> Boy 2 — Well, we have to repeat it the right number of times, but how many? That's difficult...
> Boy 1 — Let's try, 10... oh, too small
> Girl 1 — We must do it much longer, let's try 300...
> Boy 2 — No, 355!
> Girl 3 [3] — Oh that's really easy: the right number is 360.
> Boy 1 – Mmh... no wonder... you're a grown-up!

Reflection — Syntonic learning described through a dialogue
The story of Antonella, shown here by her shots and the dialogue among her kids, is a remarkable example of Papert's pedagogic ideas: activation of syntonic learning through physical activities and fostering of self-determined goals, freedom of exploring. Everything here is about discovering the rules to achieve a personally significant goal instead of teaching rules to solve problems that make no sense to the learner.

Let's now rework figure 3.3 in this way:

[3] That day some older girls were hosted in Antonella's classroom because their teacher was temporarily missing.

Figure 3.5: A small step and turn a little bit, a small step and turn a little bit...

This is a sort of "syntonic picture", in that it connects a specific physical action with a fragment of code, i.e. an abstract symbolic representation of that action. The correspondence is nice: the girl has to make a small step, in order to remain on the yellow ring. At the same time she has to turn the forward feet a little bit, for the same reason. This helps her to imagine that the turtle should receive commands with small numbers, FORWARD 1 and LEFT 1, to let it drawn something circular.

Powerful mathematical idea — Differential equations

"Powerful ideas" is a typical expression of Seymour Papert, together with the conception of learning environments that act as "incubator for powerful ideas". We mean here ideas that are pre-existent, not belonging to the learning environment, but that are activated by it. At the light of nowadays neuroscientific knowledge of learning, we could define the concept even more precisely: the activation of a powerful idea by an appropriate environment turns out in the creation of a neuronic configuration, sort of a first path in a bush, that in future may host a new mental device. The powerful idea is now activated in an unwittingly way but in future it may flourish in a thoroughly definite concept.

This is exactly the case of the syntonic experiences of drawing circles we have just described. The key point here is that in the FORWARD 1 LEFT 1 code there isn't any reference to global attributes characterizing

the concept of circle: no mention of a center, no mention of a radius. The turtle is given just a "local rule" and nothing else. Then we get a circle. Is there something wrong? Is there something inaccurate? Something not "enough mathematical"? No, this kind of "local description" is perfectly legitimate in mathematics: it falls into the world of *differential equations*. The following is the differential equation of the circle:

$$\left\| \frac{d\vec{T}}{d\vec{S}} \right\| = k, \tag{3.1}$$

where k is called the *curvature* of the circle and it is given by $k = 1/r$, where r is the radius. From the mathematical point of view, equation 3.1 is a *differential equation* because it's about variations, relating them to other quantities. In our case, talking in intuitive terms, equation 3.1 tells us how much the direction $\vec{T}$ changes for a given variation of the position $\vec{S}$ along the trajectory, that is the girl's step length, and this amount is given by curvature k, which is constant in the circle case.

The following diagram shows the "small spatial displacement", $d\vec{S}$, and the "small direction change", $d\vec{T}$.

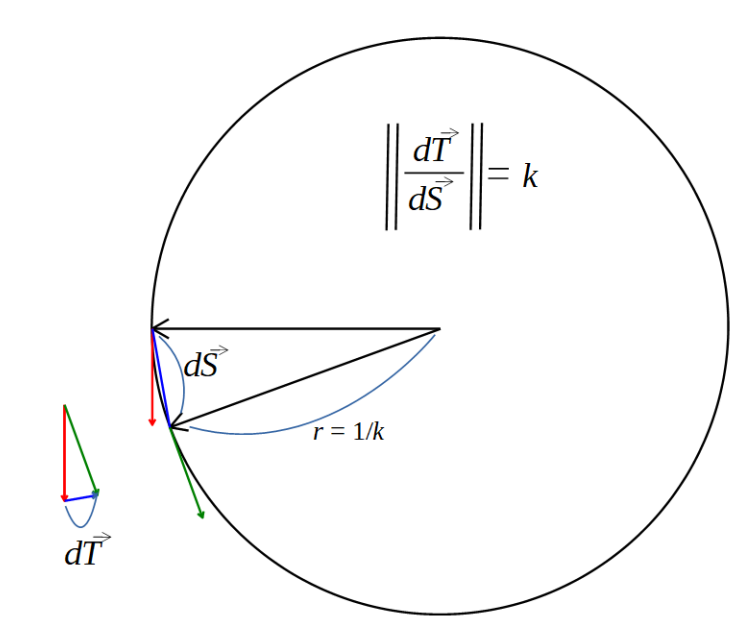


Figure 3.6: "Syntonic math"

The important concept is that, in a circle, the variation of the direction changes in a constant way while moving along the trajectory. This

constance is expressed by the constant value of the curvature k. In the following picture the relevant elements of the diagram are superimposed to the step of the girl. Here the constance of the curvature is maintained by the girl's effort of making regular steps and regular bending of the forward foot.

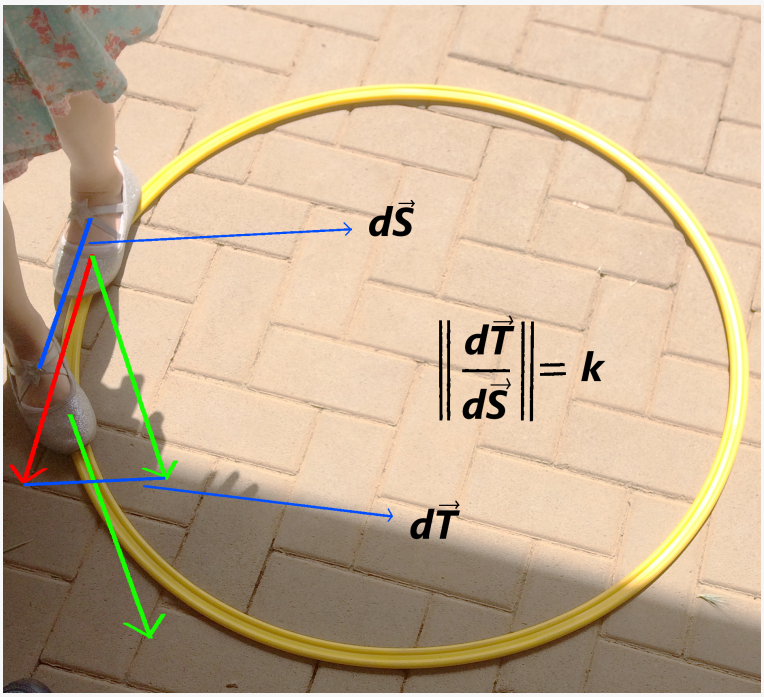


Figure 3.7: "Syntonic math"

Of course, our girl is not supposed to know anything about differential equations and vectors[a]. However, by means of this activity, she has discovered a new way to look at a geometrical figure, a way lending to the concept of differential equation. A perspective where a figure (eventually a physical phenomenon) is builded by means of a precise behaviour which is "purely local", without the need of any synthetic description of the final figure. Differential equations are a fundamental tool needed in all fields of science.

This could be the first step of a spiral curriculum, with successive explorations of the circle. Let's explore some of the successive steps, leading to other powerful matematical ideas.

[a] In exact mathematical terms, $\vec{T}$ and $\vec{S}$, represent vectorial quantities, having direction and intensity, $\frac{d\vec{T}}{d\vec{S}}$ is the derivative of the unit tangent vector $\vec{T}$ with respect to the position vector along the trajectory, $\vec{S}$. The symbolic representations $d\vec{T}$ and $d\vec{S}$ are not

finite quantities *per se*. They can be used in expressions involving the concept of *limit*, the basic tool of mathematical analysis for embedding the infinity or the infinitesimal in finite quantities. For instance the so called derivative $\frac{d\vec{T}}{d\vec{S}}$ is the result of a *limit* operation, where the imaginary very small step, at *limit*, is approaching 0. Thus we get a finite quantity, expressing the local rate of change of a variable, $\vec{T}$, with respect to another, $\vec{S}$. Later on, when talking about successive approximations, the same kind of concepts will been evoked.

3.3.2 Spira mirabilis

Figure 3.8: Jacob Bernoulli's headstone, in Basler Münster.

This is the headstone of Jakob Bernoulli (1654-1705), which can be seen in the Basler Münster. Jakob Bernoulli studied the spiral extensively. He was fascinated by its mathematical properties, which he described in a treatise entitled *Spira Mirabilis*. He was so impressed that asked to have the spiral carved on its headstone with the inscription *Eadem Mutata Resurgo*: although changed, I

keep arising the same.

Figure 3.9: The wrong spiral engraved in the tombstone: Archimede's instead Bernoulli's one...

Unfortunately, the sculptor carved the wrong spiral. Bernoulli's wonder was lifted by the logarithmic spiral, not the Archimedean one, which can now be seen on his tombstone. The difference is crucial, since the two spirals grow in fundamentally different ways.

Let's go back to figure 3.5 on page 49: by keeping doing a small step and turning a little bit regularly, we approximate a circle. But what happens if, for instance, we reduce step size? Well, we get a kind of spiral, which shape depends on how we reduce the step length. These effects can be easily explored in Logo.

```
TO SPIRA STEP ANGLE
  REPEAT 400 [
    FORWARD STEP
    RIGHT ANGLE
    STEP = STEP + 0.075
  ]
END

TO SPIRB STEP ANGLE
  REPEAT 100 [
    FORWARD STEP
    RIGHT ANGLE
    STEP = STEP + STEP * 0.035
  ]
END
```

Listing 3.19: SPIRA: Archimedean - SPIRB: Bernoulli.

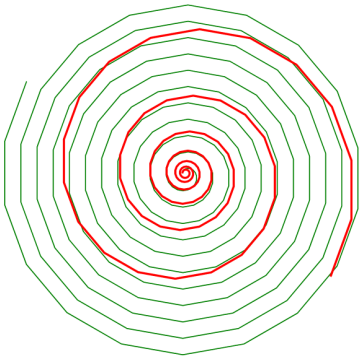


Figure 3.10: Archimedean (green) and Bernoulli's (red). Both have been called with STEP=1 and ANGLE=20.

Both programs, SPIRA and SPIRB, differ from that of a circle by the fact that they have an instruction (N. 5 in SPIRA and N. 13 in SPIRB) where the parameter STEP is increased by a certain amount at every step. The parameters have been arranged to show the basic difference between the two spirals. The two programs differ for instructions 5 and 13, where the STEP parameter is increased, but in SPIRA (Archimedean) it is increased by the fixed amount 0.075 whereas in SPIRB (Bernoulli) it is increased by STEP*0.035, that is by a quantity proportional to STEP. This means that in the former case the coils grow by constant amounts whereas in the latter the coils become larger and larger at every turn. The Bernoulli spiral is called *logarithmic* and can be found in a great number of natural shapes. Among these there are the ammonite fossils.

```
TO SPIRB STEP ANGLE
  REPEAT 300 [
    FORWARD STEP
    LEFT ANGLE
    STEP = STEP + STEP * 0.005
     ]
END

RIGHT -90
SPIRB 1 2.5
```

Listing 3.20: SPIRB with parameter adjusted to approximate the ammonite curvature.

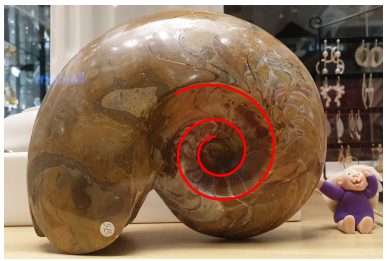


Figure 3.11: Bernoulli's spiral superimposed to an Ammonite fossil shot from a shop in the Bahnhofstrasse area, in Zürich.

Powerful mathematical idea — Linear and exponential growth

By playing with these spirals students get in touch with the essence of linear and exponential variations.

The equation of Archimedean spiral, expressed in polar coordinates, is

$$r = a + b\theta \tag{3.2}$$

This means that the radius r grows proportionally with the angle θ.

The equation of the Bernoulli's logarithmic spiral is:

$$r = ae^{k\theta} \tag{3.3}$$

In this case r grows exponentially with angle θ.

How much of these explanations should be provided to the students, depend at which point of the spiral (!) curriculum they are. What is important is to get in touch early with the linear-exponential dichotomy.

Powerful mathematical idea — Self-similarity → fractals

Among the mathematical properties of the logarithmic spiral that fascinated Bernoulli so much there is also the so-called self-similarity. Let's add a shift θ_s to the angle θ, which means rotating the spiral by an angle θ_s.

$$r = ae^{k(\theta+\theta_s)} = ae^{k\theta}e^{k\theta_s} \tag{3.4}$$

This means that by rotating the spiral we just change the scale, by a factor $e^{k\theta_s}$, while the shape remains exactly the same, i.e. $e^{k\theta}$. Vice versa, if we multiply the spiral by a scale factor, S

$$r = Sae^{k\theta} = ae^{k\theta+\ln S} = ae^{k\left(\theta+\frac{\ln S}{k}\right)} \tag{3.5}$$

we obtain the same spiral, just rotated by an angle $\frac{\ln S}{k}$.

The property of retaining the same shape while changing scale is very common in nature. It's the key to fractals, as we shall see in chapter 6.

3.3.3 Golden spiral

The Bernoulli's logarithmic spiral is related to the so-called *golden spiral*. More precisely, the *golden spiral* is a particular type of logarithmic spiral, where the rate of growth of the spiral depends on the *golden ratio*, one of the magic number of mathematics, ubiquitous in nature. Its value can be calculated with $\phi = \frac{1+\sqrt{5}}{2}$. It's approximated value is 1.618[4].

There is a nice approximation of the *golden spiral*, which can be constructed with paper, pencil, compass and scissors — its' good to mix technologies. You can proceed as follows:

1. Cut a square of paper, for instance with side of 10 cm
2. Draw a quarter of circle inscribed in the square, from one corner to its opposite

[4]The *golden ratio* is an irrational number, therefore it cannot be represented exactly with a finite number of digits.

3. Cut another square with side $10/1.62 = 6.17$

4. Again, draw a quarter of circle inscribed in this smaller square, from one corner to its opposite

5. Place the small square adjacent to the previous one in such a way that the first quarter of circle continues with the second one, without interruption

6. Repeat steps 3-5 until you can manage the progressively smaller squares

You will get something of this kind:

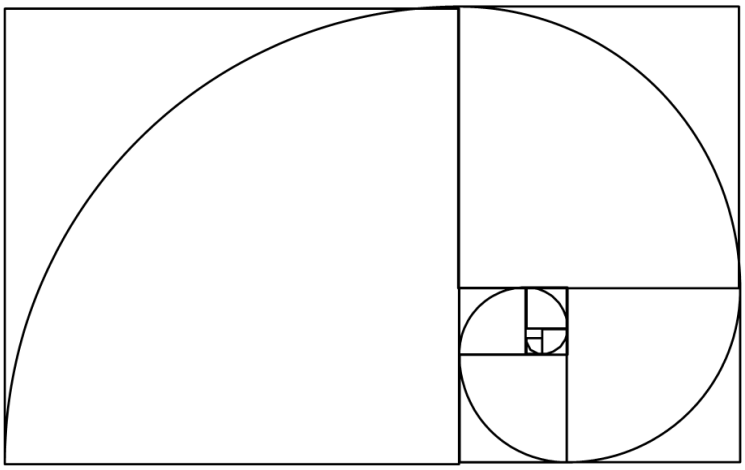


Figure 3.12: Golden spiral: with Logo
but a paperwork as well...

This is a good approximation of a *golden spiral*, which in turn is a *logarithmic spiral*, or a *spiralis mirabilis*, in Bernoulli's words. These figure was made with Logo. As usual, the result can be achieved in many ways. You can try on your own to reproduce with Logo the physical construction of the spiral.

3.3.4 POLY

Let's go back to the triangle and square codes:

```
TO TR SIZE
   REPEAT 3 [
      FORWARD SIZE
      RIGHT 120
   ]
END

TO SQ SIZE
   REPEAT 4 [
      FORWARD SIZE
      RIGHT 90
   ]
END
```

Listing 3.21: Let's look at what the triangle and square codes have in common...

The two codes are similar. What are the differences? And what do the codes have in common?
The differences are:

- the command names, TR and SQ
- the number of repetitons, 3 and 4
- the turning angle, 120 and 90

In a nutshell, the turning was done "in three goes" or "in four goes".
What else can we explore here?

Let's try to think a bit in the mathematicians way, i.e. by making things as simple as possible in order to squeeze the essence:

```
TO POLY SIZE ANGLE
   REPEAT [
      FORWARD SIZE
      RIGHT ANGLE
   ]
  END
```

Listing 3.22: The POLY code

The POLY program is an important object of turtle geometry. By playing around with POLY you may discover some pretty and relevant facts about turtle

geometry, geometry in general and computers. But more importantly, you learn to explore, formulating questions, setting and achieving goals on your own.

If you run this code, for instance in this way:

```
TO POLY SIZE ANGLE
   REPEAT [
      FORWARD SIZE
      RIGHT ANGLE
   ]
  END

POLY 100 120
```

Listing 3.23: The POLY code

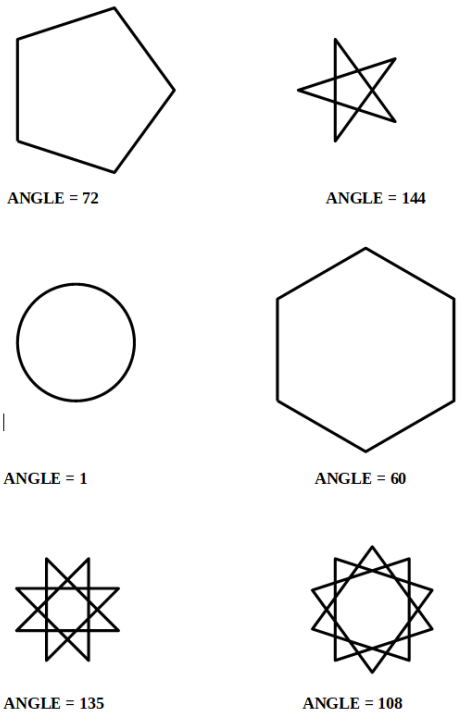
The turtle will draw a triangle, which is not a great news, since we already learned that by drawing sides with 120°turns one gets an equilateral triangle. The first observation is that, once the triangle has been drawn, the turtle is keeping drawing it, again and again. We'll come back on this aspect later on. For now, to stop the turtle, click on the red STOP button, in the LibreLogo toolbar[5]. Instead, let's go straightforward to play, taking advantage from the super concise writing, where the POLY code is reduced to its essential pattern: no numeric parameter hanging around, just the ANGLE and SIDE, which are the important parameters, especially the ANGLE one. By playing with different values of SIZE and ANGLE parameters, think about POLY as a kind of magic box for creating new objects. Try reflecting on the different effects you get by changing SIZE or ANGLE. Which are more interesting? Which kind of figures are coming out? Just polygons? Or also something else?

Those willing to experiment by themselves, may stop here and try by themselves, before going to the next page.

[5]Similarly, yo can stop the process both in the XLogo and the TigerYjthon environments, by means of the red STOP button, more or less in the same position, at the top left in the window.

```
TO POLY SIZE ANGLE
   REPEAT [
      FORWARD SIZE
      RIGHT ANGLE
   ]
  END

POLY SIZE ANGLE
```

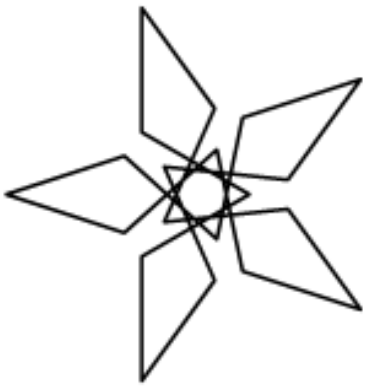
Listing 3.24: The SIZE parameter is just a scale factor.



This is just an example, we invite the reader to experiment on its own. It is fun to explore the variety of shapes and also how they grow. One can also go on tinkering with the content of POLY, inserting other instructions to see what happens. Like this, for instance:

```
TO POLY SIDE ANGLE
   REPEAT [
      FORWARD SIDE/3
      RIGHT 60
      FORWARD SIDE/3
      LEFT 120
      FORWARD SIDE/3
      RIGHT 60
      FORWARD SIDE/3
      RIGHT ANGLE
   ]
END

POLY 100 144
```

Listing 3.25: An example of POLY variation, with ANGLE = 144



In all these cases, it is interesting how some figures only need a few cycles to be completed while others need many more iterations. The question arises naturally: how many cycles are needed in POLY in order to "close" the figure, for a given value of ANGLE? Probably, you alredy found that for certain figures the answer can be found rather easily. Basically, among the previous examples we found two kind of objects: regular polygons and sort of stars. When ANGLE = 120, we get the triangle, with 90 the square, with 72 the pentagon. Since 120 x 3, 90 x 4 and 72 x 5 all make 360 we can conlude that, to draw a polygon with N sides, the turtle has to make 360 / N turns. This is a rule derived empirically, which means that we may not be sure that it is universally true. However, the *Total Turtle Trip Law*[6] holds true:

Total Turtle Trip Law *If a turtle takes a trip around the boundary of an area and ends up in the state in which started, i.e. same position and same heading, then the sum of all turns will be 360°.*

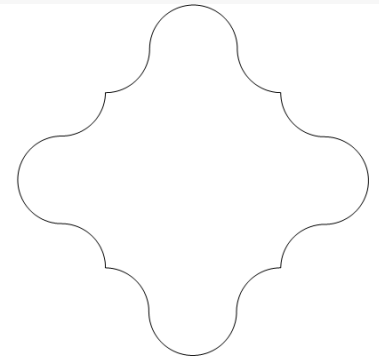
We used the word "law" because it reminds us of something that is always true, in every condition, like the laws that we must respect in our everyday life. In mathematics there are axioms, or postulates, upon which whole theories are founded; and theorems, i.e. statements which are demonstrated to be true starting from axioms. In physics we have principles, of energy conservation, of dynamics and so on. In turtle geometry there are some theorems which hold true: the Total Turtle Trip Theorem (the proper name) is the first one.

[6]This is Papert's enunciation [Papert, 1993, pag. 76:] of the Total Turtle Trip Theorem. We are using here the word "law", which is more appropriate when talking to kids.

Powerful idea: Concept of "law"

By playing with the constance of a results, independently from the path, we are developing the important conceptual device of *invariant*, such as laws or principles in physics, axioms or postulates in mathematics, which are the pillars needed by any kind of scientific theory. It is a way to give a structure to the narration of science. With children a little older, one can try to build some strange figure.

```
1
2  CLEARSCREEN
3  HOME
4  HIDETURTLE
5
6  T =0
7  REPEAT 4 [
8     FD 100
9     RT 90
10    T = T +90
11 ]
12
13 PRINT T
14
15 CLEARSCREEN
16 HOME
17 PENUP
18 LEFT 90
19 FORWARD 100
20 PENDOWN
21 RIGHT 90
22
23 T =0
24 REPEAT 4 [
25    REPEAT 90 [
26       FORWARD 1
27       LEFT 1
28       T = T  - 1
29    ]
30    REPEAT 180 [
31       FORWARD 1
32       RIGHT 1
33       T = T  + 1
34    ]
35    REPEAT 90 [
36       FORWARD 1
37       LEFT 1
38       T = T -1
39    ]
40     RIGHT 90
41     T = T + 90
42 ]
43
44 PRINT T
```



Listing 3.26: A distorted square, to show invariance of total turn

For instance here first we drew a square, accumulating in the variable T the deviation angles, printing the final value at the end, which turns out to be 360, of course. Then a new figure is drawn, which is the same square

but with highly distorted sides, again always accumulating the deviation angles in T, negative for left turns and positive for right turns. At the end the final value of T is printed, which is, again, 360. The idea is to let them create challenges: let's see if with this crazy shape it's still true...

Powerful idea: Concept of integration

In the previous example we were accumulating a quantity along a path, by summing all deviation angles while the turtle was travelling. This is the basis of a fundamental operation of mathematical analysis: the integration. In the previous example we have proposed a simple code using for instance an instruction such as $T = T \pm 90$, to update the current value stored in the variable T. However, with kids that are still not sufficiently familiar with programs, this work can be done in parallel, by reading the code and, at the same time, taking note of the values with paper and pencil, summing them by hand.

Of course, we have not to talk about integration explicitly with the students — this would be a nonsense. It is the process, the direct experience that matters. The opportunity of creating a new mental device, which one day will help to accommodate the formalisation of the concept.

Reflection — The limits of the machine (and of theory): How can the execution of a program unintentionally become a never-ending story?

This is the title of a section (2.4) of Juraj Hromkovič's book, "Algorithmic adventures" [Hromkovič, 2009, pag. 62]. It is perfect here. It may seem strange: according to common sense, computers are associated to deterministic computational procedures. Sort of complicated technology but clearly defined and behaving predictably. In reality they are complex, and therefore tricky machines, both from the practical and theoretical point of view. Well, we are already aware of the absence of a halting condition in our POLY program, this was not an error. It is a known imperfection, but imperfections (and errors) are useful in learning. The POLY "defect", that it doesn't stop itself, may turn out in an opportunity to reflect upon the machine behaviour. For instance, what it does actually mean that the computer (the turtle) is doing nothing? Or, what's going on when it appears to be stuck?

When we run POLY we saw something interesting: after a certain number of cycles, the figure was finished but the program was still running. The two things do not coincide. It is almost always easy to see when the figure is finished, starting from the time where the turtle begins to travel over lines that were already drawn. But try to hide the turtle, before starting POLY, by means of the HIDETURTLE command (same command in XLogo, hideturtle, and hideTurtle() in TigerYjthon). At the beginning the invisible turtle draws the figures and you see it clearly, but once the figures is completed, there is no way to see what the turtle is doing, since it is invisible! You know that is there, because you put the HIDETURTLE command, so you know what's going on, but imagine that you did not know it. What would you conclude? That the task is finished? Not

exactly. At first glance yes, but after a while you will notice something wrong. For instance, if you try to start the code again, you cannot do so and the reason is that the invisible turtle is still busy. If you want to start a new run you have first to stop the current one. It is exactly what we did by playing with POLY. But what is worthwhile to note here is that, once the figure is finished *the system appears to be stuck*. What we learn thus is that when we say that the computer is stuck it is actually working, only we don't see it. In our case the turtle is keeping doing something useless, we now it perfectly. But it is important to know that, when running a program, wrong conditions causing a similar condition are quite common, even with very famous (and expensive) software.

It is not to say that software designer are incapable, they are very good, generally. Instead, there are serious reasons for this state of affairs. A first issue is that software systems are extremely complex nowadays. It's often impossible to foresee all the potential problems. However the reality is even harsher: there is no way to build automatic testing methods (algorithms, technically speaking) for assessing the correctness of a program. This is a strong statement, in the mathematical sense, meaning that it has been demonstrated theoretically that there is no way to answer questions such as [Hromkovič, 2009, pag. 155]:

- Is a program correct? Does it fit the aim for which it was developed?
- Does a program avoid infinite computations (endless repetitions of a loop)?

We aren't computer scientists, nor software engineers, and only a very small number of our students will take one of these careers. The point here is the attitude towards these complex machines and technology in general. An attitude aware of the limits and of the unavoidable risks of errors or misbehaviours. A more watchful attitude towards technologies is crucial in education, nowadays, if we want to educate aware citizens instead of passive consumers.

For the curious — Trying to find a stopping condition for a (simple) program

Probably, at this point it is clear that, when ANGLE is a submultiple of 360, the figure is a regular polygon with N=360/ANGLE sides. Therefore, in these cases the number of repetitions in POLY needed to close the figure is N. But in all other cases? When ANGLE is not a submultiple of 360 or ANGLE $>$ 360?

Three years ago a primary teacher student sent me a letter — it was not an assignment — where, starting with the description of an exploration with Logo, written "as if I were a child", she ended up with a mathematical conjecture: is there a way to establish *a priori* how many iterations are needed to close these figures? Actually, what she was exploring, was the behaviour of a kind of POLY program. More precisely, she was playing with a POLY structure with the construction of a house inside it![a]

The solution to this problem comes from turtle geometry. It is based

on some theorems, where the fundamental one is a generalised version of the Turtle Total Trip Theorem we have seen before. It is the *Closed-Path Theorem* [Abelson and diSessa, 1980, pag. 24]:

Theorem 1 *The total turning along any closed path is an integer multiple of 360°.*

Where total turning is an intrinsic property of a path. It does not depend on where the path starts, or how it is oriented. The total turning of a path is determined by the integer that multiplies 360. That integer is called the *rotation number* of the path. It is interesting to try evaluating the rotation numbers for different paths. Here you have some examples:

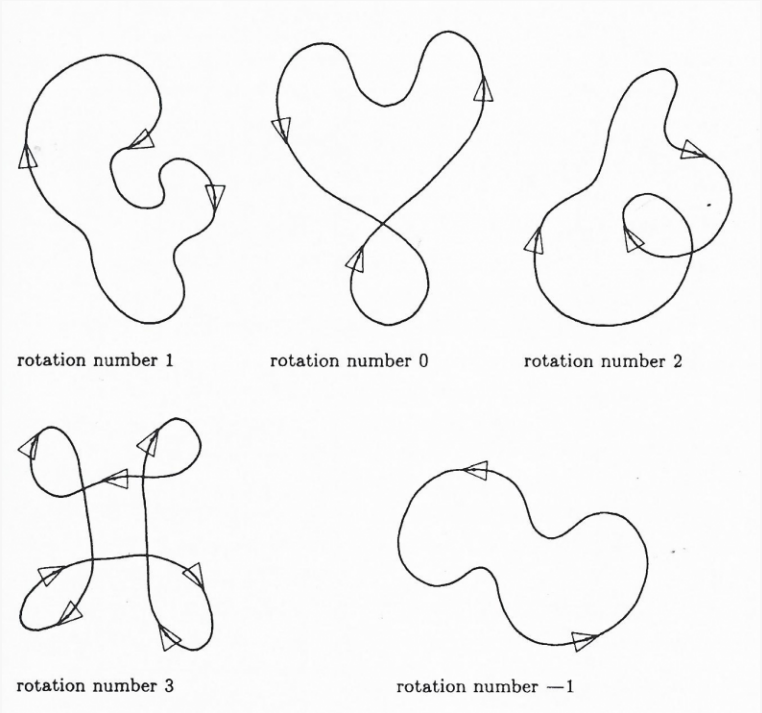


Figure 3.13: Close path examples and relative rotation numbers

[a] You can find the detailed story in my MOOC about coding: *https://federica.eu/l/martas_story_the_turtle_total_trip_theorem* — the MOOC is free: in order to access all the lessons you have to make just a free account.

The general outcome is that the number of cycles needed in POLY to close a figure is given by

$$n = \frac{\text{LCM(ANGLE,360)}}{\text{ANGLE}} \tag{3.6}$$

where ANGLE is the input angle to POLY and LCM(ANGLE,360) is the least common multiple between ANGLE and 360.

The exact derivation of this formula can be found in [Abelson and diSessa, 1980, pag. 24-32], the answer to Marta's conjecture in [Abelson and diSessa, 1980, pag. 32-36] and a synthetical derivation in the above mentioned MOOC lesson.

Thus, this was for the curious, but some of the readers could find material to propose some interesting explorations to somewhat older kids, here.

3.3.5 Successive approximations

We have seen that with POLY we can draw regular polygons and "stars". Let's write a version of POLY for drawing just polygons. At this point it's easy:

```
1 TO POLYG SIDE N
2    REPEAT N [
3       FORWARD SIDE
4       RIGHT 360/N
5    ]
6 END
```

Listing 3.27: A new version of POLY with SIDE and N, number of sides, as input parameters

This is a sort of "educated" POLY, which knows how to stop. This version is limited to regular polygons[7] but this is exactly what we want right now. In this version we have changed one input parameter: instead of ANGLE we have put N, the number of sides. Then, within the body of the program we calculate the turning angle as 360/N.

Now we can play with POLYG focusing on regular polygons. The game is simple: try drawing polygons with an increasing number of sides. Well, from time to time you will need to readjust SIDE, otherwise with large number of sides the figure will exceed the page. What will you see from a certain point?

Let's try, for instance, to superimpose the decagon (10 sides) with the icosagon (20 sides):

[7] It would be possible to write an "educated" general POLY program but we would need to use the result given in equation 3.6, therefore we should write the code for calculating the least common multiple.

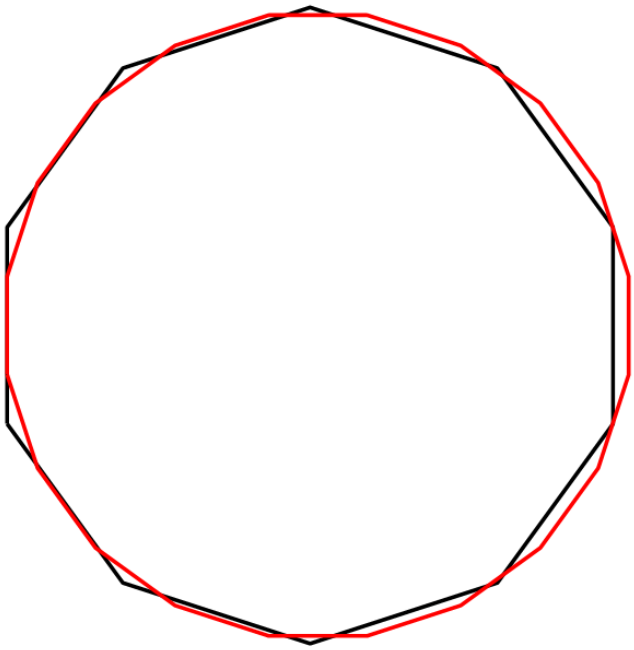


Figure 3.14: 10-sided and 20-sided polygons

Do you see how the icosagon is "more round" with respect to the decagon? Now let's compare the icosagon (20 side) with the triacontagon (30 sides).

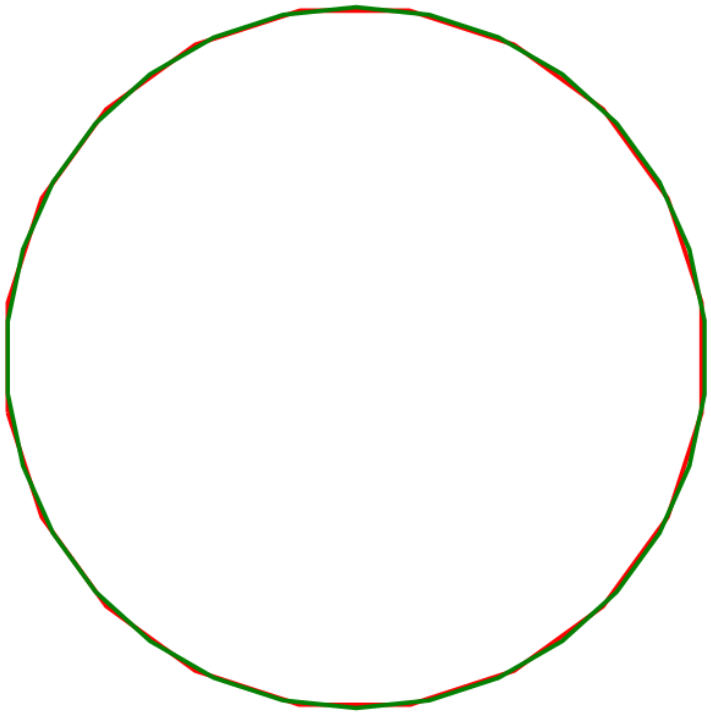


Figure 3.15: 20-sided and 30-sided polygons

Here we see that, in turn, the 30-sides polygon is "more round" than the 20-sides one. Now, let's exaggerate, comparing the 30-sides polygon with the 360-sides polygon — 360 1°turns!

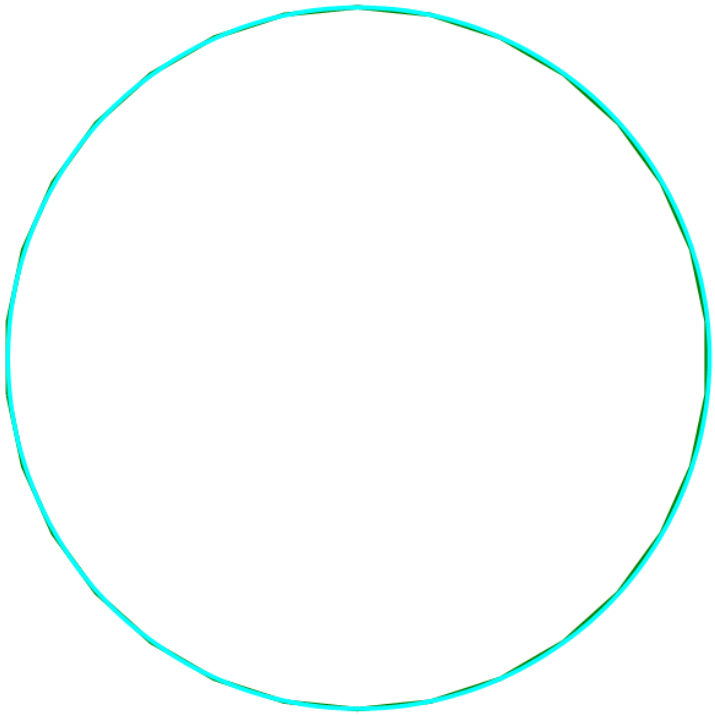


Figure 3.16: 30-sided and 360-sided polygons

The differences are minimal and, even if you zoom in deeply the image the 360-sided polygon appears to be "round". This happens because the length of the sides are comparable with the pixels, thus we cannot see them any more. Therefore, in Logo we can assume that a circle can be drawn by means of an instruction like REPEAT [FORWARD 1 RIGHT 1]. Recently, a smart 16 years old student told me promptly: "Prof. you are claiming we are drawing a circle that way, but it cannot be, it is still a polygon, even with many very small sides!". She was perfectly right. The discussion arised by this remark was extremely fruitful, leading to a crucial consideration: we can choose an extremely high number N to build an N-sided polygon, and this will be "rounder", even with respect to our last 360-sided polygon, but still, it will not be a true circle. Nobody's stopping us from picking an even bigger number to draw an even rounder polygon, but still a polygon! Thus, are we able to approach the circle as much as we want? Yes! Will we be able to reach the circle? No, unless we have the eternity! Or the mathematical concept of limit.

Powerful mathematical idea — Successive approximations

Considerations of this kind are the roots of reasoning for handling the infinite and the infinitesimal in mathematical analysis. Limits, derivatives, integrals, approximation by infinite series and other mathematical devices allows us to solve all the basic problems of science, by tackling the concept of infinite. The experience of approximating a circle, as we have shown, will turn out to be extremely useful for those students who will face a STEM path. By the way, once posed in rigorous formal terms, this is one of the possible definitions of a circle.

The idea of a thought process, composed by perfectly defined steps but "requiring eternity", falls in the realm of the considerations made in the note *a* on page 51. It is by means of the *limit* tool of mathematical analysis that infinity and infinitesimal may be "packaged" in finite and manageable expressions. Again, this is not about telling your students all this, but it is important to be aware of what is behind these Logo exercises.

The ability to manage the concept of infinity and infinitesimal is of crucial importance in all domain of science. Hromkovič, in his book *Algorithmic adventures* devotes an entire chapter to a brilliant description of the concept of infinity and why *infinity is infinitely important in computer science* [Hromkovič, 2009, pag. 73].

Chapter 4

Simulation: physics dynamic

4.1 Free-falling body

This chapter is aimed at those involved in learning or teaching physics at school, particularly classical mechanics. We are therefore no longer in primary school. Since the idea here is to reproduce a sort of virtual physics laboratory, we will introduce some new commands, in addition to the basic Logo command, to better focus on the relevant concepts.

Let's consider a free-falling body in proximity of earth surface. When you got into contact with these issues, you were probably told that there is a formula, probably you have been told that there is a formula describing the motion of the falling body, which can be written as

$$y(t) = y_0 + v_0 t + \frac{1}{2} g t^2 \tag{4.1}$$

where y is the position of the body at each time instant t, y_0 is its initial position, that is when we open the hand to let it fall, v_0 is its initial velocity, 0 if you just let the body fall, and g is the gravity acceleration constant in proximity of earth surface, its value being 9.8 m/s^2. Depending on the curriculum, the teacher method and your age, you may be told this formula derives from the second principle of dynamics, which states that

$$\mathbf{F} = m\mathbf{a} \tag{4.2}$$

where $\mathbf{F}$ is the force applied to the body (gravity in this example), m is the body mass and a its acceleration[1]. To understand this derivation you have to master the basic tools of mathematical analysis, i.e. integration in this case. Most probably you learned these formulas by heart, in order to be able to solve the exercises and pass the exams. Which, unfortunately, is not enough for the

[1] Later on in your studies, you will learn that equation 4.2 is a differential equation of the second order, since it's involving variations of variations: acceleration is the variation of velocity whereas velocity is the variation of space. This is written as $\frac{d}{dt}\frac{dy}{dt} = mg$. By integrating this equation two times with respect to time we obtain equation 4.1.

thorough and substantial understanding of the matters. Logo can help us, let's see how.

In the following figure, on the left, we show the relevant part of Logo code for simulating the free fall of a body, on the right you have the output of the program, where the position of the body at successive equal interval of times is drawn with a small green circle.

```
1
2  YPOS = 0.0 ; initial position
3  VEL = 0.0 ; initial velocity
4  ACC = 9.8 ; constant acceleration
5  DT = 0.5   ; time interval
6
7  WHILE YPOS < 500 [
8    VEL = VEL + ACC*DT ; update velocity
       for next point
9    YPOS = YPOS + VEL*DT ; update position
       for next point
10   PENDOWN
11   CIRCLE 5
12   PENUP
13   FORWARD VEL*DT
14   ]
```

Listing 4.1: Logo code for a free falling body experiment

There are a couple of new commands here. First we made a loop by means of a WHILE instruction instead of a REPEAT one. In particular we used

WHILE YPOS < 400

When using a REPEAT we have to specify a certain number of repetitions, for instance REPEAT 10. Here we have the expression YPOS $<$ 400 instead of the number of repetitions. What does it mean? The expression is a so-called *condition*: the WHILE command will repeat the sequence of instructions between the square brackets, [...], until the *condition* is satisfied, i.e., until the value of the variable YPOS is less then 400. The first time that YPOS will result to be greater or equal than 400 the loop the turtle will get out of the loop, executing the following instructions, eventually. The second new command is CIRCLE. This is a ready-made LibreLogo command: CIRCLE R, where R is

the radius of the circle. We used it just for brevity. In LibreLogo there are such ready commands for the most common figures. We did not used them so far because for the first basic explorations of turtle geometry it is better to do things "by hand". Nobody's stopping us from building our own MYCIRCLE R program, starting from our basic REPEAT 360 [FORWARD 1 RIGHT 1] code — it could be a good exercise. Here we go straightforward to the physics concepts but before discussing them we suggest to try the Python version. We used here the ABZ's TigerJython Python environment[2] Python is more suited for the next programs and it is also perfectly adequate for students facing the first physics issues.

```
1  from gturtle import *
2
3  makeTurtle()
4  clearScreen()
5
6  setPos(0,200)
7  setHeading(180)
8
9  YPOS = 0.0  # initial position
10 VEL = 0.0   # initial velocity
11 ACC = 9.8   # constant acceleration
12 DT = 0.5    # time interval
13
14 while YPOS < 400:
15     VEL = VEL + ACC*DT      # update
       velocity for next step
16     YPOS = YPOS + VEL*DT    # update
       position for next step
17     penDown()
18     dot(5)
19     penUp()
20     forward(VEL*DT)
```

Listing 4.2: Python code for a free falling body experiment. Steps are increasing because of uniform acceleration.

Let's comment this program in detail. Instruction N. 1 imports all the resources needed by Python to manage a turtle. It's a common praxis in Python, so as to load the necessary resources only. Instruction N. 3, makeTurtle(), creates a so called turtle instance, in substance a new turtle listening to your commands. By means of instructions N. 6-7 we send the turtle at the top of the page to drop it on the ground, at the bottom of the page. That's trivial. Instead, instructions 9-10 are important because they set the so-called *initial conditions*. This is the first crucial step for solving any physics problem. In

[2] Available at *https : //webtigerjython.ethz.ch/*

the algebraic formulation of the problem, namely in equation 4.1, the initial conditions consists in the values of the initial position y_0 and initial velocity v_0. You will realize (perhaps) the meaning of the values when you will have to solve some problems but here it is more direct: you have to decide at once where to place the turtle and which initial velocity you have to assign to it. Not only that, you have also to decide which unit of measurement you should use. In other words, you have to build a thorough knowledge of the concept by using it at once. Similarly, you have to fetch the correct value for the acceleration of gravity in instruction N. 11.

Powerful concept in physics — Initial conditions

The setting of initial conditions is the first essential step needed to solve any problem in physics. Initial conditions are embedded in formulas taught in secondary schools but, at this age, the capability of understanding all the implications which are synthesised in formulas is rare — building the mental device for understanding mathematical formulas requires time, especially if physical meaning is involved. The problem is not trivial and not a novel one: a famous physicist, Enrico Persico, master of Enrico Fermi, wrote an interesting paper in 1956 [Persico, 1956], wondering about the good formal preparation of some students but their poor comprehensions of the core matters:

> Why does this girl, who is not stupid, but who finds it so difficult to describe a capacitor, once put on writing formulas [Maxwell equations], runs like a locomotive?

The computational formulation of physical problems favours the dynamic perception of physical phenomena, with respect to the conventional algebraic formulation, as it has pointed out in a very thoughtful paper by Sherin [Sherin, 2001]. It's not about dropping algebraic description of physical phenomena. Instead, it's about integrating both approaches.

In instruction N. 12 we set the *time interval*. This is a necessary parameter because continuous quantities become discrete, when solving problems numerically. That is, when using time, we have to divide it in many equispaced intervals, and refer to all quantities — position, velocity... — evaluating them once for each time interval, for instance every second. The length of the interval has to be chosen by means of a trade-off: many short intervals describe better what's going on but the computation and other burdens may turn out to be too high.

Powerful concept in physics — Computational vs algebraic approach

When facing problems from the numerical point of view, students are obliged to enter the anatomy of phenomena, once again. Appreciating the numerical face of problems, besides the algebraic one, is essential to shape adequately the scientific culture of a student. Nowadays, the computational face of all sciences is equally important with respect to the classical descriptions. Moreover, entire new crucial fields are totally numerical. All technologies are based on numerical solutions of mathematical problems.

For instance, ubiquitous computed tomography images represent the numerical solution of the tomographic problem, based on the inversion of the Radon Transform. Technologies are based on a fine blend of analytical formulations and numerical methods. It is important to give students the possibility to dive into numerical exploration.

Instructions N. 14-20 solve the problem. What does it mean "solving a problem" here? From the classical point of view, it means to find the function $y(t)$, given the physical circumstances. Equation 4.1 is the solution of problem 4.2, given the acceleration g and the initial conditions y_0 and v_0. In the numerical way, solving the problem means to calculate the value of the position YPOS for a sufficient number of time points and, in our case, representing them graphically. In the numerical way the phenomenon is simulated, actually. Instruction N. 14 initiates a loop which will last until the position YPOS will keep being less than 400. For each cycle, first the velocity for the next time point is evaluated with VEL = VEL + ACC * DT (instr. N. 15), based on the definition of acceleration. Once we have the new value of VEL, we can find out the new value of position: YPOS = YPOS + VEL * DT, according to definition of velocity (instr. N. 16). Finally, dot(5) draws a point in the current position and forward(VEL*DT) send the turtle to the next point. The figure shows the expected uniformly accelerated motion.

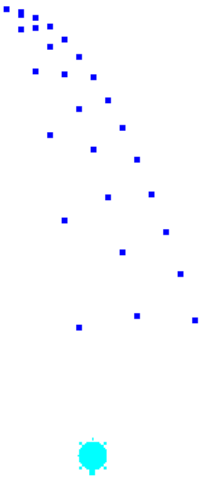
The reader is invited to play with this code, trying for instance to calculate more data points or to change the acceleration, i.e. who would the body fall on the Moon? And on Jupiter?

4.2 Free-falling body with constant acceleration and horizontal velocity component

In this and the following examples we are going to show some variations on the previous case. We leave to the reader the pleasure to discover the differences. We will only quote the added effects.

Here we show the same free-falling body situation but with a constant horizontal velocity component. Three trajectories are shown, for three different values of the gravity acceleration constant. This physics exercise allows to see how a mathematical objects — the parabola — stems from a natural phenomenon. It is interesting to explore the effect of the a parameter in the parabola equation $ax^2 + bx + c = 0$, which in this context is equal to the acceleration gravity constant G.

```
from gturtle import *
from math import *

makeTurtle()
clearScreen()


def PARAB(ACC):
    setPos(0,200)
    setHeading(180)

    YPOS = 0.0
    VEL = 0.0
    DT = 1.

    while(YPOS < 500):
        VEL = VEL + ACC * DT
        YPOS = YPOS + VEL * DT
        penDown()
        dot(5)
        penUp()
        forward((VEL/2-1) * DT)
        left(90)
        forward(10 * DT)
        right(90)

PARAB(5)
PARAB(9.8)
PARAB(30)
```



Listing 4.3: Free-falling body with a constant horizontal velocity component. Three trajectories are shown, for three different values of the gravity acceleration constant.

4.3 Free-falling body with air resistance

Here we have an air resistance component, RES, which we express by means of a force proportional to the velocity VEL by means of constant KA. The acceleration, or better said, the deceleration caused by this force will be RES/M, where M is the body mass. Try for instance to simulate a thicker atmosphere...

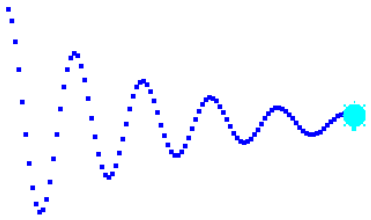
```
from gturtle import *

makeTurtle()
clearScreen()

setPos(0,200)
setHeading(180)

YPOS = 0.0
VEL = 0.0
ACC = 9.8
DT = 0.5
RES = 0.0

G = 9.8
M = 10.0
KA = 2.0

while YPOS < 350:
    RES = VEL * KA
    ACC = G - RES / M
    VEL = VEL + ACC*DT
    YPOS = YPOS + VEL*DT
    penDown()
    dot(5)
    penUp()
    forward(VEL*DT)
```

Listing 4.4: Python code for a free falling body experiment with air resistance. Steps are becoming constant because the balancing between gravity and air resistance.

4.4 Body hanging from spring

We maintain here the air resistance component proportional to body velocity, but we attach it to a spring. This effect is achieved by adding an elastic force component, proportional to the displacement with respect to the equilibrium position. The intensity of this force is given by constant K. In this version we also put a horizontal displacement to obtain the effect of plotting the motion with respect to time. For simulating just the motion you can set as comment the last three instructions, by placing a # character at the beginning of the lines.

```
from gturtle import *
from math import *

makeTurtle()
clearScreen()

setPos(0,200)
setHeading(180)

YPOS = 0.0
VEL = 0.0
ACC = 9.8
DT = 1.0
RES = 0.0

G = 9.8
M = 10.0
KA = 0.5
K = 1.0

repeat(100):
    RES = KA * VEL
    SPR = K * YPOS
    ACC = G - RES / M - SPR / M
    VEL = VEL + ACC * DT
    YPOS = YPOS + VEL * DT
    penDown()
    dot(5)
    penUp()
    forward(VEL * DT)
    # set as comments the following
    instructions
    # if want to see oscillating along y
    -axis
    left(90)
    forward(3*DT)
    right(90)
```

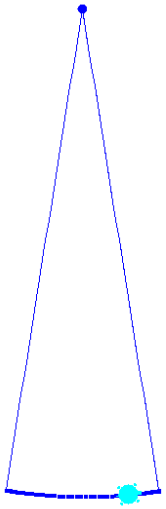


Listing 4.5: Python code for a body hanging from a spring in presence of air resistance.

4.5 Pendulum

Here a little bit of trigonometry is needed. There is something to improve in this code but still a good starting point...

```
from gturtle import *
from math import *

makeTurtle()
clearScreen()

penUp()
forward(290)
right(180)
penDown()

# physical context

G = 9.8/1000/2.857  # constant acceleration
                    # expressed in points/sec^2
L = 500             # Pendulum length

# Where are we starting?

PHY = -pi/20        # initial angle

# Whith which velocity?

OMEGA = 0.0         # initial angular velocity
DT = 1.0            # integration time interval

# Position the turtle in an appropriate location

dot(10)
setHeading((pi + PHY)/pi*180)
forward(L)
back(L)
setHeading((pi - PHY)/pi*180)
forward(L)
penDown()

repeat(200):

# how much do I have to turn the moving foot?

    setHeading((pi/2-PHY)/pi*180)

    OMEGA = OMEGA - G*sin(PHY) * DT
    PHY = PHY + OMEGA * DT

# mark current location

    penDown()
    dot(5)
    penUp()

# do next step

    forward(OMEGA*L * DT)
```



Listing 4.6: Basic code for the simulation of a rigid pendulum.

4.6 Body free-falling from space

This is the same case of the free-falling body, except that we have to drop the assumption of constant gravity acceleration, because as long as we move away from Earth the gravity force is fading. This means that we have to use explicitly Newton gravity law. Are you able to find out in which line of code we are using Newton's law?

```
from gturtle import *

makeTurtle()
clearScreen()

setPos(0,-200)
setFillColor("blue")
dot(70)
setFillColor("lime")
XPOS0 = getX()
YPOS0 = getY()

setPos(0,200)
setHeading(180)
hideTurtle()

XPOS = getX()
YPOS = getY()
VEL = 0.0
DT = 2.0
KG = 9800

repeat(500):
    DY =  YPOS-YPOS0
    print(DY)
    if(DY != 0.):
        VEL = VEL + KG / DY**2 * DT
    YPOS = YPOS - VEL * DT
    penDown()
    dot(5)
    if(YPOS-YPOS0 < 10):
        break
    penUp()
    moveTo(XPOS,YPOS)

setPenColor("red")
dot(20)
```

Listing 4.7: Body free-falling from space

4.7 Calculating the orbit of Halley's comet...

To tell the truth, I hesitated to add this section, being afraid of scaring the reader. Actually, it is somewhat outside the scope of this text but, I believe, it could also be fascinating.

In order to adhere to Papert's statement about educational programming languages that should have both a "low floor and high ceiling", with this example we show that even rather complex tasks can be done with these educational programming languages. Therefore, among the downloadable software, you find

also a LibreLogo version of this program. Here we are going to comment the Python version[3].

But another reason to go through this example is to see how similar are the structures of the simple algorithm we have seen for drawing a circle and that of drawing the orbit of a celestial body, despite the remarkable difference in complexity. The reason of such similarity stays in the same differential local nature in which the two problems are posed. We show this fact by sketching the two algorithms with the so-called *pseudo code*, which is not an executable code but one that allows to grasp the essence of the algorithms:

Data: (small) step length and (small) turning angle
Result: plot of the circle
1 go to starting point;
2 **while** *circle is not closed* **do**
3 | step forward;
4 | turn right;
5 | plot the step;
6 **end**

Algorithm 1: Drawing a circle: REPEAT [FORWARD 1 RIGHT 1]

Data: astronomical data about Halley's comet
Result: plot of the orbit
1 initialization;
2 **while** *orbit is not closed* **do**
3 | calculate acceleration in current point;
4 | given this acceleration value, calculate velocity;
5 | given this velocity value, calculate position of next point;
6 | move to next point;
7 | plot the step;
8 **end**

Algorithm 2: Calculating Halley's comet orbit

[3] A note about the expression "program" we used throughout the listing. In all the "Einfach Informatik" volumes published by ABZ, for instance in "Einfach Informatik — Programmieren — Sekundarstufe I"' [Hromkovič and Kohn, 2018a], the word "command" (*Befehl*) is used for designing sequences of instructions which can be invoked by means of a single name for being executed altogether. This choice makes sense since, when starting with Logo, it is very appropriate to give the kids the idea of building their own self-made commands, because with the turtle these encapsulated pieces of code are usually new commands for the turtle. However, in this case it is somewhat weird to call "command" the instructions used to make an interpolation, for instance. Hromkovič and Tobias [Hromkovič and Kohn, 2018b, pag. 45] raise the point about the variety of names which have been used to call these objects, for instance "routines", "procedures", "subprograms", "subprograms", "methods", "functions". They do not mean exactly the same things, depending on historical context and on some specific behaviour. For instance, in Python jargon they are usually called "functions". However, there is also the tendency to assume that "functions" calculate a value and renders it. Well... in this listing we used the term "subprograms".

The previous example of a body free-falling from space is opening the way to the evaluation of bodies orbits in space. Adding an initial horizontal velocity component lend us to the simulation of orbits. Let us sketch the problem in the context of Newton's gravity, first in algebraic notation and, of course, limiting us to the two-dimensional problem. As far as the algebraic approach is concerned, here we just pose the problem. The analytic solution of the differential equations is a higher education level problem. However the algebraic approach serves as starting point for the programmed numerical solution.

To solve the problem we have to start from the second principle of dynamics

$$\mathbf{F} = m\mathbf{a} \tag{4.3}$$

where the force is given by Newton's law

$$\mathbf{F} = -\frac{GMm}{r^2}\frac{\mathbf{r}}{r}. \tag{4.4}$$

G is the universal gravitational constant, M is the mass of the largest of the two bodies, m the smaller one, r is the distance between the two masses, between the barycenters of the two bodies to be precise. Bold symbols are vectors, the others are scalars. First we have to derive the acceleration of the smaller body — the larger one will stay still; assuming that its mass is negligible with respect to that of the sun, i.e. $M \gg m$

$$\mathbf{a} = -\frac{GM}{r^2}\frac{\mathbf{r}}{r}, \tag{4.5}$$

and then one should solve the differential equation

$$\frac{d}{dt}\frac{d\mathbf{r}}{dt} = -\frac{GM}{r^2}\frac{\mathbf{r}}{r}. \tag{4.6}$$

Of course, this goes beyond our aims here. However we are going to use this relation as a basis for the numerical approach. Let' see the listing.

```
1  # Halley-RK-4-AU-90-sharable.py
2
3  # Copyright 2020 Andreas Robert Formiconi (arf@unifi.it)
4
5  # Program distributed under the terms of the GNU
6  # General Public License
7
8  # This program is Free Software: it can be redistributed
9  # and modified under the terms of the GNU General Public
10 # License published by Free Software Foundation, in version 3
11 # or one of the following ones. The text of the license is
12 # accessible in
13 # <https://www.gnu.org/licenses/licenses.en.html>.
14
15 # Calculation of the orbit of a celestial body around the sun
16 # by  numeric integration of the motion equations by Newton
17 # gravitation law. The problem is posed in two dimensions and
18 # assumes that there are no perturbations from other bodies.
19 # The code is adjusted to solve the case of a highly eccentric
20 # orbit like that of Halley's comet.
```

```
21
22 # The turtle plays the role of the comet. The sun is at the
23 # center, which in TigerJython has coordinates (0,0), where
24 # the unit of measurement is the "point" (p).
25 # Warning: if you run this code as it is, be patient: the
26 # trajectory takes a minute or so to appear, on the right,
27 # since a lot of points have to be calculated, in order to
28 # achieve reasonable accuracy...
29
30 from gturtle import *
31
32 makeTurtle()
33 clearScreen()
34 showTurtle()
35
36 # Global variables, i.e., variables that are "visible" in both
37 # the main program and within each subprogram, such as
38 # NEWTON, STP and so on.
39
40 # global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS,
41 # XVEL, YVEL, XACC, YACC
42
43 # In the following, subprograms in which specific functions
44 # have been encapsulated: NEWTON calculates the acceleration
45 # at a given point, STP evaluates the next step, WRITEPOINT
46 # plots the point as long as they are calculated
47
48 # ************************************************
49 # Subprogram NEWTON: calculates the acceleration at point of
50 # coordinates X, Y returning two acceleration components
51 # ACCX and ACCY
52
53 def NEWTON(X,Y):
54     global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS, XVEL, YVEL
      , XACC, YACC
55     DX = (X-XPOS0)
56     DY = (Y-YPOS0)
57     R2 = (DX**2 + DY**2)
58     R = sqrt(R2)
59     XACC = - GG / R2 * DX / R
60     YACC = - GG / R2 * DY / R
61
62
63 # ************************************************
64 # Subprogram STP: calculates the next step with the
65 # Runge-Kutta interpolation of the fourth order.  This
66 # interpolation represents a fairly sophisticated way to
67 # calculate the trajectory points. It is needed to reduce
68 # the approximation errors inherent to the calculation of a
69 # continuous function in a discrete set of points. The
70 # algorithm is taken from W.H. Press et al, Numerical
71 # Recipes - The Art of Scientific Computing, Cambridge
72 # University Press, 1992, pp. 704-708.  The coordinates
73 # of the new position, XPOS and YPOS, and the speed at
74 # that point, XVEL and YVEL, are returned.
75
76 def STP():
77     global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS, XVEL, YVEL
      , XACC, YACC
78
```

```
79      NEWTON(XPOS, YPOS)
80      KX1 = Dt * XACC
81      XVEL1 = XVEL +KX1 / 2.
82      KY1 = Dt * YACC
83      YVEL1 = YVEL +KY1 / 2.
84      XPOST = XPOS +XVEL1 * Dt / 2.
85      YPOST = YPOS +YVEL1 * Dt / 2.
86
87      NEWTON(XPOST, YPOST)
88      KX2 = Dt * XACC
89      XVEL2 = XVEL +KX2
90      KY2 = Dt * YACC
91      YVEL2= YVEL +KY2
92      XPOST= XPOS +XVEL2 * Dt / 2.
93      YPOST= YPOS +YVEL2 * Dt / 2.
94
95      NEWTON(XPOST, YPOST)
96      KX3 = Dt * XACC
97      XVEL3 = XVEL +KX3
98      KY3 = Dt * YACC
99      YVEL3= YVEL +KY3
100     XPOST= XPOS +XVEL3 * Dt / 2.
101     YPOST= YPOS +YVEL3 * Dt / 2.
102
103     NEWTON(XPOST, YPOST)
104     KX4 = Dt * XACC
105     XVEL4 = XVEL + KX4
106     KY4 = Dt * YACC
107     YVEL4 = YVEL +KY4
108
109     XVEL= XVEL + (KX1 + 2 * KX2 + 2 * KX3 + KX4) / 6.
110     YVEL= YVEL + (KY1 + 2 * KY2 + 2 * KY3 + KY4) / 6.
111
112     XPOS= XPOS + (XVEL1 + 2 * XVEL2 + 2 * XVEL3 + XVEL4) * Dt /
         6.
113     YPOS= YPOS + (YVEL1 + 2 * YVEL2 + 2 * YVEL3 + YVEL4) * Dt /
         6.
114
115
116 # ************************************************
117 # Subprogram WRITEPOINT. It does two things:
118
119 # 1) it sends the turtle to the next point of the trajectory
120 # by means of a POSITION instruction - that's how the
121 # the orbit is drawn.
122
123 # 2) it writes in a file all the relevant values of each point
124 # of the trajectory: two position, speed and acceleration
125 # components - that is the solution of the motion problem.
126 # These data can be used successively for creating graphical
127 # representations with other software or for further
128 # processing.
129
130 def WRITEPOINT():
131     global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS, XVEL, YVEL
        , XACC, YACC
132
133     moveTo(XPOS, YPOS)
134
135 # ************************************************
```

```
# ************************************************
# Main program

# First of all, the physical constants involved are calculated
# in the M.K.S. system (Meter, Kilogram, Second), with the
# exception of the distances of the orbits because, given the
# enormous values at stake, the AU (Astronomical Unit) is
# more handy, where 1 AU = 1.495978707 x 10^11 meters.
# One AU corresponds to the average distance between the
# earth and the sun. Thus, for example, when it is found that
# Halley's aphelion is about 35.08 AU, it means that the
# maximum distance of the comet from the sun is equal to
# about 35 times the distance between the earth and the sun.
# In the end, however, all distance measurements are
# transformed into "points" for the purposes of the graphic
# representation.

G = 6.67E-11             # (N*m^2/Kg^2)  Gravitation constant
Ms = 1.99E30             # (Kg) Mass of the sun

Dp = 200.0               # Aphelion expressed in points
rAf = 35.08              # Aphelion (AU)
Dt = 0.001               # Integration interval (time)

K = Dp/rAf               # Scaling factor: number of points/AU
GAU = G / 1.496E11**2    # Gravitation constant expressed in AU
Gp = GAU * K**2          # (N*p^2*Kg^2)
GG = Gp * Ms             # Gravitation constant inclusive of the
                         # solar mass (to reduce the number of
                         # multiplications in cycles)
eps = 0.967              # Halley's comet orbit eccentricity

clearScreen()

setFillColor("yellow")  # sun color
setPenColor("black")
dot(5)                   # Given the size of the comet's orbit
                         # the sun cannot be in scale

# These are the coordinates of the center of the page, which
# we assume to be at the origin of the reference system and
# that we make coincident with sun position.

XPOS0 = getX()          # origin coordinates (centre of page)
YPOS0 = getY()

  hideTurtle()           # better hiding the turtle: too slow
    ...

# Determining initial conditions

# Let us place the comet at its initial position

setPos(XPOS0 + rAf*K, YPOS0)

XPOS = getX()
YPOS = YPOS0

# Initial velocity of the body, which is given by II Kepler law
    ,
```

```
194 # once the eccentricity and the solar mass are given
195
196 XVEL = 0.0
197 YVEL = sqrt(GG/(rAf*K)*(1-eps))
198
199 setPenColor("blue")
200 setPenWidth(1)
201
202 # Here begins the cycle for drawing the trajectory.
203 # The points of the trajectory are calculated via subprogram
204 # STP, then they are drawn by the WRITEPOINT subprogram,
205 # which takes also care of writing data (position, speed,
206 # acceleration) in a file. However, WRITEPOINT is invoked
207 # only once a while, since the trajectory is calculated in a
208 # very large number of points, in order to evaluate the path
209 # with reasonable accuracy and this number would be far to
210 # large for drawing purposes.
211 # This implementation, which is adjusted to reproduce
212 # strongly eccentric orbits, like that of Halley's comet,
213 # uses only one point out of 10000 for drawing the path.
214 # This feature is controlled by means of the nWrite counter.
215 # A flag (yes/no variable), yIsNegative, is then used to
216 # check that the orbit will be drawn only once.
217
218 nWrite = 0
219 yIsNegative = False    # Flag one orbit only
220
221 while not ( yIsNegative and (YPOS-YPOS0) > 0):
222     nWrite = nWrite + 1
223     if not yIsNegative and (YPOS-YPOS0) < 0:
224         yIsNegative = True
225     STP()
226     if nWrite == 1:
227         WRITEPOINT()
228     if nWrite == 10000:
229         nWrite = 0
230
231
232 print("Done!")
```

Listing 4.8: Calculating the orbit of Halley's comet

Here you have the result, as plot on the TigerYjthon graphic output window. Of course, you could also print the point coordinates or save them on a file for further elaborations or other graphic representations. The point on the left is the sun.

Let's go on through the program listing. There are three subprograms: NEWTON (Inst. 53-60) for calculating acceleration, based on equation 4.4, STP (Instr. 76-113) for calculating position (XPOS, Ypos) and velocity (XVEL, YVEL) in the next point[4], WRITEPOINT moves the turtle to the next point

[4]In this program we use a particular interpolation to reduce the approximation errors inherent to the calculation of a continuous function in a discrete set of points. We use here the Runge-Kutta interpolation of the fourth order [Press et al., 2007, pagg. 704-708]. We do

drawing the path at the same time. Let's rewrite here the pseudo code of page 87 to show the role of the different subprograms:

```
   Data: astronomical data about Halley's comet
   Result: plot of the orbit
1  initialization;
2  while orbit is not closed do
3  |  calculate acceleration in current point (NEWTON);
4  |  given this acceleration value, calculate velocity (STP);
5  |  given this velocity value, calculate position of next point (STP);
6  |  move to next point (WRITEPOINT);
7  |  plot the step (WRITEPOINT)
8  end
```

Algorithm 3: Calculating the orbit of Halley's comet

The main program begins at Inst. 137. Instructions 153-166 set the problem data: sun mass (or whatever), aphelion (largest distance from sun) and eccentricity of orbits, integration time interval. The initial conditions are set between instructions 184 and 197. The turtle (comet) is started at the aphelion, at the far right in the figure, where the velocity vector is vertical and directed upwards and vertical[5]. As far as the velocity module is concerned, it can be determined by Kepler II law and ellipse properties:

$$v = \sqrt{\frac{GM}{r_a}(1-\epsilon)} \tag{4.7}$$

$$r_a = a(1+\epsilon) \tag{4.8}$$

where a is the length of the semi-major axis, ϵ is the eccentricity of ellipse and r_a is the aphelion.

Finally, the core of the main program between instructions 221 and 229. It's just a loop for calling STP anf WRITEPOINT, with some controls to stop once the orbit is closed.

By changing the appropriate astronomical data, this program can be used to simulate orbits for other two-bodies systems, provided the mass of the smaller one is negligible with respect to the other. It could be interesting to look for such data and experiment with the code.

---

not need to enter into the details of this interpolation here.

[5]This holds true for the TigerYjthon environment, where the Cartesian origin is in the center of the image space, x axis is positive towards right and y axis towards the top. In the case of LibreLogo, the same condition holds except for the y axis which is positive towards the bottom of the page.

Chapter 5

Simulation: behaviour

5.1 Is there a place for randomness in computers? The Turtle plays the turtle...

So far we have depicted a sort of deterministic vision of computer programming. In our context this means giving the Turtle clear commands - do this, do that. Once the program is written the game is over. No matter the complexity, the drawing is frozen in the code. Thus, is there no place for randomness in the computer?

Yes and no. A detailed technical explanation of the response would be too complex here. In a first approximation we can say that, no, a computer cannot produce true randomness but a sort of pseudo-randomness, thanks to appropriate mathematical tricks.

Basically, in order to produce randomness one has to be able to generate random numbers, by means of so called random number generators. In reality, they generate periodic sequences, in the sense that after a certain number of random extractions, the same initial sequence is started again. The trick consists in using algorithms that produce extremely large periods so that one never reaches the end of the sequence by means of successive number extractions.

Thus, they appear as true random numbers. In LibreLogo the Turtle understands the RANDOM command:

- X = RANDOM 100 ; gives back a random float number[1] ($0 \leq X < 100$), that is equal or greater than 0 and smaller than 100.
- X = RANDOM “abcde” ; gives back a random letter among a, b, c, d, e
- X = RANDOM [1, 2, 3] ; gives back a random element among 1, 2 and 3

You can also mix different items, for instance

X = RANDOM [1, “pippo”, 3.14]

[1] In computer science a float number is a number with decimal digits. For instance, 3.14 is a float number, 18 is an integer number.

But randomness is also embedded in the special default variable ANY, for instance with

PENCOLOR ANY

you are going to use a random pen color.

Let' take our original program POLY (pag. 59) again, and change it by choosing randomly direction of steps and turning angles, instead passing them as parameters:

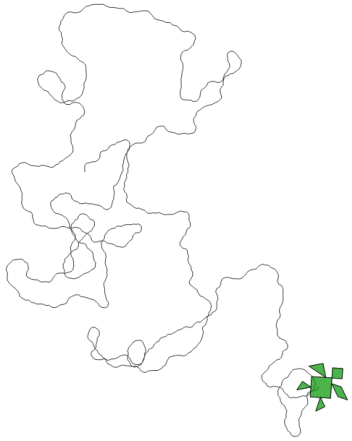
```
TO RAND
   REPEAT [
      FORWARD RANDOM(10)
      RIGHT RANDOM(360)
   ]
END
```

Listing 5.1: POLY in random version

The structure is the same of POLY, apart the absence of parameters, but the behaviour is totally different! What's doing here the Turtle? Well, the Turtle is doing the turtle, even if a bit crazy one. It's a game, it's a simulation. You can play with it. In the previous code the steps are randomly chosen between 0 and 10, whereas the turning angles between 0°and 360°. These values make the Turtle's behaviour hectic, let's give it a tranquillizer...

```
TO RAND
   REPEAT [
      FORWARD RANDOM(1) + 1
      RIGHT RANDOM(90) - 45
   ]
END
```

Listing 5.2: RAND with different random choices



With FORWARD RANDOM(1) + 1 we are choosing step lengths between 1 and 2. With RIGHT RANDOM(90) - 45 we are restricting the deviations to angles comprised between -45°and 45°— after all, who have seen a turtle turning by 180°all of a sudden?

You can try the same code but I could easily bet that your turtle didn't travelled exactly along the same pattern as mine. To tell the truth, the probability that you get the same picture is not exactly zero but it is extremely low: I'm pretty safe...[2].

Powerful concept — Randomness in science

So what? How can we be happy of producing unpredictable results by means of a machine conceived for making complex and accurate calculations? Well, actually, we are very glad because with this simple code we open a window on the huge and crucial world of scientific simulation. Nowadays, in every scientific field the simulation is an essential investigation tool, the only one possible to face the overwhelming complexity of natural phenomena.

But simulations are not restricted to the domain of physics, on the contrary, the more complex the phenomena, the more simulations may turn out to be the unique possible investigation tool. In the context of biological disciplines we talk about *in silico* experiments, referring to the fact that they are realized through calculations done with digital computers that run on silicon chips - the expression is an allusion to the Latin phrases *in vivo*, *in vitro* and *in situ*, commonly used in biology.

Nowadays *in silico* experiments include molecular biology, genetic assays, tumor growth, dermatology, bone remodelling, organ failure, clinical trials, just to mention a few. In medicine, for instance, *in silico* studies are used to discover new drugs because it is faster and costs much less. In

[2]Further considerations and examples can be found in lesson 7 of the MOOC *https* : *//federica.eu/l/the_turtle_does_the_turtle*

biology they are used to study to formulate behaviour model of cells and in genetics to analyse gene expression (In molecular biology "gene expression" means the way a set of genes determines the functioning of the cell at the macromolecular level).

Beyond the realm of simulations, the understanding of the role of randomness in nature is one of the most relevant achievement of science. Mathematicians have been building the edifice of statistics since the 18th century, the branch of mathematics needed to manage randomness. During the 20th century randomness bursts into several fields of physics, for instance in statistical mechanics to relate macroscopic properties of gases with their microscopical particle nature, in quantum mechanics to describe the behaviour of subatomic particles, in nonlinear dynamics to describe chaotic degeneration of deterministic systems. The overwhelming complexity peculiar of biological, atmospheric and social systems makes the dealing with randomness an everyday business in these fields. The blending of technologies with social life drives us in the area of data mining. Last but by no way least, computer scientists find in randomness a terrific tool for managing problems otherwise unsolvable [Hromkovič, 2009, pag. 201].

The philosopher and sociologist Edgar Morin describes human condition as about *navigating in an ocean of uncertainty through archipelagos of certainty* [Morin, 1999]. Generally, school curricula do very little to give a correct perspective of our contemporary knowledge of the world, where the uncertainty plays a crucial role. Therefore, the scientific vision which is given is skewed towards a falsely deterministic description dominated by one-answer-only problems. Which is wrong.

5.2 Shaping the environment

Let's provide some context to our simulated turtle. In the previous examples, the Turtle was wandering around the page with no notion about the environment. Now, we are willing to let the turtle walk in an enclosed space. So we need a shape for the allowed space and a behavior to determine the turtle's actions once it has reached the boundary. Let's switch to TigerYjthon which is more appropriate for the forthcoming considerations. Here we are going to use the "function" name instead the "new command" one we used when talking about Logo. First of all we try to create the space, for instance a circular one. We make it green, turtles enjoy grass.

```
from gturtle import *
from random import randint
from math import *

# Define a circular region of radius r
# and make it green
def circle(r):
    c = r * 2 * pi
    s = c / 360

    penUp()
    forward(r)
    right(90)
    penDown()
    setPenColor("green")
    setFillColor("green")
    startPath()
    repeat(360):
        forward(s)
        right(1)
    fillPath()

# Do one step and one turn
def step(data, ll, aa):
    forward(ll)
    right(aa)

# Setup Python Turtle environment
setPlaygroundSize(250, 250)
makeTurtle()
hideTurtle()

# Draw the circular garden
circle(50)

# ... continue to new page
```

Listing 5.3: A round garden for the turtle but no fence.

```
# ... from previous page

# Setup limits for random choices of
# step lengths and turning angles
l1 = 1
l2 = 10
a1 = -90
a2 = 90
data = (l1, l2, a1, a2)

# Place the Turtle and start setps loop
n = 200
showTurtle()
setColor("darkgreen")
setPenColor("darkgreen")
setPos(0,0)
repeat(n):
    step(data, randint(l1, l2), randint(
    a1, a2))
```

Listing 5.4: A round garden for the turtle but no fence.

The permitted area is defined here using the circle(r) function. This produces a bitmap that can be used as a mask to check whether the turtle is inside or outside the allowed area. It is very easy to produce arbitrary shapes in this way, a feature that will come in handy later.

This code is a little more complex than the two minimum examples of LibreLogo but it should make subsequent developments easier. Comments have been added to make the code self-explanatory. The turtle's behaviour is the same: apart from the presence of a circular green area it is free to walk everywhere. Now let's put a fence.

In the following listings (5.9 and 5.10), the changes needed to obtain the fence effect are reported on the right. The step(data, ll, aa) function in listing 5.9 (left) is very simple. The version in 5.10) is somewhat more complicated because, before executing the usual forward-right combination, calls the checkStep(data, ll) function, which moves the turtle "invisibly" to check if the arrival point lies on the green area. If not, the turtle is turned right by 10°, until the arrival point is within the area. Then execution proceeds as before.

```
# Do one step and one turn
def step(data, ll, aa):
    forward(ll)
    right(aa)
.
```

Listing 5.5: Function step(data, ll, a) with no fence.

```
# Do one step and one turn if
# arrival point is inside the
# green area. Otherwise keep
# turning right till arrival
# point is inside
def step(data, ll, aa):
    if checkStep(data, ll):
        forward(ll)
        right(aa)
    else:
        while checkStep(data,
    ll) == False:
            aa = aa + 10
            right(aa)

# Check if, given position and
# direction, arrival point is
# inside green area
def checkStep(data, ll):
    oldPos = getPos()
    oldHead = heading()
    penUp()
    hideTurtle()
    forward(ll)
    color = getPixelColorStr()
    if color != "green":
        setPos(oldPos)
        heading(oldHead)
        return False
    else:
        setPos(oldPos)
        heading(oldHead)
        showTurtle()
        penDown()
        return True
.
```

Listing 5.6: Function step(data, ll, a) with fence.

Here we have the result. Of course, these codes are by no means the best ones. On the contrary, they merely outline the possibilities. There are many possible improvements, even in these minimal versions. For example, are the contacts between the turtle and the fence properly described? Try running the program several times and observe the collisions. Or, what changes if we choose different parameters? Or different angle increments when the turtle get into the function checkStep(data, ll)? There are many opportunities to explore and reflect.

Now we have provided our turtle with a virtually arbitrary garden but life is turning out to be a little boring, admittedly.

5.3 Modeling smell

We could imagine that there is some food located in the garden and design some mechanisms that allow the turtle to find it "by sense of smell". A very interesting aspect of computer simulations is that phenomena can be modeled in countless ways. Modeling is about trying to read the reality and guessing essential facts that determine behaviors. It is a game and a thoughtful experimentation ad the same time.

Let's assume here that there is some salad located somewhere and that the turtle is able to perceive variations of smell, when it is moving. To do this, in the following code we have to provide to:

1. define the garden boundary

2. plant a salad somehere

3. provide the turtle wiht the sense of smell, which depends on her distance from the salad

4. place the turtle somewhere

5. start the game

In order to manage smell we are going to use an extremely simple model: if the turtle finds that the smell is getting stronger it keeps going in the same direction, otherwise it turns. We build the code on top of that giving in listing 5.9 at 98. The "circle(r)" function is already there, instructions N. 7-21.

The salad is provided as follows.

```
def salad(sPos):
    oldPos = getPos()
    oldHead = heading()
    setPos(sPos)
    setPenColor("darkgreen")
    setFillColor("green")
    left(90)
    repeat(5):
        leaf()
        right(30)
    setPos(oldPos)
    heading(oldHead)

def leaf():
    startPath()
    arc()
    right(120)
    arc()
    right(120)
    fillPath()

def arc():
    repeat(60):
        forward(1)
        right(1)
```

Listing 5.7: Function step(data, ll, a) with no fence.

This is a very simple function, jotted down just to have a salad to be placed in the garden. The "sPos" parameter of function "salad(sPos)" represent the two-coordinates position where we want to place the salad. The function uses "leaf()" to draw the single leafs and this, in turn, uses "arc()", to draw the single arcs delimiting a leaf. It has be written having an example quoted by Seymour Papert [Papert, 1993] in mind. It could be quite different, of course, and for instance it could be parametrized to adjust the dimensions.

```
def dist(p1,p2):
    return sqrt((p2[0]-p1[0])
    **2+(p2[1]-p1[1])**2)
```

Listing 5.8: Function step(data, ll, a) with no fence.

Then, to give the turtle the power of detect smell variations we need to evaluate its distance from the food at every step. With function "dist(p1,p2)" we teach the Pythagorean theorem to the Turtle.

Now, what we have to change the execution loop for using the sense of smell:

```
repeat(n):
    step(data, randint(l1, l2)
    , randint(a1, a2))
.
```

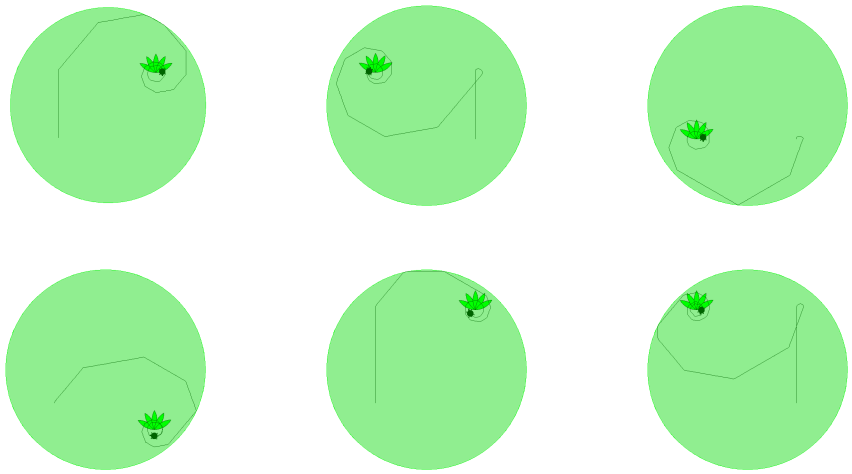
Listing 5.9: Previous walking loop, without any further action

```
p1 = getPos()
p2 = getPos()
p2[1] = p1[1] + 1
count = 0
while True:
    count = count + 1
    d1 = dist(p1,sPos)
    d2 = dist(p2,sPos)
    if d2 > 15:
        if d1 < d2:
            aa = 20
        else:
            aa = 0
        step(data, randint(l1,
     l2), aa)
        p1 = p2
        p2 = getPos()
    else:
        print(count, "
    iterations!")
        break
```

Listing 5.10: Function step(data, ll, a) with fence.

The value returned in "d2 = dist(p2,sPos)" is the distance at the last step and "d1 = dist(p1,sPos)" at the previous one. A "while" loop is used instead of the "return" one so that we can let it last forever. It stops once the Turtle gets sufficiently close to the food — "d2 < 15" in this listed version. The smelling is modeled in instructions 10-19: if distance increased in last step, then call the "stepstep(data, randint(l1, l2), aa)" function with turning angle "aa = 20", otherwise go straithforward.

This may seem a far to simple model but look how it works...



Figures show the turtle path for six different combinations of turtle starting point and light source — turtle is shown ats its arrival position. Left-right, top-bottom, respectively: $(-100, -100) \to (100, 100)$, $(100, -100) \to (-100, 100)$, $(100, -100) \to (-100, -100)$, $(-100, -100) \to (100, -100)$, $(-100, -100) \to (100, 100)$, $(100, -100) \to (-100, 100)$.

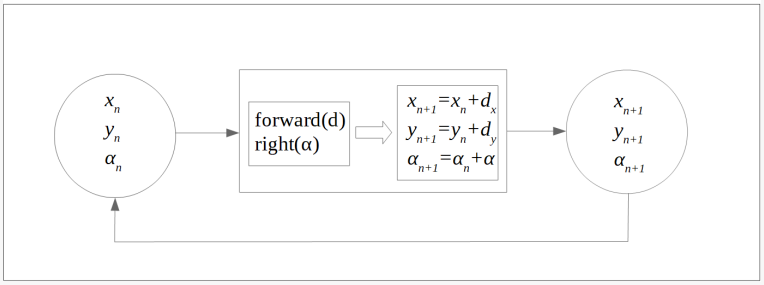
Our olfactory model may seem unrealistic, as it provides the turtle with stereotypical behaviour: if the distance increases, just turn right 20 degrees. However, regardless of the initial position of the turtle and the salad, the turtle always succeeds in getting the food.

This is the power of *feedback*. Feedback mechanisms are omnipresent in nature's dynamic processes.

Powerful concept — Feedback

The ideas of dynamic systems and of growth, which is itself intrinsically dynamic, are almost completely absent from most educational contexts. This attitude of presenting stationary scenarios reflects the substantial static nature of the educational system, which is largely unsuitable for allowing students to make sense of the world in which they live.

Natural systems grow and evolve thanks to continuous adaptation to the context, which is also dynamic. Feedback's systems are the engine of these processes. The very basic experiment proposed here helps focusing on this concept. We have provided the turtle with the capability of getting a feedback from the context at any further step, according to the following scheme:



The turtle step, coded by the sequence of commands "forward(d)" and "right(α)", causes the increments of the position coordinates, $x_{n+1} = x_{n+1} + d_x$ and $y_{n+1} = y_{n+1} + d_y$, where d_x and d_y are the Cartesian components of turtle's d step, and the change of $\alpha_{n+1} = \alpha_{n+1} + \alpha$ of heading.

The distance from food location is returned as a feedback to the turtle so that it can be taken into account in the successive step.

The turtle is capable of turning just by 20°, nonetheless it succed in reaching the goal: the power of feedback, despite the poor model.

Although the success of the previous model is remarkable, the broken line paths of the turtle are not very natural. It's interesting here to focus on another ingredient for modeling processes, in addition to feedback: randomness. Again, very small amounts of the ingredient are sufficient to improve the plausibility of the models.

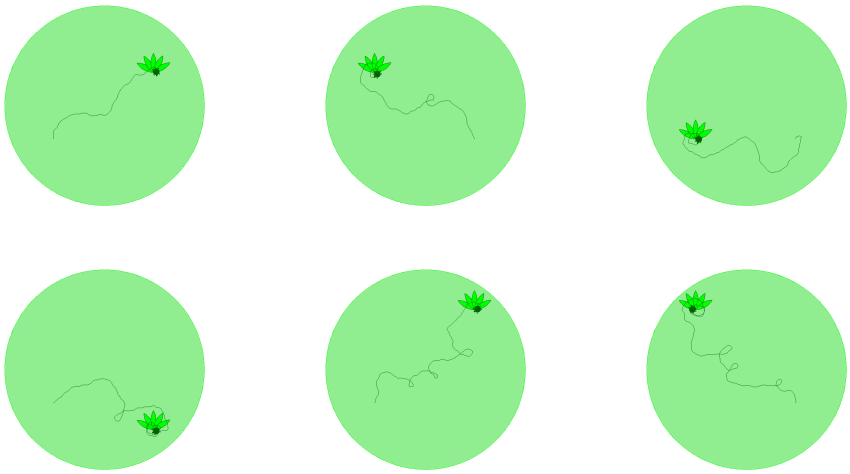
```
p1 = getPos()
p2 = getPos()
p2[1] = p1[1] + 1
count = 0
while True:
    count = count + 1
    d1 = dist(p1,sPos)
    d2 = dist(p2,sPos)

    if d2 > 15:
        if d1 < d2:
            aa = 20
        else:
            aa = 0
        step(data, randint(l1,
     l2), aa)
        p1 = p2
        p2 = getPos()
    else:
        print(count, "
    iterations!")
        break
```

Listing 5.11: Previous walking loop, without randomness

```
p1 = getPos()
p2 = getPos()
p2[1] = p1[1] + 1
count = 0
while True:
    count = count + 1
    d1 = dist(p1,sPos)
    d2 = dist(p2,sPos)
    right(randint(1,40) - 20)
    if d2 > 10:
        if d1 < d2:
            aa = 20
        else:
            aa = 0
        step(data, randint(l1,
     l2), aa)
        p1 = p2
        p2 = getPos()
    else:
        print(count, "
    iterations!")
        break
```

Listing 5.12: Walking loop with randomness: instruction N. 9

Just a drop of randomness: with instruction N. 9 — “right(randint(1,40) - 20)” — the turtle rotates by a random angle between -20 and 20 at any step. Here we are:



Figures show the turtle path for six different combinations of turtle starting point and light source — turtle is shown ats its arrival position. Left-right, top-bottom, respectively: $(-100, -100) \to (100, 100)$, $(100, -100) \to (-100, 100)$, $(100, -100) \to (-100, -100)$, $(-100, -100) \to (100, -100)$, $(-100, -100) \to (100, 100)$, $(100, -100) \to (-100, 100)$.

5.4 Modeling sight

The sense of smell is based on a sort of local measurement of an intensity. In our previous example we used distance from the food location as a smell measurement. This is dependent on the Turtle position but not on its orientation. The Turtle had "no idea" of the direction pointing to the salad. It was only comparing the intensity of smell — measured by the distance — in the current position with the previous one. When modeling sight, it is different, since in order to "see" a light source one has to turn the head. Therefore, in order to model sight we need a way of evaluating the orientation towards a given point with respect to the Turtle's heading. We call this *bearing*, which, in navigation, is the angle between the ship's course and another direction. In our context, this is the angle by which the Turtle must rotate in order to face a given object. First of all, we need the direction of the line connecting the Turtle position and the object, say a light source.

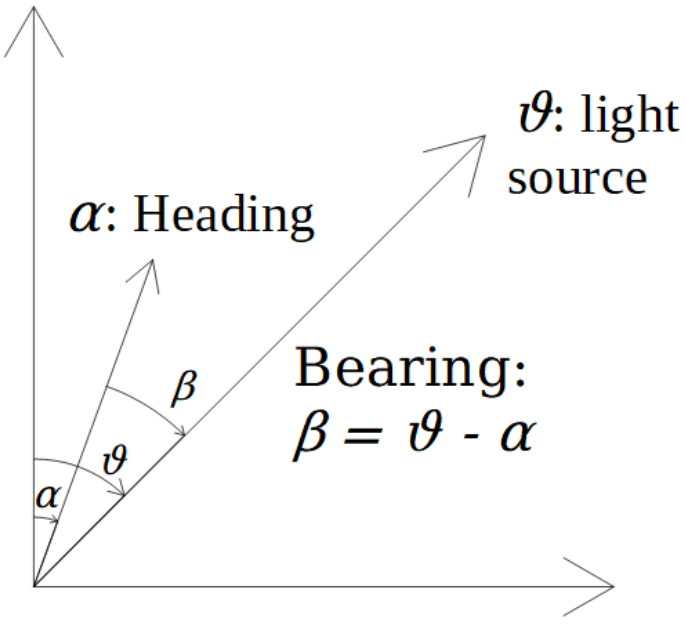


Figure 5.1: Angles are counted clockwise from 0°to 360°, α is the heading angle, θ is the light source direction angle and $\beta = \theta - \alpha$.

The angle θ is calculated (Listing at page 172) with the function "towards(px, py)". The px and py parameters are the coordinates of the light source. The pcor tuple holds the two current coordinates of the Turtle. The function returns the θ angle, taking into account the domain of the atan mathematical function.

```
def towards(px, py):
    pcor = getPos()
    dx = px - pcor[0]
    dy = py - pcor[1]
    if dy > 0:
        if dx == 0:
            return 0
        if dx > 0:
            return r2d(atan(dx/dy))
        else:
            return 360 + r2d(atan(dx/dy)
    )
    if dy < 0:
        return 180 + r2d(atan(dx/dy))
```

Listing 5.13: Function "towards(px, py)".

5.4.1 Facing light

By means of the bearing command the turtle can be directed straightforward to the light source but it can also be moved while keeping some point at a fixed bearing.

The function "face(px, py)", which is called by function " keepBearing(px, py)" (Complete listing at page 172) lets the turtle face the light source, then the turtle is turned right by a fixed amount (86°in this example) and sent one step forward. The px and py parameters are the coordinates of the light source.

```
def face(px, py):
    right(bearing(px, py))

def keepBearing(px, py):
    i = 0
    while True:
        i+=1
        face(px, py)
        right(86)
        forward(1)
        if i == 10000:
            return
```

Listing 5.14: Function "towards(px, py)".

So what is the effect of such a model? If we try we see that the turtle spirals about the point.

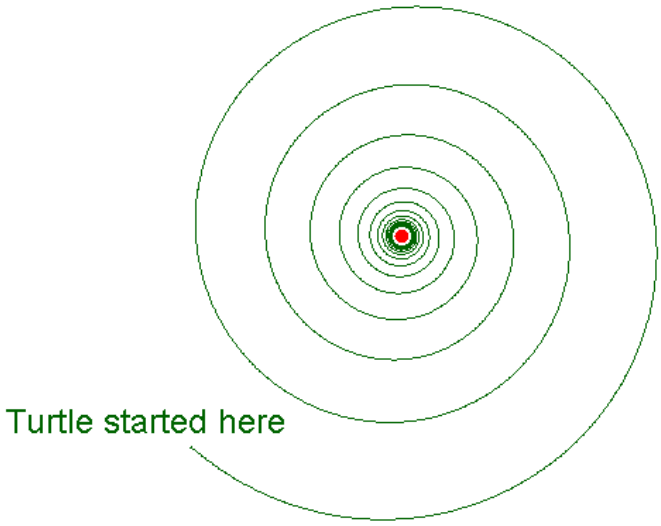


Figure 5.2: The fixed bearing model causes the turtle to spiral about the point.

Why should be worth to model such an improbable behaviour? Well, actually there is a theory about the night flight of some insects: it seems that they fly along straight paths by keeping sky lights, such as that of the moon, at a constant bearing as they fly. This happens because the moon is very distant but if they confuse the moon with a nearby light, the fixed-bearing mechanism causes them to spiral. In this way, artificial lights become traps and entomologists talk about "fixated or capture effect" [Eisenbeis, 2006, pag. 281]. Knowing that, our program could be used to explore the distance concept in this context. For instance, at which distance the trajectory begins to approximate a straight path? You can do this placing the source light out of the field of view...

Powerful concept — Scope of a theory

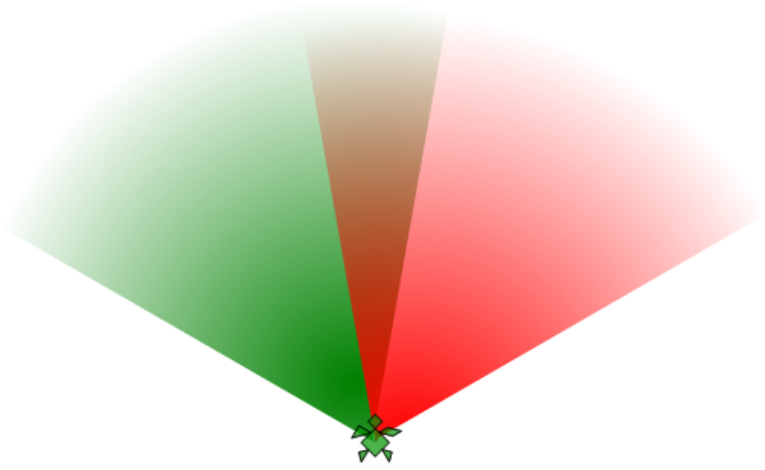
Here we have the opportunity to get an idea of what it means that a theory is true, albeit in a somewhat simplistic way. Theories stem from specific experiences about the physical world. If we start from the context of night-flying insects, we get in touch with the concept that insects are able to fly in straight path using fixed lights as reference points. And this minimal theory is true, but as long as new experiments will confirm it. Suppose we begin to make experiments with closer lights. Soon we will discover that our theory does not hold true any more, since by maintaining a fixed bearing to a close light, insects will spiral. So, does it mean that the previous theory is wrong? No. Very often in common life answers to this kind of questions are seen as dichotomous ones. That's a wrong conception of theories. A theory makes sense with respect a given context. When

context changes we have to make sure that the theory keeps holding true. Actually, in our minimal example, the first theory is somewhat "contained" in the second one. Insects always spiral around lights but in the case of the moon they would spiral in an extremely large path in space. However, within short ranges a curved path is approximated very well by a straight one.

This is, for instance, what happens with Newton and Einstein gravity theories. Newton theory is perfectly fit to send someone to the moon but if you have to explain the effects gravity on light you need the Einstein's one, however, the latter does not contradict the former but includes it.

5.4.2 Two-eye vision

Vertebrate animals have two eyes and so turtles do. Let's provide our turtle with a two-eye vision mechanism. We start establishing a separated field of view for each eye:



The idea is based on the ability to tell whether the light source is within each eye's field of view.

```
# check if seen by left eye
def leftEye(b):
    if b > 350:
        return True
    if b < 60:
        return True
    return False

# check if seen by right eye
def rightEye(b):
    if b > 300:
        return True
    if b < 10:
        return True
    return False
```

Listing 5.15: Functions rightEye(b) and leftEye(b) render True value if the point is in that eye's field of view.

Here the b parameter is the bearing, which we have already seen, i.e. the angle that the turtle would need to turn right in order to face the light source.

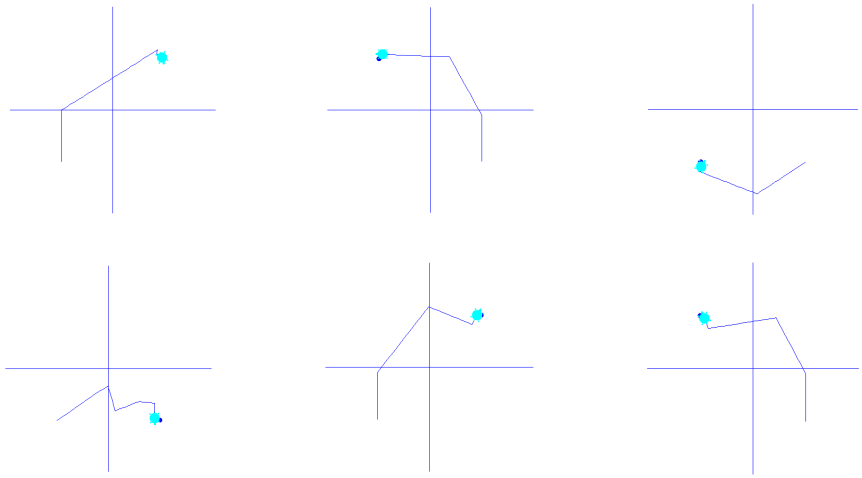
Once we have given this kind of sight sense to the turtle, we have to model the behaviour. As in the case of smelling, we define a very simple model: if the source is seen by one or both of the eyes, the turtle goes ahead 10 points, otherwise choose a random angle between 1 and 359 an turn by that angle.

```
# heading for light source
def headFor(px, py):
    while dist1(px, py) > 10:
        b = bearing(px, py)
        if leftEye(b) or
    rightEye(b):
            forward(10)
        else:
            r = randint(1,359)
            left(r)
```

Listing 5.16: Function headFor(px, py) drives the turtle in the search of the light source.

The code works with a loop which stops as soon as the distance form the light source is less than or equal to 10 points. At each iteration, first the bearing is calculated and then a check about the presence of the light within the field of view is made.

In the following figure the paths chosen by the turtle are shown for six different combinations of starting point and light source position.



Figures show the turtle path for six different combinations of turtle starting point and light source — turtle is shown ats its arrival position. Left-right, top-bottom, respectively: $(-100, -100) \rightarrow (100, 100)$, $(100, -100) \rightarrow (-100, 100)$, $(100, -100) \rightarrow (-100, -100)$, $(-100, -100) \rightarrow (100, -100)$, $(-100, -100) \rightarrow (100, 100)$, $(100, -100) \rightarrow (-100, 100)$.

Of course, this code can be improved in many ways. For example, you could improve the random choice of a new direction by narrowing the range of possible angles. For instance, one could improve the random choice of a new direction, by restricting the range of possible angles (instruction 8 of listing 5.16. Or by experimenting with a different field of view or introducing asymmetry between the left and right eye. Or injecting some randomness into the movement. Here again, as in the case of the previous model of smelling, we can appreciate the effectiveness of the feedback mechanism, despite the extreme simplicity of the sight model.

5.4.3 Two-eye vision with intensity perception

A further refinement of the vision model is the intensity of the light stimulus that is perceived by each eye. This is slightly more complex, since the strength and distance of the light source as well as the angle of incidence in relation to the viewing plane must be taken into account.

```
# intensity perceived from
# left eye
def intensityLeft(px, py):
    strength = 1000000
    b = bearing(px, py)
    if not leftEye(b):
        return 0
    fact = strength / (dist(px, py)**2)
    a = bearing(px, py) - 45
    return fact * cos(a*pi/180)

# intensity perceived from
# right eye
def intensityRight(px, py):
    strength = 1000000
    b = bearing(px, py)
    if not rightEye(b):
        return 0
    fact = strength / (dist(px, py)**2)
    a = bearing(px, py) + 45
    return fact * cos(a*pi/180)
```

Listing 5.17: Functions intensityLeft(px, py) and intensityRight(px, py) return the perceived intensity.

The light source variable, "strength", was adjusted to an arbitrary value, so to obtain an adequate drawing scale. If the source is not within the field of view, a 0 value is returned. Otherwise the returned value is given by "strength" times the inverse of the squared distance and a $cos(\alpha)$ factor, where α is the incidence angle: 0 for perpendicular light rays ($cos(\alpha) = 1$.

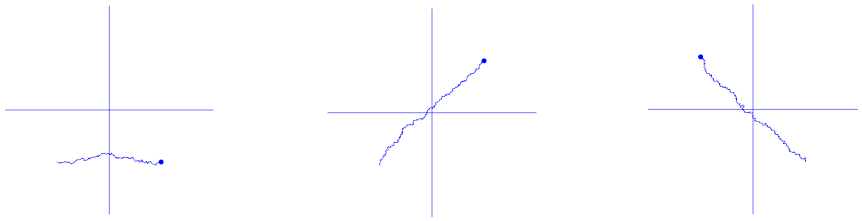
In the calculation of the angle α we have to consider the fact that both eyes are offset 25°from the turtle's heading, in one direction the right one, in the left one the other — in other words, the bearing has to be calculated with respect to the eyes heading instead of to the Turtle's one. In the following listing, the headBySight(px, py) function drives the turtle's path by means of a while loop which (in this example) will stop once the turtle will be 3 point distant from the source light.

```
# heading for light source

def headBySight(px, py):
    i = 0
    while dist1(px, py) > 3:
        i = i + 1
        left(randint(-90,90)) # some randomness here
        forward(1)
        iL = intensityLeft(px, py)
        iR = intensityRight(px, py)
        if iL > iR:
            left(10)
        elif iL < iR:
            right(10)
        else:
            while intensityLeft(px, py) == 0 and
    intensityRight(px, py) == 0:
                left(randint(-180,180))
```

Listing 5.18: Functions intensityLeft(px, py) and intensityRight(px, py) return the perceived intensity.

In this version some randomness is added with the instruction N. 7, where the turtle turns left at a random angle between -90°and 90°. By deleting this instruction, the algorithm works the same way, the only difference being that the turtle travels more directly to its target. At every cycle, the turtle does one-point step (instruction N.8), then it compares the intensities perceived by the two eyes: if it sees more light to its right, it turns slightly to the right and if it sees more light to the left it turns slightly to the left (instructions N. 11-14). In this way the turtle walks forward while trying to keep the amount of light received by the two eyes equal. If the intensities are the same but both equal to zero, then the turtle keeps trying new random left turns until at least one of the eyes sees something. If the intensities are the same but not equal to zero, the turtle does nothing.

In the following figure the paths chosen by the turtle with the intensity-based model are shown for six different combinations of starting point and light source position.



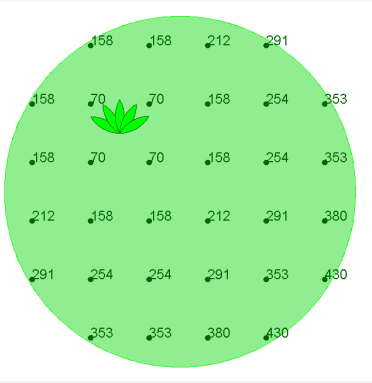
Figures show the turtle path for six different combinations of turtle starting point and light source — turtle is shown ats its arrival position. Left-right, top-bottom, respectively: $(-100, -100) \rightarrow (100, 100)$, $(100, -100) \rightarrow (-100, 100)$, $(100, -100) \rightarrow (-100, -100)$, $(-100, -100) \rightarrow (100, -100)$, $(-100, -100) \rightarrow (100, 100)$, $(100, -100) \rightarrow (-100, 100)$.

Again, there are unlimited possibilities of exploring different situations. For instance, what happens if one of the eyes sees less light than the other? And if you mask one of them? Or, one could observe how the turtle behaves in presence of one or more additional light sources.

Powerful concept — Scalar and vector fields

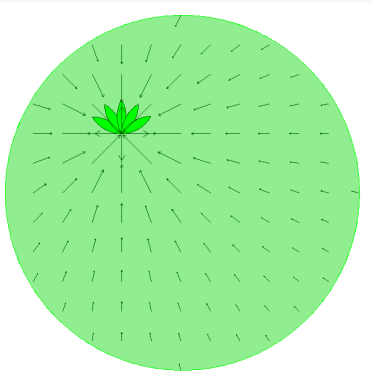
The examples that we have explored so far in this chapter all evoke, implicitly, the concept of field. In physics the concept of field is the basis of a wide range of topics. In this context it suffices to describe it as a physical quantity, represented by a number or vector, that has a value for each point in space. For instance a weather map giving a temperature value for each point describes a scalar field. In physics, with the term scalar we mean a quantity described by a simple number. The graphical representation of a scalar field, for example in a two-dimensional space, may consist in numbers printed at certain points on a map. Instead, a vector is a physical quantity which needs to be described by an intensity, a direction and a sense. Graphically, vectors are represented in the form of arrows, whose length represents intensity, while the arrowhead indicates sense. A weather map showing the distribution of winds will have vectors represented as arrows superimposed to the map.

The salad search tasks we've seen so far are not physics problems, strictly speaking, however they easily evoke the concept of field. In the case of the smell model, the turtle was concerned just with smell intensity, trying to find its way by comparing the intensity with that experienced in the previous step. In this model we assumed that the intensity of smelling is proportional to distance.



Instead, in the case of the sight models, where the turtle was trying to reach a light source, we were concerned with direction and sense and not only with intensity, thus involving the concept of vector field.

Each arrow represents the light stimulus at the point of its origin, the length being proportional to the intensity of light at that point and the arrowhead giving the sense along the arrow's direction. In this case we assumed that the intensity of a point light source is inversely proportional to the square of distance.



5.5 Modeling interactions

5.5.1 Multitasking

A "handmade" example

The previous examples pave the way to modeling interactions between turtles. However, first of all we need a way to handle multiple turtles. This is possible in the TigerJython environment, not with LibreLogo and Xlogo. Basically, what we need, in order to make several turtles work at the same time, is the capability of performing *parallel processing*, which means executing multiple sequences of instructions independently at the same time. The Python language used in the TigerJython environment has a good support for multithreading. We will see some examples, but first let's try to build a multi-turtle program with LibreLogo,

although it is a strictly single-turtle environment. Such a "handmade" example allows us to dig a little bit into the concept of multitasking, to which we are so used today.

The trick consists in creating two separate sequences of instructions, one for each turtle, where just a single step is executed. Assuming we have a red and a green turtle, we have:

```
TO EXECRED
  PENCOLOR ''red''
  FORWARD RANDOM(10) + 1
  RIGHT RANDOM(120) - 60

END

TO EXECGREEN
  PENCOLOR ''green''
  FORWARD RANDOM(10) + 1
  RIGHT RANDOM(120) - 60
END
```

Listing 5.19: Each turtle has its own processing function.

Then we need a function to manage the turtles. This is very simple: just a loop where the two functions "EXECGREEN" and "EXECRED", are executed alternatively, where the state of each turtle has to be restored before its next move, and saved after completion.

```
TO EXECBOTH
  REPEAT 50 [
    RESTGREEN EXECGREEN SAVEGREEN
    RESTRED EXECRED SAVERED
]
END
```

Listing 5.20: Managing two turtle simultaneously

The states of the two turtles are saved and restored through functions "SAVEGREEN", "RESTGREEN", "SAVERED" and "RESTRED". For instance, for the red turtle we have (complete program list at page 180):

```
TO SAVERED
  GLOBAL PRED, HRED, PGREEN, HGREEN
  PRED = POSITION
  HRED = HEADING
END

TO RESTRED
  GLOBAL PRED, HRED, PGREEN, HGREEN
  X = PRED[0]
  Y = PRED[1]
  PENUP
  POSITION [X,Y]
  PENDOWN
  HEADING HRED
END
```

Listing 5.21: Managing two turtle simultaneously

Here we have an example of three runs.

Three runs of LibreLogo EXECBOTH program (page 179).

Multitasking: the computer juggling trick

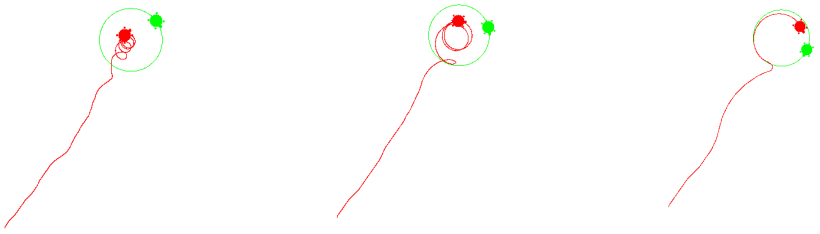
Computers are designed according to the *von Neumann model*, i.e. as sequential machines which, on the basis of a program, execute instructions, one after the other, in small successive intervals of time. True multitasking is only possible if different processors run different tasks simultanoeusly. Nowadays there are *massively parallel* computers, which are composed by a large number of specially interconnected von Neumann processors, but these are high-end research machines. A limited amount of parallelism is also present in most computers today, that are usually equipped with *multicore processors* — for instance the laptop where I'm writing right now has a four-core processor. However this feature is unvisible to users, being used by some application software to accelerate processing speed in peculiar ways. In fact, in our daily life we take for granted that our computers are perfectly capable of doing many things at once. But in most cases this is an illusion, a trick. It's actually a kind of juggling, where the computer handles one task at a time, in turn, very quickly. The example we have shown is very basic, indeed, but it can be used to develop some reflections about

the behaviour of computers, as far as the managing of concurrent tasks is concerned. The concept of multitasking can be considered at various levels, for instance as concurrent *threads* in a single application — in this case one talks about *multithreading* — or as competing processing within the operating system. In any case, this exercise can be used to point out that in order to manage simultaneous activities there has to be a kind of software hierarchy. For instance the procedure EXECBOTH runs at higher level since it is responsible of the assigning tasks to the EXECRED and EXECGREEN procedures.

5.5.2 Interacting turtles

Predator prey

Let's turn to the TigerJython environment to tinker a little bit with interacting turtles. TigerJython is more suitable since it has true multithreading support. In section 5.4.3 on page 108 we modeled a two-eye vision system based on intensity perception and we used it to simulate the path followed by a turtle towards a source light. Here we are going to take advantage of multithreading to let two turtles run simultaneously. In our simulation the turtle 1 plays the role of the predator while the turtle 2 plays the role of the prey. This one is simply turning in circles, unaware of all the rest — could be the situation of a rainforest bird intent on performing its complex dance moves, unaware of the incoming predator. Turtle 1, the predator, is chasing its prey using its own sight, based on the same model described in section 5.4.3 where the turtle had to reach a light source. Of course, the target is moving in this case, so the paths of the predator can vary a lot, depending on a number of conditions.



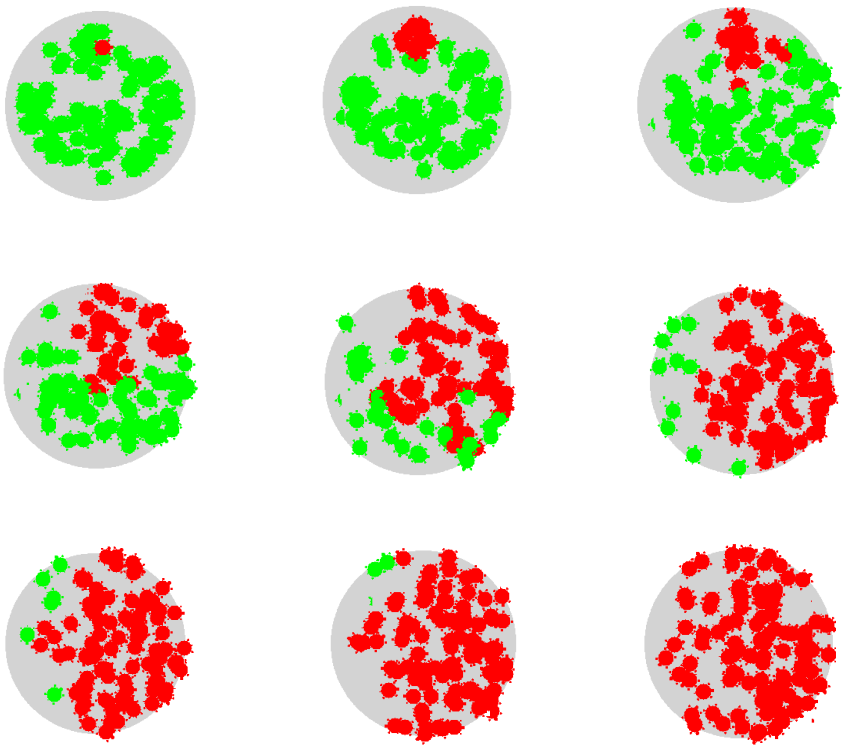
Three runs with different speeds of the predator, from left to right: 1/10, 1/4 and 1/2 of the prey speed.

Here there is much to experiment and think about. For example, one could study the effects of different choices for the relative initial positions, the path of the prey — in this case, for example, the radius of the trajectory circles — and the relative speeds. One could also change the sight model parameters of the prey. Or give a chance to the prey to sense the presence of the predator, for

instance through the smelling model we discussed in section 5.3 (pag. 96)[3].

Contagion dynamics

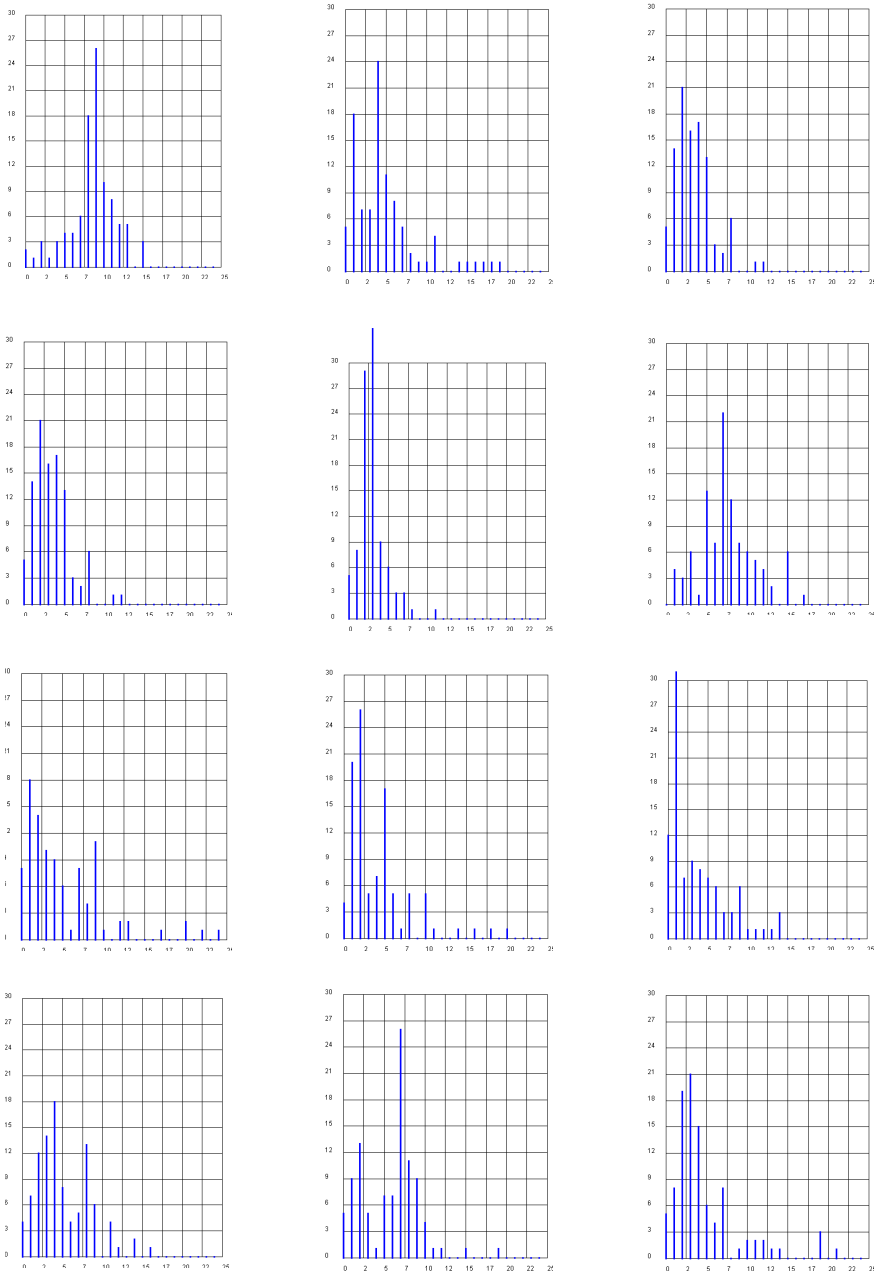
The year in which I am writing this text is marred by the COVID-19 pandemic caused by the coronavirus SARS-CoV-2. As a result, analyses around the dynamics of COVID-19 infections are widely reported in all media. Thanks to TigerJython's turtle geometry and multithreading, it is possible to simulate the dynamics of contagion in a community. In the following example, we have 100 turtles free of moving randomly in a circular region. At the beginning, they are all healthy except one. While the turtles are walking around, every time a healthy turtle stumbles on a sick one, it gets sick too, becoming contagious in turn. In this version, the level of contagiousness of the virus can be adjusted by means of the threshold distance below which a sick turtle is able to infect a healthy one.



Progression of contagion in a population of 100 turtles of which only one is initially ill.

[3]Recently, it has emerged that cheetahs are excellent night hunters thanks to their extraordinary night vision while their prey, to prevent their attacks, must rely primarily on the sense of smell.

In this setting, two factors have an important influence on dynamics: the contagiousness of the disease and the closed environment. Therefore the typical trend is a sudden growth of contagions followed by a more or less fast decrease. The first part is due to the exponential nature of the contagion phenomenon, due to the fact that the number of new infections is proportional to the number of ill turtles. The second one is due to the saturation of the population. What we get is a kind of simulation of herd immunity.



Progression of contagion in 12 different runs of the simulation. The histogram reports the number of new infections at each iteration.

However, already in this small example, we can see how the contagion progresses in bursts that suddenly involve a large number of subjects. Then the contagions decrease because the available healthy subjects are no more available. Of course, the simulation can be extended to larger samples, possibly chang-

ing other parameters such as infection distance or starting conditions. Here we limit ourselves to highlighting how those bursts represent the germ of exponential growth that can characterise infectious diseases. Exponential growth occurs when the increase is proportional to the existing amount and has the general form a^x, which is exactly what happens in epidemics, where each new patient becomes a vehicle for disease.

But are we sure we "feel" the vertigo of exponential growth?

Exponential growth should be anxiogenic, because of what it entails: in essence an irreparable loss of stability. But the expression 2^n hardly evokes this sensation, except in those who are provided with some mathematical insight. Maybe the story of the inventor of chess and the king of Persia can help.

The story is so famous that it was even mentioned by Dante Alighieri [Alighieri, 1321, canto XXVIII - vers. 88-93]:

> E, poi che le parole sue restaro,
> non altrimenti ferro disfavilla
> che bolle, come i cerchi sfavillaro:
> l'incendio suo seguiva ogni scintilla;
> ed eran tante, che 'l numero loro
> più che 'l doppiar de li scacchi s'immilla.
>
> And when her words had ceased, not otherwise
> doth iron when still boiling scintillate,
> than yonder circles sparkled. Every spark
> followed its Kindler; and so many were they,
> that their whole number far more thousands counts,
> than ever did the doubling of the chess.

The number of the sparks that come out of Beatrice's words multiply like the flames of iron in the forge, more than the doubling of chess.

According to the legend, the inventor of chess would have asked the king of Persia who wanted to compensate him, one grain of wheat on the first chess box, two on the second, four on the third and away doubling. To the king this seemed to be nothing, but when the calculation was made, it emerged that not even cultivating all the known lands with wheat would have been possible to satisfy this request!

Let's try to get an idea of this quantity. First of all it is necessary to determine how many grains of wheat there are. In the first square we place a grain. In the second 2, thus in all they are 3. In the third 4 and in all they are 7. It is easy to see that if the number of squares we are filling in is n , the number of grains is $2^n - 1$. So for 64 boxes, the total number of grains will be given by $2^{64} - 1$.

This small script seems harmless and instead it's a bomb! To get an idea of its value it is better to express it in powers of 10 instead of 2. This is done by solving the following equation

$$10^x = 2^{64} \tag{5.1}$$

Using logarithms you get x:

$$x = 64 \log_{10} 2 = 19.3 \tag{5.2}$$

So the number of seeds is about $10^{19.3}$. In other words, they would be more than 10 billion billion seeds. What does it mean? Using a scale it is easy to check how a handful of about 100 seeds weighs about 5 grams. So a seed weighs 5×10^{-2} grams and, consequently, the amount required by the mathematician would be $5 \times 10^{11.3}$ tons. That's a lot, since the world wheat production amounts to "just" 35 million (35×10^6) tonnes per year on average: the amount of wheat required by the mathematician would therefore last more than 28000 years for the whole humanity! Beware of exponentials!

Chapter 6

Simulation: fractals growth

6.1 Recursion

Everyone knows how two opposing mirrors generate an amazing fugue of images . Mirror number 1 can do only one thing: reproduce the scene in front of it. Even mirror number 2 can do only the same thing, but in doing so it also reproduces mirror number 1, including the scene it contains, which in turn reproduces the scene in mirror number 2 and so on, *ad libitum*. It is a phenomenon that strikes because it allows us to peek into the infinity, normally inaccessible to human experience. That is recursion.

```
1 TO RECURSION
2   RECURSION
3 END
4
5 RECURSION
```

Listing 6.1: Minimal recursion program

What's going on here? Nothing: program RECURSION keeps calling itself — a perfect representation of self-referentiality! In this code fragment the Turtle is executing just one command: RECURSION. Actually, if you try to write this code in LibreLogo and you run it, nothing will happen except, after a while, a message box will appear telling you "Program terminated: maximum recursion depth (1000) exceeded." This is a kind of safety measure, because recursive programs may put the computer in trouble, if a stopping criterium is not provided. Every time a procedure is called, the system will allocate some memory needed for its activities. If you let the machine go on with this process, an infinite quantity of memory will be claimed, which is not the case of your computer, of course.

Let's make this silly program a little bit more interesting.

```
TO RECURSION D
  CIRCLE D
  RECURSION D+1
END

RECURSION 1
```

Listing 6.2: Less minimal recursion program

This is more fun, try it: what do we get? You should see a balloon growing. And you'll need a stop rule, because this code will make the balloon go over the page: recursion programs need a stopping rule. For instance in this way:

```
TO RECURSION D
  IF D < 100 [
    CIRCLE D
    RECURSION D+1
]
END

CLEARSCREEN
RECURSION 1
PRINT ``Done!''
```

Listing 6.3: Recursion program with stopping rule

To create a stopping rule we need an instruction able to evaluate a given condition. In LibreLogo we can do that with the IF command together with a condition, which in this case could be "D $<$ 100". Instruction N. 2 says that if D is less than 100 then instructions 3 and 4, within the square brackets, will be executed, otherwise no. In this case, we get out from program RECURSION and instruction N. 10 is executed.

In TigerYjthon this program would look like that:

```
1  from gturtle import *
2
3  makeTurtle()
4  clearScreen()
5  hideTurtle()
6
7  def rec(d):
8      delay(100)
9      if(d < 100):
10         dot(d)
11         rec(d+1)
12
13 rec(1)
14 print("Done!")
```

Listing 6.4: Recursion Python program with stopping rule

Apart from syntactical differences, here we added a delay at instruction N. 8 because in TigerYjthon the program runs much faster and the growth of the balloon would not be visible.

So far, what could these simple and unhelpful examples be used for? Considering that they could also easily be made with simple loops? Why bother with these fanciful but strange constructions if loops can do the same? Actually, it's true, everything[1] that can be done with recursion can also be done with loops. However, there are problems that lend themselves easily to a recursive description, for instance *fractals*.

6.2 Fractals

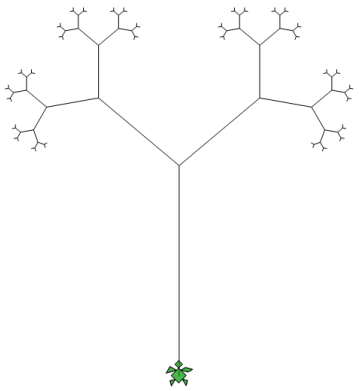
A fractal is a never-ending geometrical pattern, i.e. infinitely complex patterns that are self-similar across different scales. They became popular, even in mathematical circles, in the 1980s, thanks to the work of Benoît Mandelbrot. In fact, fractals were known before they got a name. For instance, the Cantor set, a set of numbers (we give an example later on) which is fundamental in many branches of mathematics, was published by Georg Cantor in 1883 [Peitgen et al., 1992, pag. 67]. However, before Mandelbrot, Cantor set and other strange entities were regarded as exceptional objects, as "mathematical monsters". Mandelbrot merit was to have shown that these "extreme shapes", which now are called *fractals*, represents actually the normality, instead of the exception, showing that in Nature there are countless examples of fractal structures. Thus the title: *The Fractal Geometry of Nature* [Mandelbrot, 1967].

Fractals look the same at different scales, that's what self-similarity means. We already encountered self-similarity, in the Bernoulli's, *spira mirabilis*, that is the logarithmic spiral (pag. 53 and following), another over-represented shape in nature too. To get an idea about growing fractals, a very basic example may be appropriate.

[1] The topic is not trivial, experts may discuss at length on this subject. In our context "everything" is ok.

```
1  TO TREE LL
2    IF LL > 2 [
3    FORWARD LL
4    LEFT 50
5    TREE LL/2
6    RIGHT 100
7    TREE LL/2
8    LEFT 50
9    BACK LL
10   ]
11 END
12
13
14 TREE 200
```

Listing 6.5: Simple fractal tree in Logo

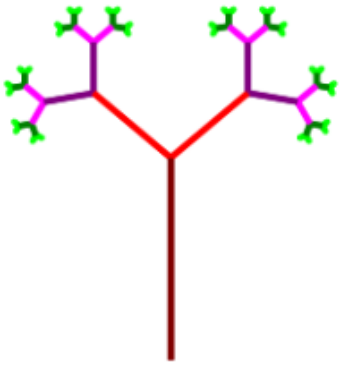


The TREE procedure is recursive because it calls itself, twice. The fact that the calls are two is related to the recurring bifurcated structures that represents the "basic idea" of this tree. To tell the truth, there are nothing else than bifurcations in this tree, which is simply made of bifurcations, the unique differentiation being the spatial scale. Let's go into some details. First, the program is called in instruction N. 14: TREE 200. Therefore, when TREE is entered, the value of LL is checked if it is less than 2 (Instr. N. 2). Since this time it's value is 200, execution goes ahead. With instruction N. 3 the turtle goes forward by LL=200 points and turns left by 50°, thus drawing the trunk and turning itself left. Then, instead of keeping drawing something it calls TREE (N. 5), but passing it a value of LL/2. Let us refrain from diving into this TREE call and assume to have it done. In N. 6 the turtle turns right by 100°, in N. 7 calls TREE LL/2 again, then in N. 8 turns left by 50°and comes back along the last branch. [2].

The simple example of the stick tree lends itself to some interesting considerations. First of all, it can be used to reflect on the recursive process, since recursive codes are not so intuitive at the beginning. When you try to figure out what the code is doing by following instructions in sequence, you easily get lost, especially if the procedure calls itself more than once at each recursion level. To help understanding you can think of a team of turtles instead of just one turtle doing all the work. The idea is that each turtle operates at a specific recursion level.

[2]In the home page of the downloadable materials you can see the animation of the tree growth: $http://iamarf.ch/Codice-LOGO/$. This address point to a page I made available for the "Exploring the land of powerful mathematical ideas with Logo's Turtle workshop" (N. 5) which has been hold at the 10. Schweizer Tag für den Informatik unterricht, on 5th February 2020.

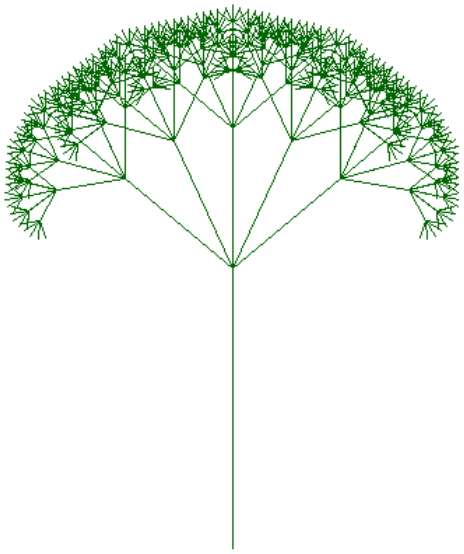
We are distinguishing turtles working at different level of recursion by colour. In this example, at the first level we have the brown turtle, who is drawing the trunk. At the second one we have the red one who is called two times, for the two successive first, larger, branches. Then at the third one we have the violet turtle who is called four times fot the next set of smaller branches, and so on...



The example of the stick tree can also be expanded a bit to explore the interplay between infinity and infinitesimal. Let's rewrite the code in slightly more complex way. We use here a Python version because the program runs faster[3].

```
from gturtle import *
from math import *

makeTurtle()
clearScreen()
hideTurtle()

def tree(ll,aa):
    if(ll > 10):
        forward(ll)
        left(aa)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        left(aa)
        back(ll)

setPenColor("dark green")
setPos(0,-100)
ll = 200 # First branch length
aa = 50 # Deviation angle

tree(ll,aa)
```

Listing 6.6: Stick tree with more branches.

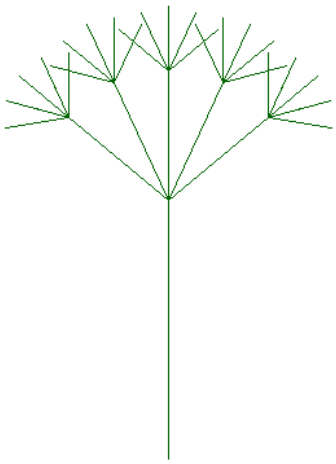


Two things have changed since the previous Logo version: 1) we have added

[3]We're actually jumping between the Logo and Python programs. It's a good exercise, try translating some small programs from one language to another yourself.

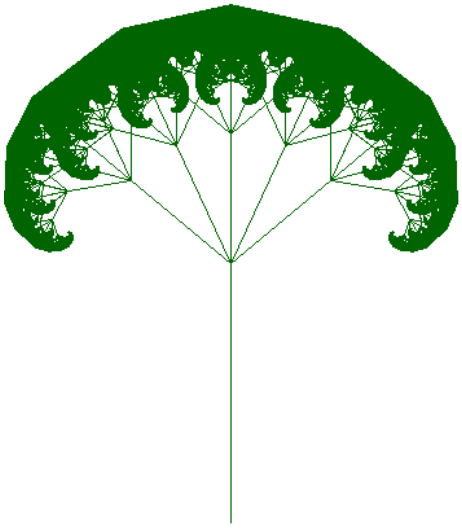
the "aa" (angle) parameter to the "tree" subprogram and 2) rather than generating two branches at each recursion level, five branches are now generated. Instead we mantained the scale factor 1/2, i.e. at each recursion the tree suprogram is called with the branch length "'ll" divided by two. The result is nicer, it looks like a dandelion, or a pine tree. But apart from that, this version allows interesting reflection about infinity and infinitesimal. In subprogram "tree" we have inserted a stop rule which requires that the length of the branches should not be less than 10. Since we started with ll =200, we have obtained four recursion levels with values ll = 100, 50, 25 and 12.5.

It is obvious that if we had set the threshold at a higher level, we would have obtained a shorter tree with less branches, such as this one, where we set the threshold at 30 obtaining only two additional levels: ll = 100, 50. On the other hand, by reducing the threshold, allowing the creation of smaller branches, the tree will become more intricate but also taller. But what will happen if we allow the recursion process to continue indefinitely? By adding more and more branches will the tree become infinitely tall, or will the progressive reduction in branch size compensate for the explosion of the tree? Which one of these two opposite tendencies will prevail?



You can try by yourself, it's very easy: you have just to change the stopping rule at instruction N. 9. For instance, we try now with "if(ll >' 0.1):'

```
from gturtle import *
from math import *

makeTurtle()
clearScreen()
hideTurtle()

def tree(ll,aa):
    if(ll > 0.1):
        forward(ll)
        left(aa)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        right(aa/2)
        tree(ll/2,aa)
        left(aa)
        back(ll)

setPenColor("dark green")
setPos(0,-100)
ll = 200 # First branch length
aa = 50 # Deviation angle

tree(ll,aa)
```



Listing 6.7: Stick tree with more branches and deeper recursion.

In this case the recursion levels are ten: ll = 100, 50, 25, 12.5, 6.25, 3.125, 1.56, 0.78, 0.39 and 0.19. The vegetation over there is thick, indeed! But is this tree really higher? Perhaps. Let's try to compare them by placing a 400 points ruler at the center of each tree.

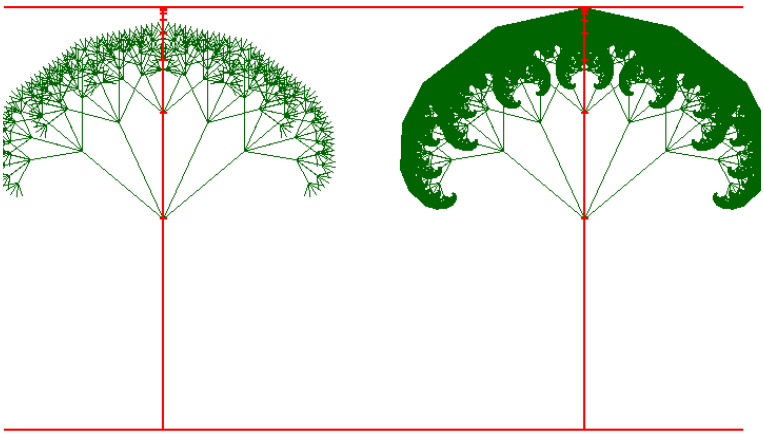


Figure 6.1: Transition from four to ten recursion level. Distance between the two horizontal bars is of 200 pixels. On the vertical bars, tick marks beginning of successive branches, proportional to 1/2, 1/4, 1/8...

Yes, the tree continues to grow with the number of recursions, but at a progressively slower rate. Will this growth remain confined below a certain limit, while pushing recursion forever? Computer experiments could be perfected, but only mathematics can be expected to provide a definitive answer. The structure of our tree is suited to get this answer from a very well known formula. At each recursion, five equispaced branches are generated and the central one has no deviation angle with respect to its originating branch. Thus, at the center we have all the successive branches aligned on a vertical line reaching the top of the tree, and this is the tree height. The successive branches are reduced by a factor 2, at every recursion. Thus, in the case of the first tree the length of the central vertical line is given by

$$H = d + d/2 + d/4 + d/8 + d/16 + d/32 \tag{6.1}$$

where H is the total height and d is the length of the first branch, the trunk, $d = 200$ in our code. Mathematics give us the tools to manage the infinity. We cannot let our computer do infinite recursions but we can write:

$$H = d \sum_{n=0}^{\infty} \left(\frac{1}{2}\right)^n \tag{6.2}$$

This is a geometrical progression, for which

$$\sum_{n=0}^{\infty} a^n = \frac{1}{1-a} \tag{6.3}$$

holds true if $a < 1$. Therefore in our case we have

$$H = d\frac{1}{1-\frac{1}{2}} = 2d \tag{6.4}$$

Well, it is this value we used to draw the red rulers. Now we know that the recursion process will make a dense tree crown but the total height will remain equal to $200 \times 2 = 400$ points. This is another example of an infinite process giving rise to a finite result.

If we look at figure 6.1 we see that, while going on further with the recursions, the tree will not grow more than $2d$ in height — we *really know* this thanks to math — but what happens is actually that we are *filling some patch of plane*. Although we are just drawing segments, the recursive process leads us to filling the plane, in some measure, but how much? Let's try to answer in the footsteps of Mandelbrot, along the coast of Great Britain...

6.2.1 L-systems: the turtle language for growing plants

Turtle geometry was used to model plant growth [Prusinkiewicz et al., 1988, Peitgen et al., 1992]. The stick tree algorithm (6.5) that we wrote in the previous section actually represented plant growth, although it was extremely simple. However, the algorithm we wrote was to make the recursion mechanism as simple as possible to understand. In fact, scientists describe the movements of turtles in a special formal language called the L-system. The advantage is that once the method is understood, it is much easier to tinker with different and more complex shapes. So let's rewrite the stick tree algorithm using the L formal language.

A so called tree L-system is specified by three components: an *alphabet*, an *axiom* and a set of *production rules*. Each production rule is a string composed with labels belonging to a predefined alphabet. The alphabet determines the category of plants we intend to model. The axiom and the production rules determine the specific plant we want to simulate. Square brackets are used to delimit branches: elements between square brackets determine a specific branch. The [and] brackets are coded as stack push and pop operations on the turtle states stack. The following table shows the alphabet we are going to use for modeling herbaceous plants.

Label	Instruction
F	move forward by a certain fixed step length l drawing a line
f	move forward as above for F but do not draw the line
$+$	turn left by a fixed angle
$-$	turn right by a fixed angle
$[$	new branch opening...
$]$	... current branch closing
B	do nothing (i.e. turtle does not go anywhere)

Table 6.1: Tree L-system we are going to use

How are these instructions translated into turtle commands? Tree L-systems are set up by defining once for all forward steps and rotation angles so that F could be coded as forward(l), "+" as left(d) and "-" as right(d) where, for instance, $l = 100$ points and $d = 50°$. The B L-system instruction is a kind

of *no operation* command that results in a recursion invocation without any other action taking place. From a general point of view, an L-system generates a succession of stages where, the rules of production tells how to move from one stage to the next one, regardless of the stage level, by means of a set of substitutions. From a coding perspective, the construction of an L-system is initiated by calling a recursive function, possibly containing multiple recursive calls. To show how this works, let us look at the simple example of the stick tree[4] we have already seen on page 124.

Axiom	B
Production rule	$B \rightarrow F[+B][-B]$
Production rule	$F \rightarrow FF$

Table 6.2: Axiom and production rules for the stick tree

And this is the program listing, which can also be found at page 181, together with those relative to the successive examples.

[4]Since the shape evokes a basic tree we keep calling it that way. However, the simple growth scheme is that of an herbaceous plant.

```
# Stick tree "plant"
# with L-system
# 18.6.2022

# Axiom: B
# B -> F[-B][+B]
# F -> FF


from gturtle import *

makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
        setPos(lp.pop())
        heading(lh.pop())

def B(l, delta, iter, lp, lh):
    if iter == 1:
        forward(l)
    else:
        forward(l)                    # F forward
        tpush(lp,lh)                  # [ beginning right branch
        right(delta)                  # - right rotation
        B(l/2, delta, iter-1, lp, lh) # B recursive branch call
        tpop(lp, lh)                  # ] closing right branch
        tpush(lp,lh)                  # [ beginning right branch
        left(delta)                   # + left rotation
        B(l/2, delta, iter-1, lp, lh) # B recursive branch call
        tpop(lp, lh)                  # ] closing right branch


l = 100
delta = 50
iter = 7

lp = []
lh = []

B(l, delta, iter, lp, lh)
```

Listing 6.8: Basic recursive function for the L-system stick tree

Let us analyse this program in detail. In instructions 10-14 the turtle library is imported, a turtle instance is created and the screen is cleaned. Finally, the option to hide the turtle is chosen allowing for the fast execution of drawings — if one needs to follow the turtle path the instruction showTurtle() must be given. Then the definition of three functions follow: tpush(lp , lh), tpop(lp , lh) and B(l , delta , iter , lp , lh). The first, tpush(lp , lh), is used to store the turtle state in a stack, tpop(lp , lh) to retrieve the turtle state and B(l, delta, iter, lp, lh) is the recursive function.

Concept of stack

Stack is a fundamental construct of computer science. Basically, a stack is a collection of elements that can be accessed by means of two operations: *put* to add new elements and *pop* to retrieve elements, removing them from the stack. Stacks have a LIFO (Last In First Out) data structure: the last stored element is retrieved first. Imagine a stack of paper: the last piece put into the stack is on the top, so it is the first one to come out. Adding a piece of paper is called *pushing*, and removing a piece of paper is called *popping*.

Besides stack there is the concept of queue, which is about storing information as well, but the access is different. Queue has a FIFO (First In First Out) data structure: the first stored element is retrieved first. Imagine a queue at the store. The first person in line is the first person to get out of line. A person getting into line is "enqueued", and a person getting out of line is "dequeued".

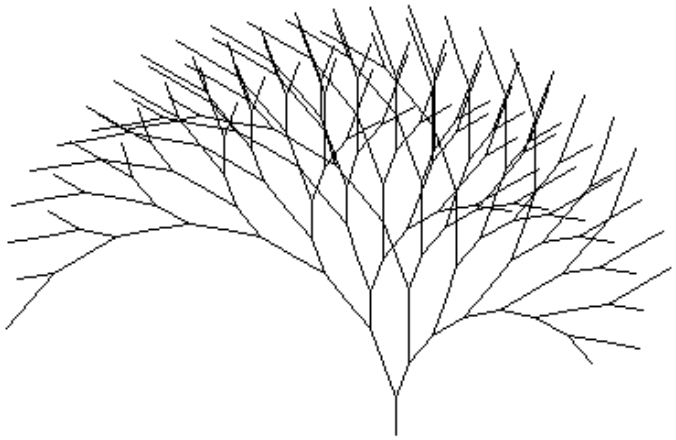
In our case we need a LIFO data structure, to build branch structures. Data are turtle states, another powerful concept we have already mention at page 35. Therefore, each turtle state element is composed by three numbers: two coordinates for the turtle position in the plane and an angle for the turtle pointing direction[a]. State stacks are useful because, every time the turtle has finished drawing a branch it must remember the state it had just before beginning the branch. It is a pretty good opportunity to experience both the state of a system and the stack concepts in an easily understandable circumstance.

[a]We refer here to the 2-dimensional turtle. If we plan to send the turtle in three-dimensional space then we have a total of 6 numbers: three coordinates for the turtle position in space and three angles for turtle orientation. I'm not considering the three-dimensional case here. I shall do this in the next version.

The use of the tpush() and tpop() function allows us to think in terms of a unique state stack. In reality the stacks are two: lp for the positions and lh for the headings — that's why lp and lh are passed to tpush(lp , lh), tpop(lp , lh) as arguments.

The recursive process is determined thoroughly by the B(l, delta, iter, lp, lh) function (instructions 24-34). The function takes 5 arguments: l is the length of the forward F and f steps; delta is deviation angle of the + and - rotations — these are parameters influencing the final aspect of the plant; iter is the number of recursions, lp and lh are the postions and headings stack, respectively. Two kind of actions can take place in this function, depending on iterations number. If the iteration counter is equal to 1, i.d. we have reached the bottom of the recursive process, then a last F step is made and the function is exited. It is important to realize that this is the only exit way from the function, meaning that each call to B() remains pending till the recursion bottom is reached. If the iteration number is different from 1 the production rules are applied. The seven instructions between lines 28 and 34 correspond biunivocally to the seven characters $F[-B]+B$ of the first production rule, $B \rightarrow F[-B]+B$ as specified in the comments at the end of these lines. Finally, the second production rule, $F \rightarrow FF$, translates in the fact that the step length is halfed in every function call: B(l/2, delta, iter, lp, lh) (instructions 31 and 34); this is what provides progressively smaller branches in our tree.

Let us see now how different shapes can be obtained by playing with the tree L-system. The following one is a variation of the previous stick tree which is that shown on the cover of Turtle Geometry [Abelson and diSessa, 1980, pag. 85].



This tree is obtained with the following code:

```
# Stick tree "plant" Abelson version
# with L-system
# 18.6.2022

# Axiom: B
# B -> F[-FB][+B]

from gturtle import *

makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
        setPos(lp.pop())
        heading(lh.pop())

def B(l, delta, iter, lp, lh):
    if iter == 1:
        forward(l)
    else:
        forward(l)                  # F forward
        tpush(lp,lh)                # [ beginning right B
        left(delta)                 # + left rotation
        forward(l)                  # F forward
        B(l, delta, iter-1, lp, lh) # B recursive branch call
        tpop(lp, lh)                # ] closing right B
        tpush(lp,lh)                # [ beginning right B
        right(delta)                # - right rotation
        B(l, delta, iter-1, lp, lh) # B recursive branch call
        tpop(lp, lh)                # ] closing right B

l = 20
delta = 20
iter = 8

lp = []
lh = []

B(l, delta, iter, lp, lh)
```

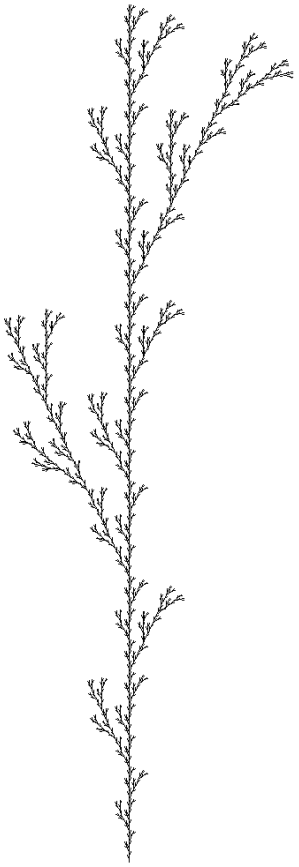
Listing 6.9: Basic recursive function for the L-system stick tree

This asymmetrical version of the stick tree is given by a slightly different first production rule: $B \to F[-FB]+B$ instead of $B \to F[-B]+B$; moreover, in this variant the second production rule is simply dropped, meaning that we have no branch length reduction throughout the recursive process.

In the following example a grass-like plant is simulated by means of some more substantial variations of the production rules:

```
# Grass "plant"
# with L-system
# 18.6.2022

# Axiom: F
# Production rule: F -> F[+F]F[-F]F

from gturtle import *

Options.setPlaygroundSize(600, 1000)
makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
        setPos(lp.pop())
        heading(lh.pop())

def F(l, delta, iter, lp, lh):
    if iter == 1:
        forward(l)
    else:
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpush(lp,lh)                # [ beginning right F
        left(delta)                 # + left rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpop(lp, lh)                # ] closing right F
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpush(lp,lh)                # [ beginning left F
        right(delta)                # - right rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpop(lp, lh)                # ] closing right F
        F(l, delta, iter-1, lp, lh) # F recursive forward call

l = 4
delta = 25.7
iter = 6
setPos(0,-490)

lp = []
lh = []

F(l, delta, iter, lp, lh)
```

Listing 6.10: Basic recursive function for the L-system stick tree



This kind of grass is obtained by means of the following tree L-system:

$$\begin{cases} \text{Axiom: } F \\ \text{Production rule: } F \to F[+F]F[-F]F \end{cases} \quad (6.5)$$

Also in this case we have just one production rule, as in the previous one, i.e. there is no scaling during recursions. However both the axiom and the production rule look different: the axiom is given by an F element and no "no operation" B elements appear in the production rule.
Which is the difference in using F or B as the axiom? To understand better, let us delve into the role of the axiom for a moment. In this example the axiom is F, this means that the substitutions specified by the production rule have to be applied to all the F occurrences, at any iteration.

This is better illustrated in the following picture.

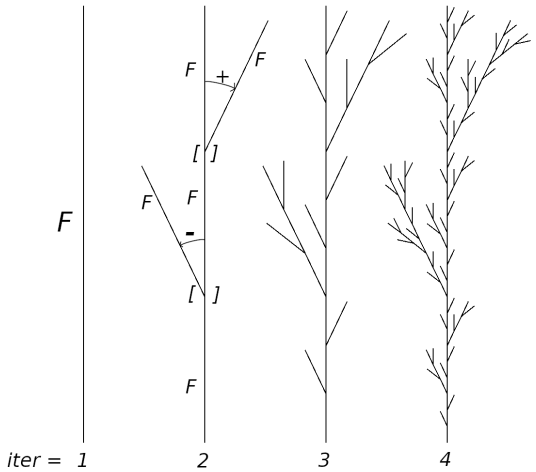


Figure 6.2: First four recursions of grass fractal simulation. Images are scaled so that overall grass height is the same. Comparing the first and second images we see the graphical expression of the production rule.

Whereas it is rather straightforward to see what one means by substituting $F4$, what could it mean to substitute B instead, the "no operation" label? Let us look at the three first iteration of the simple tree construction process.

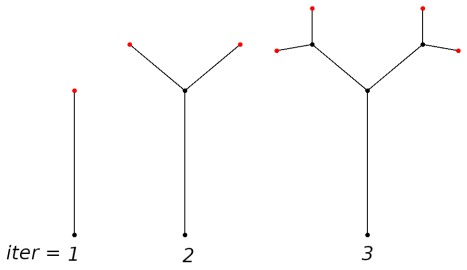


Figure 6.3: The first three recursions of the fractal grass simulation. The idea of this image is to show which "buds" are about to flower (the red ones) and which (black ones) have already flowered and given rise to branch bifurcations.

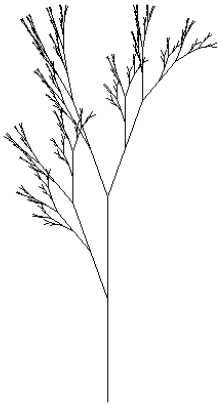
With this picture in mind is probably easier to grasp the meaning of a "no operation" label B in the simple stick tree listing at page 134, whose tree L-system is the following onew:

$$\begin{cases} \text{Axiom: } B \\ \text{Production rule: } B \rightarrow F[-B][+B] \\ \text{Production rule: } F \rightarrow FF \end{cases} \tag{6.6}$$

The stick tree is all about bifurcations. Actually, the stick tree is made just by bifurcations, with progressively smaller branches. Each "B-point" is nothing else than a bifurcation, with two branches stemming from it. The axiom is B since the process is started in instruction N.46 with a call to B(l ,delta ,iter ,lp ,lh). The first production rule, $B \rightarrow F[-B][+B]$, tells what to substitute to every B occurrence. The second production rule, $F \rightarrow FF$ tells whato to substitute to F in the previous rule — this second rule tells how to reduce branch length at each recursion level.

Here we have another example of grass simulation, which is quite different from that shown at page 136 obtained by means of bifurcations:

$$\begin{cases} \text{Axiom: } B \\ \text{Production rule: } B \to F[+B]F[-B] + B \\ \text{Production rule: } F \to FF \end{cases} \tag{6.7}$$

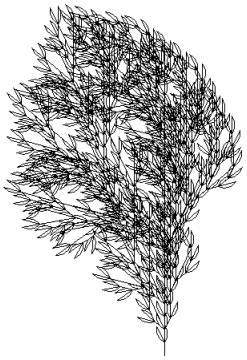


```
# Stick tree "plant"
# with L-system
# 24.5.2022

# Axiom: B
# B -> F[+B]F[-B]+B
# F -> FF

from gturtle import *

makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
        setPos(lp.pop())
        heading(lh.pop())

def B(l, delta, iter, lp, lh):
    if iter == 1:
        pass
    else:
        forward(l)                          # F forward
        tpush(lp,lh)                        # [ beginning right B
        left(delta)                         # + left rotation
        B(l/2, delta, iter-1, lp, lh)       # B recursive branch
    call
        tpop(lp, lh)                        # ] closing right B
        forward(l)                          # F forward
        tpush(lp,lh)                        # [ beginning left B
        right(delta)                        # - right rotation
        B(l/2, delta, iter-1, lp, lh)       # F recursive branch
    call
        tpop(lp, lh)                        # ] closing right B
        left(delta)                         # + left rotation
        B(l/2, delta/2, iter-1, lp, lh)     # F recursive branch
    call

l = 100
delta = 20
iter = 7
setPos(0,-280)

lp = []
lh = []

B(l, delta, iter, lp, lh)
```

Listing 6.11: Basic recursive function for the L-system stick tree

Finally the simulation of a bush:

$$\begin{cases} \text{Axiom: } B \\ \text{Production rule: } B \to F[+B]F[-B] + B \\ \text{Production rule: } F \to FF \end{cases} \tag{6.8}$$



```
# Bush
# with L-system
# 24.5.2022

# Axiom: F
# Production rule: F -> FF+[+F-F-F]-[-F+F+F]

from gturtle import *

makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
#         setPos(lp.pop())
        moveTo(lp.pop())
        heading(lh.pop())

def F(l, delta, iter, lp, lh):
    if iter == 1:
        forward(l)
    else:
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        left(delta)                 # + left rotation
        tpush(lp,lh)                # [ beginning right F
        left(delta)                 # + left rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        right(delta)                # - right rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        right(delta)                # - right rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpop(lp, lh)                # ] closing right F
        right(delta)                # - right rotation
        tpush(lp,lh)                # [ beginning left F
        right(delta)                # - right rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        left(delta)                 # + left rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        left(delta)                 # + left rotation
        F(l, delta, iter-1, lp, lh) # F recursive forward call
        tpop(lp, lh)                # ] closing right F

l = 8
delta = 25
iter = 5
setPos(0,-200)

lp = []
lh = []

F(l, delta, iter, lp, lh)
```

Listing 6.12: Basic recursive function for the L-system stick tree

Thus we have seen various examples from which the reader can invent new ones, playing with the elements of the production rules. At first glance this seems a somewhat complex topic to be proposed at school, but once patiently introduced and the process well clarified, it should be possible to use it at the secondary school level. The result is a very rich activity that combines two languages with different levels of abstraction: the Python language for computer programming and the formal L-system language for creating plant forms. This is an activity that lends itself to practicing discovery, mixing reasoning and tinkering. Here is a table that helps translate the execution of L-system elements into Python instructions.

Label	Instruction
F	forward(l)
f	penUp() forward(l) penUp()
$+$	left(delta)
$-$	right(delta)
[	tpush(lp,lh)
]	tpop(lp,lh)
B	recursive call B(...)

Table 6.3: Where l is the step length and delta is the deviation angle, determined at the beginning once for all.

6.2.2 The Cantor powder

In the above table there is one element, f, which we did not use in the examples. It is the element that performs a step but without drawing anything, a "dumb" step. We illustrate its use in an example that is not about plant simulation but about a classic of the "monster math literature": the Cantor set. George Cantor, the father of set theory, lived between 1845 and 1918, so well before Benoit Mandelbrot introduced the concept of a fractal in the 1960s. But that's right: there were several fractal objects circulating even before that but they were considered mathematical monstrosities that were the exception rather than the rule. The construction of Cantor's set is very simple. It starts by drawing a horizontal line. Then this is divided into three equal parts of which only the first and last are drawn, leaving a gap in between. And so below each drawn line is in turn subjected to the same process. Thus it can be guessed how in this way a large number of spaces are created with a series of scattered dots: the "Cantor dust" precisely. The process of creating the Cantor set is very simple to express with an L-system:

$$\begin{cases} \text{Axiom: } F \\ \text{Production rule: } F \to FfF \\ \text{Production rule: } F \to FFF \end{cases} \tag{6.9}$$

```
# Cantor
# with L-system
# 18.6.2022

# Axiom: F
# F -> FfF
# F -> FFF


from gturtle import *

Options.setPlaygroundSize(1600, 600)
makeTurtle()
clearScreen()
hideTurtle()
setPos(-250,0)
heading(90)

def C(l, iter):
    print(iter, l)
    if iter == 1:
        forward(l)       # Axiom: F
    else:
        C(l/3, iter-1)   # F
        penUp()
        forward(l/3)     # f
        penDown()
        C(l/3, iter-1)   # F

setPos(-500,200)
l = 1000

for iter in range(6):
    C(l, iter+1)
    setPos(-500,100-(iter-1)*50)
```

Listing 6.13: Basic recursive function for the L-system stick tree

This code produces the following picture:

With this example, we have added more cues for you to try to explore on your own. With this example, we have added more cues for you to try to explore on your own. It is amazing what can be created with this system. The paper of Prusinkiewicz et al shows some amazing example in three dimensions. It's really worthwhile to have a look. With this example, we have added more cues that you can try to explore on your own. With this example, we have added more insights that you can try to explore on your own. It is amazing what you can

create with this system. The paper by Prusinkiewicz et al [Prusinkiewicz et al., 1988] shows some amazing examples in three dimensions. It is really worth a look.

6.3 Fractals and randomness

The previous natural-looking shapes were created by working on fractal growth patterns. However, we can use another ingredient to simulate natural shapes, which is randomness. Let's take our first simple tree, which is quite geometric, and try to inject some "life" into it.

```
1  TO TREE LL
2    A = RANDOM 50
3    IF LL > 2 [
4      FORWARD LL
5      LEFT A
6      TREE LL*3/5
7      RIGHT A*2
8      TREE LL*3/5
9      LEFT A
10     BACK LL
11   ]
12 END
13
14 TREE 50
```

Listing 6.14: A livelier tree

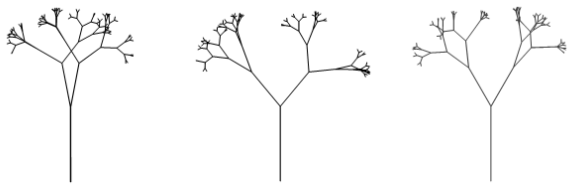


Figure 6.4: Three different executions of the previous code

The substantial difference with respect to the original code is in instruction N. 2, where the turning angle A is chosen as a random number between 0°and 50°.

We can try adding a random number between -2 and 2 to the step length - in this example 50 at the first iteration.

```
TO TREE LL
  A = RANDOM 50
  L = LL - 2 + RANDOM 4
  IF LL > 2 [
    FORWARD L
    LEFT A
    TREE L*3/5
    RIGHT A*2
    TREE L*3/5
    LEFT A
    BACK L
]
END

TREE 50
```

Listing 6.15: An even livelier tree

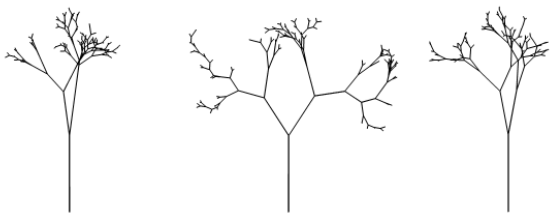


Figure 6.5: Three different executions of the previous code

It's fun to see what happens when you try to introduce a different amount of randomness into a fractal — a whole world of wonder and opportunity for reflection.

6.4 Mathematical and natural fractals: the impossible task of measuring the length of coastlines

An English meteorologist, Lewis Fry Richardson (1881-1953) [Richardson, 1961] faced the problem of measuring coastlines. He used a simple process to find the perimeter of a natural object by going around the coast with a ruler (yardstick) of a certain length. Doing this he discovered a puzzling fact: measuring a coastline is an impossible task. Or rather: it can be measured, but the method must be chosen with the specific practical objective in mind, because the result depends on how the measurement is made. For example, by decreasing the length of the ruler, the resulting perimeter will increase rather than converge to a precise value. This is counter-intuitive, since if we do the same with a geometrical shape we know that the result will converge to the perimeter of the figure. For instance, when applying Richardson method to a circle we can use a series of inscribed regular polygons. Each polygon is what we obtain by using a ruler equal to its side.

The circumference c of a circle of radius r is equal to $2\pi r$. The angle subtended by each side of an inscribed regular polygon with N sides is given by $\alpha = 2\pi/N$, and the side is $l = 2r \sin \alpha/2 = 2r \sin \pi/N$. Therefore the perimeter is $c_N = N2r \sin \pi/N$.

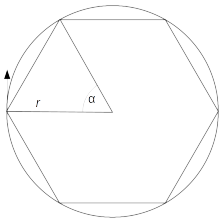


Figure 6.6: Hexagonal approximation of a circle. This figure was drawn with LibreLogo. The upward arrow shows initial position and direction of the turtle.

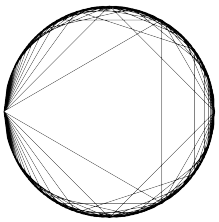


Figure 6.7: Regular polygons approximating the circle. The first 30 regular polygons are shown, starting with the equilateral triangle. Logo code in listing 7.13 at pag. 181.

The fact that polygons with increasing number of sides tend to approximate the circle is intuitive. The correct result is

$$\lim_{N \to \infty} N2r \sin \frac{\pi}{N} = 2\pi r \tag{6.10}$$

But what about measuring a coastline? Why shouldn't it be possible? Let's apply the Richardson's method to the coast of Britain[5]:

[5]The plots and the maps shown in this chapter have been obtained with a program written in R, a free software language for statistical calculation and processing of big data. The knowledge of this language is not among the objectives of this book. We put the code (listing 7.15) in appendix 7.3 (pag. 169)

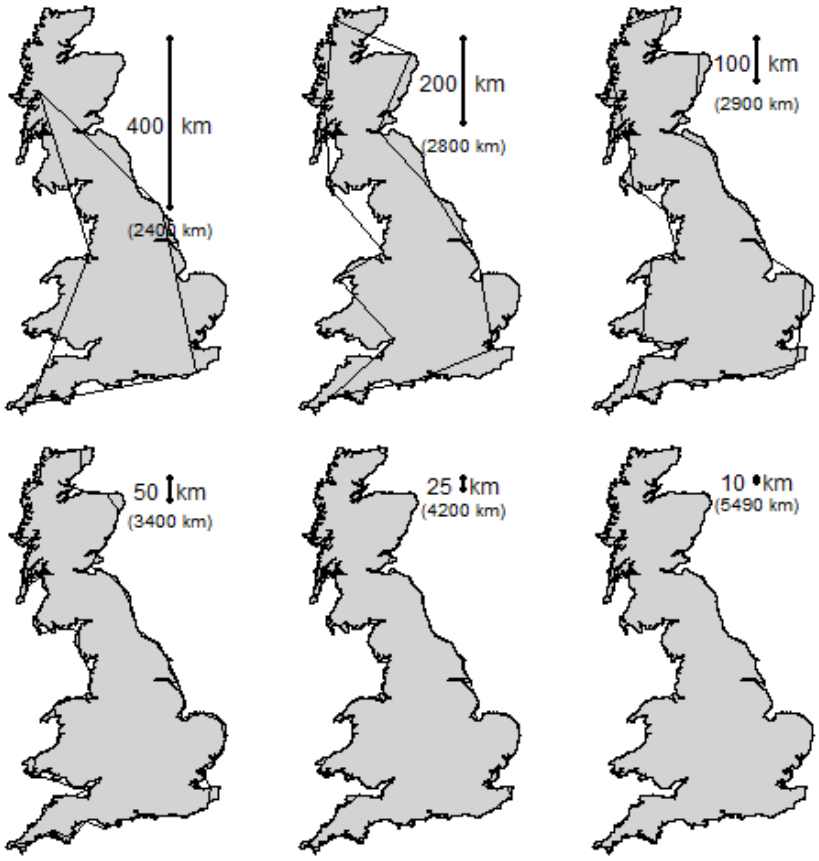


Figure 6.8: Meausure of Britain's perimeter with different rulers: 400, 200, 100, 50, 25 and 10 km wide, respectively. On each side of the map are shown the length of the ruler and, within the parenthesis, the estimated perimeter.

While the approximation improves considerably, the perimeter is steadily increasing at first glance. Let's trace the length of the perimeter for a wider range of rulers.

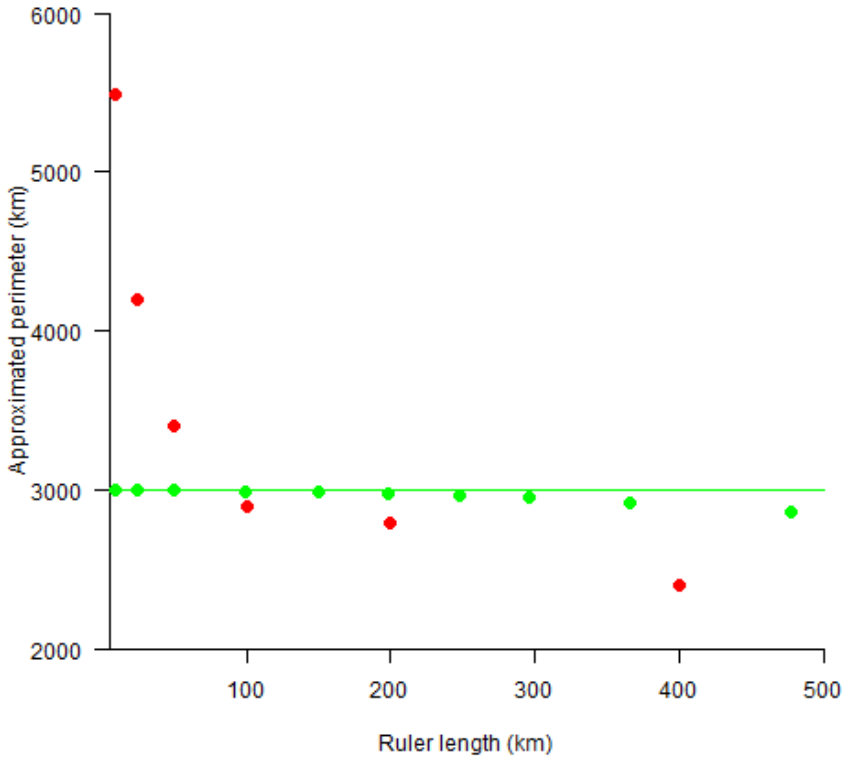


Figure 6.9: The red dots represent the estimated perimeter of Britain with rulers of 10, 25, 50, 100, 200, 400 km, from left to right, respectively. The green points represent the same estimates of the perimeter of a circle with circumference equal to 3000 km, using values of the rulers (side of the inscribed regular polygon) equal to 3, 4, 6, 8, 10, 12, 15, 20, 30, 60, 120, 300 km, from left to right, respectively. The green horizontal line represents the exact value of the perimeter of the circle, 3000 km.

Surprisingly, the same procedure, applied to Britain or a circular island of about the same size, gives very different results! Not only that, while in the case of the circular island the estimate of the perimeter tends asymptotically to the true value, that of Great Britain seems to explode, giving the impression of following rather a law of power, such as $1/x^s$. Power laws can be linearised

by plotting them in a log-log graph, where the slope of the resulting linear relationship equals the power coefficient s.

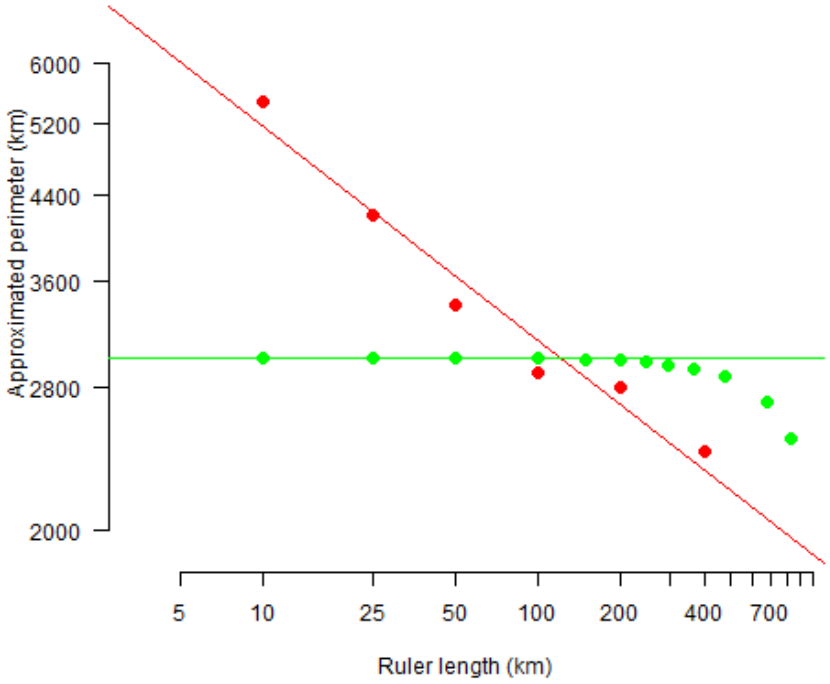


Figure 6.10: Again, the red dots represent the estimated perimeter of Britain with rulers of 10, 25, 50, 100, 200, 400 km, from left to right, respectively. The green points represent the same estimates of the perimeter of a circle with circumference equal to 3000 km, using values of the rulers (side of the inscribed regular polygon) equal to 3, 4, 6, 8, 10, 12, 15, 20, 30, 60, 120, 300 km, from left to right, respectively. The green horizontal line represents the exact value of the perimeter of the circle, 3000 km.

The data of Britain are fitted quite well by a straight line (red in the plot) whose slope turns out to be $s = -0.22$. We'll come back to this later.

So, which is the correct ruler? That was Richardson's finding, who was investigating how countries with common borders often reported different border

lengths[6]. So far the empirical results.

Not many years later, Benoit Mandelbrot (1924-2010), laid down the foundation of fractal geometry by reasoning on Richardson empirical finding, in a famous paper published in Science: "How long is the coast of Britain? Statistical self-similarity and fractional dimension" [Mandelbrot, 1967]. Let's go over his reasoning in a simplified way.

Let's take a segment of length 1 and broke it in three smaller parts, 1/3 long (Figure 6.11). Each of the smaller parts is deducible from the whole by a similarity of ratio 1/3, in general $r(N) = 1/N$, if N is the number of parts. In the plane, if we decompose a square of side 1 in equal sub-squares, each of side 1/3, we get 9 sub-squares. Again, if we decompose a cube of side 1 in equal sub-cubes, each of side 1/3, we get 27 sub-cubes. What we are finding here is a relationship between the number of elements N in which we divide a geometrical figure and the self-similarity ratio $r(N) = 1/N^{1/D}$, where D is the dimension of the space occupied by the figure: one-dimensional, two-dimensional or three-dimensional for the line, the square and the cube, respectively:

[6]Richardson reported discrepancies in borders measurements while he was investigating reasons of conflicts that resulted in war. He spent a great deal of work on peace and conflict studies [Richardson, 1961], probably motivated by the fact that he lived through the terrible events of World War II.

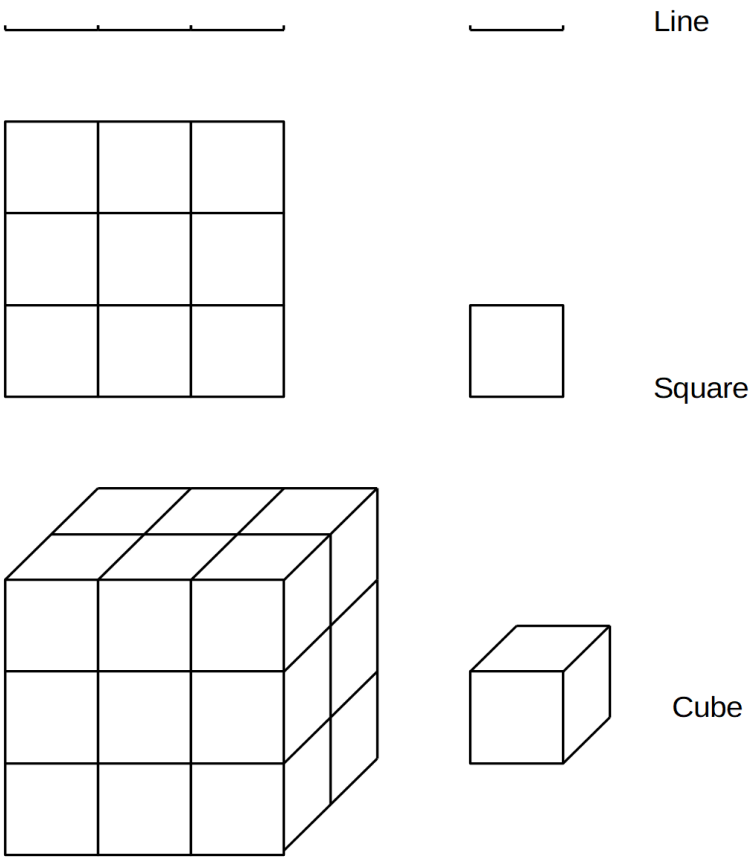


Figure 6.11: The concept of self-similarity in relationship with the dimension of simple geometrical structures.

Object	Number of parts	self-similarity ratio
line	3	$1/3$
square	9	$1/9^{1/2}$
cube	27	$1/27^{1/3}$

Pretty familiar, so far. But now let's do what mathematicians love to do:

generalize. First, focus on the relationship of self-similarity:

$$r(N) = 1/N^{1/D} \tag{6.11}$$

and then derive the dimension D:

$$D = -\frac{\log N}{\log r(N)} \tag{6.12}$$

Here we can take a leap of imagination. This last property of the quantity D means that it could be also evaluated for other figures that can be exactly decomposed into N parts, such that each of the part is deducible from the whole by a similarity of ratio $r(n)$. If such figures exist, they may be said to have $D = -\log N/\log r(N)$ for dimension, regardless of the fact that it may result in a non-integer value.

Math: thinking outside the box

Do such figures exist? The ability to think outside the box is the essence of mathematical creativity and this is a good example. The self-similarity relationship 6.11 has provided a solid picture, dependent on the convenient notion of size, which can be worth one, two or three; how could it be otherwise? Yes, we know that scientists can talk about space with more dimensions — think of Einstein's four-dimensional space-time. But in any case, the idea is always to describe dimensionality by means of integers, even when it comes to solving mathematical problems that require thinking in terms of spaces with many thousands of dimensions — for example, the production of medical images requires the solution of this sort of problem. Our decisions in the face of the new are always the result of a compromise between curiosity and fear of the unknown. Often, the feeling of discomfort in the territory of the new leads us to prefer the security of a familiar refuge, but in this way no growth is possible!

Mathematics requires imagination. But to get out of the box you have to be in the right mood, you have to feel good. The opposite of the mood that is created by performing automatic exercises whose meaning is not understood, in competitive contexts and with time constraints: they are poisons that prevent the development of mathematical thinking and thinking in general. And this is also what neurosciences are telling us, with the emphasis on the emotional nature of cognitive processes we have already quoted [Dehaene, 2020]:

> Negative emotions crush our brain's learning potential, whereas providing the brain with a fear-free environment may reopen the gates of neuronal plasticity.

You have to be very curious but also very confident to let the D variable float, free to assume non-integer values. When exploring outside the box you need to get rid of the need to understand everything at once, at all times. Instead, you have to accept the idea that it takes some time to familiarize yourself with new scenarios. In our example, the first step is

to accept the idea that the D value may not be integer, but this does not imply that you understand the meaning of this step right away. However, taking time and playing with the iterations of Koch's process, in the end the idea may emerge that a very circumvoluted line occupies a *significant portion of the plane* and that at the limit it is something hybrid between a line and a flat figure, characterized by a dimension between 1 and 2. Such a mental process belongs to the domain of intuition and lies at the basis of mathematical creativity. The theoretical systematization comes later. Mandelbrot's essay "How long is the coast of Great Britain? Self-similarity and fractional dimension statistics" [Mandelbrot, 1967] does not report a complete theory on fractional dimensions (which would be the Hausdorff dimension to be precise) but the essential elements of primitive intuition.

When you learn mathematics, you obviously do not always expect to have insights of the caliber of Mandelbrot's. But every new mathematical notion should be acquired in this sort of mood, and in order to do this you have to feel good and to feel good about yourself, first of all.

To show how such strange curves exist that they are associated with a fractional dimensionality, Mandelbrot suggests to study the Kock curve. The Koch curve is constructed as the limit of an iterative process. First, at step 0, we start with a segment, say of length 1. Step 1 is to draw a kinked curve made up of 4 segments of length 1/3 in the following way:

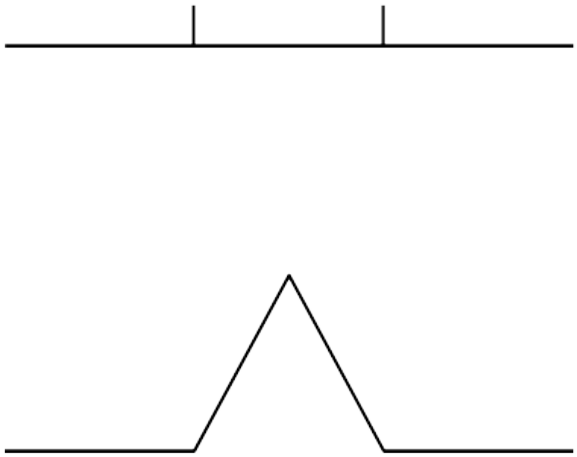


Figure 6.12: Steps 0 and 1 of Koch' curve construction process.

Here we have decomposed a figure, i.e. a segment of length 1, in four elements ($N = 4$), each reduced by a factor $R(N) = 1/3$. The process is repeated at the successive steps:

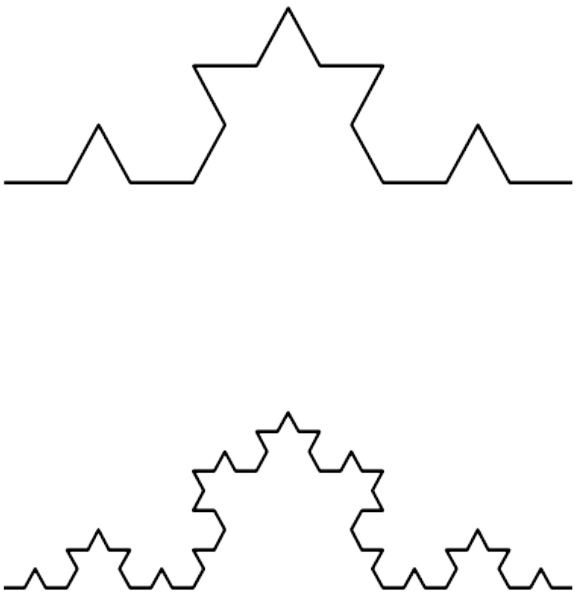


Figure 6.13: Steps 2 and 3 of Koch' curve construction process.

As long as the process goes further, the curve becomes more detailed. However, no particular improvement is seen beyond step 7, since the details are too dense for the resolution and the line is so convoluted that it appears just as a thicker, less detailed line.

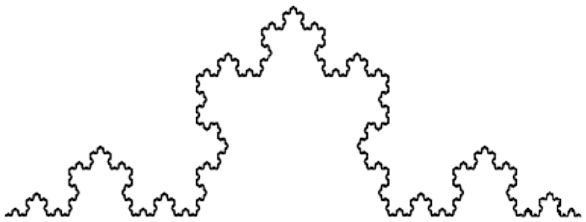


Figure 6.14: Step 7 of Koch' curve construction process.

Thus, even if we cannot see it, we can imagine that, while the iterative process progresses, the line "occupies" an increasing amount of space, within the thick line we see at step 7.

That said, we are ready to give a meaning to equation 6.12. In fact, if we subtitute $N = 4$ and $r(N) = 1/3$, we get $D = -\frac{\log 4}{\log 1/3}$, i.e.

$$D = \frac{\log 4}{\log 3} = 1.26 \tag{6.13}$$

Since we started with the familiar one-, two- or three-dimensional concept, the idea of a fractional dimension is still weird but our reasoning about a line wich is so convoluted to occupy "a lot of space" may match the idea of a fractional dimension: not so thin as a line, not so thick as a full two-dimensional figure. Well, this is a typical feature of fractals object: fractals have fractional dimensions. The Koch curve is a fractal with dimension $D = 1.26$.

Koch's curve looks like a coast and that's why we introduced it here. We started talking about the coast of Great Britain, now we consider the island of Koch.

In order to build the island we have to browse the code we used to draw the previous examples. The KOCH recursive function requires the length, FF, of the starting segment and the number of desired steps, ITER, as input data. At each step, the function is called three times, rotating appropriately according to the shape of the Koch curve; that is, traveling from left to right: first segment, turn 60°left, second segment, turn 120°right, third segment, turn 60°left, fourth segment.

```
1  TO KOCH FF ITER
2     IF ITER = 1 [
3        FORWARD FF
4     ] [
5       KOCH FF/3 ITER-1
6       LEFT 60
7       KOCH FF/3 ITER-1
8       RIGHT 120
9       KOCH FF/3 ITER-1
10      LEFT 60
11      KOCH FF/3 ITER-1
12    ]
13 END
14
15 RIGHT 90
16 KOCH 150 7
```

Listing 6.16: Logo code to draw the approximation at step 7 of Koch curve limit process'.

The segments are reduced to 1/3 with respect to the previous iteration. At each step, the ITER parameter is decreased by one, so that only when ITER reaches the value of 1 the recursive process halts and a segment is actually drawn. Thus, at step k, the curve is composed only by segments $(1/3)^k$ long. The KOCH function is defined between instructions 1-13. The function is actually executed at instruction 16. The "RIGHT 90" instruction 15 determines the orientation of the curve — this comes in handy for building an island.

The trick is really simple since we are going to draw a sort of baroque equilateral triangle, using Koch curves in place of straight lines.

```
1 REPEAT 3 [
2    KOCH FF 5
3    LEFT 120
4 ]
```

Listing 6.17: Snippet of Logo code to draw step 5 of Koch island'.

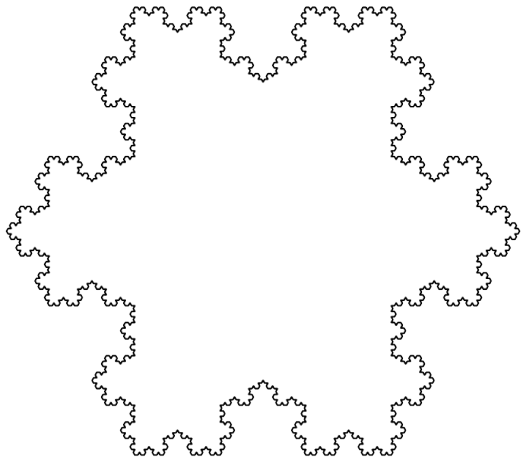


Figure 6.15: Step 5 of Koch island.

Now the fun begins: How long is the coast of Koch island? The mathematical derivation is simple. Let's focus on the Koch curve, which is only one of the "baroque sides" of the island. If the initial length of the segment at step 0 is 1, the length at step 1 will be four times one third, which is 4/3, since each

of the four subsegments is 1/3 long. At step 2 the process is repeated for each sub-segment, so $4(1/3)(4/3) = (4/3)^2$. So, on to step n, where the length will be $(4/3)^k$. Therefore, as long as n increases, the length will increase more and more. Actually we have that

$$\lim_{k \to +\infty} a^k = \infty, \tag{6.14}$$

provided that $a > 1$, as it is the case with 4/3. Of course, the result easily extends to the perimeter of the island of Kock, as the sum of infinite sets is also infinite:

$$\lim_{k \to +\infty} 3(4/3)^k = \infty. \tag{6.15}$$

So what? The perimeter of Koch's island is infinite? Is there something wrong? Let's try to investigate further by calculating the area enclosed by this perimeter.

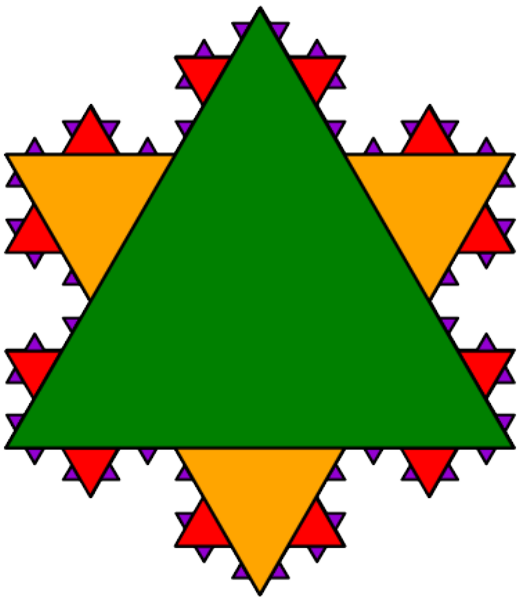


Figure 6.16: Area calculation of the Koch island at iteration 3. At the beginning the island is just an equilateral triangle (green). At the first step three smaller triangles (orange), scaled by 1/3 are added. At the second step 3×4 smaller triangles (red), at the third step $3 \times 4 \times 4$ smaller triangles (violet) and so on.

At the beginning we have just an equilateral triangle (in green). If the side is l, its area is $A_0 = l^2\sqrt{3}/4$. At step 1 we have to add three triangles (in orange) scaled by 1/3; at step 2 the new triangles (in red), smaller by 1/3, are 3×4; at step 3 we have $3 \times 4 \times 4$ new triangles (in violet) and so on. In general, at step k, the number of sides will be $n_k = 3 \times 4^{k-1}$. The sides of the smaller triangles are derived by successive downscaling by a factor of 1/3, i.e. $l_k = l(1/3)^k$. Therefore, if A_k is the area of the approximation at step k, we have the following recurrence relation:

$$A_{k+1} = A_k + n_k l_k^2 \frac{\sqrt{3}}{4} = A_k + l^2 \frac{\sqrt{3}}{12} \left(\frac{4^{k-1}}{9^{k-1}} \right). \tag{6.16}$$

We can expand this expression in this way:

$$A_{k+1} = A_0 + l^2 \frac{\sqrt{3}}{12} \left(1 + \frac{4}{9} + \frac{4^2}{9^2} + \cdots + \frac{4^{k-1}}{9^{k-1}} \right). \tag{6.17}$$

We recognize here that the expression within brackets is the partial sum of the first k terms of the geometric series $\sum_n \left(\frac{4}{9}\right)^n$. Considered that

$$\sum_n \left(\frac{4}{9} \right)^n = \frac{1}{1 - 4/9} = 9/5, \tag{6.18}$$

we obtain the following value for the area of a Koch island generated by the equilateral triangle of side l:

$$A = A_0 + l^2 \frac{\sqrt{3}}{12} \frac{9}{5} = l^2 \frac{\sqrt{3}}{4} + l^2 \frac{\sqrt{3}}{12} \frac{9}{5} = l^2 \frac{2}{5} \sqrt{3}. \tag{6.19}$$

This confirms the strange result: a finite area is enclosed by an infinitely long perimeter. Apparently, here we are faced with a new concept of perimeter. Actually, our idea of perimeter is deeply linked to the classic figures of geometry: squares, circles and so on. But here we are dealing with a coast, or something very similar, like the coast of Koch's imaginary island. And what is the fundamental difference? The details. The coasts have much more details than the outline of regular geometric figures. This result encourages a reflection on the concepts of finite and infinity, an intuitive one but useful for facing basic concepts of mathematical analysis in the future.

Finite and infinity

The Koch island is a mathematical abstraction: we can draw whatever intermediate step we want but not the final result, which is a mathematical limit. The figure obtained at each step is composed by segments, although progressively smaller, and the total number of them is always finite, although increasing. Thus, the perimeter of each intermediate approximation is finite. The infinity comes through the mathematical limit process. This means that we cannot draw the true Koch curve, nor we can represent it in digital form. Exactly as it happens with irrational numbers, such as π, which can never be represented as a digital number, but only as an approximation. For instance, there is no advantage to go beyond seven iterations when drawing the Koch curve, since the graphical resolution of whatever imaging device doesn't allow to see the finest details of the curve. By playing with the number of iterations, reflecting upon the quantity of visible details, trying to obtained zoomed images to see more of them, realizing that you will never get the true curve but you can always add a next iteration to the previous one to improve the approximation, well all these reflections are useful to pave the way to the concept of limit, which will be fundamental for anyone who will face the study of mathematical analysis -

all STEM disciplines but also many others in the domain of social sciences. And for the others they will be useful reflections to broaden the mind.

Therefore, it seems that, from a geometric point of view, Great Britain has much in common with the island of Koch, despite its abstract (and rather crazy) nature, rather than with some classical geometric shape. Since the Koch curve, at each step, N, is composed of segments, we can assimilate these to the rulers used by Richardson to calculate the lengths of the coastlines. In this way we can add the island of Koch to the previous graphical comparison between Great Britain and the circular island:

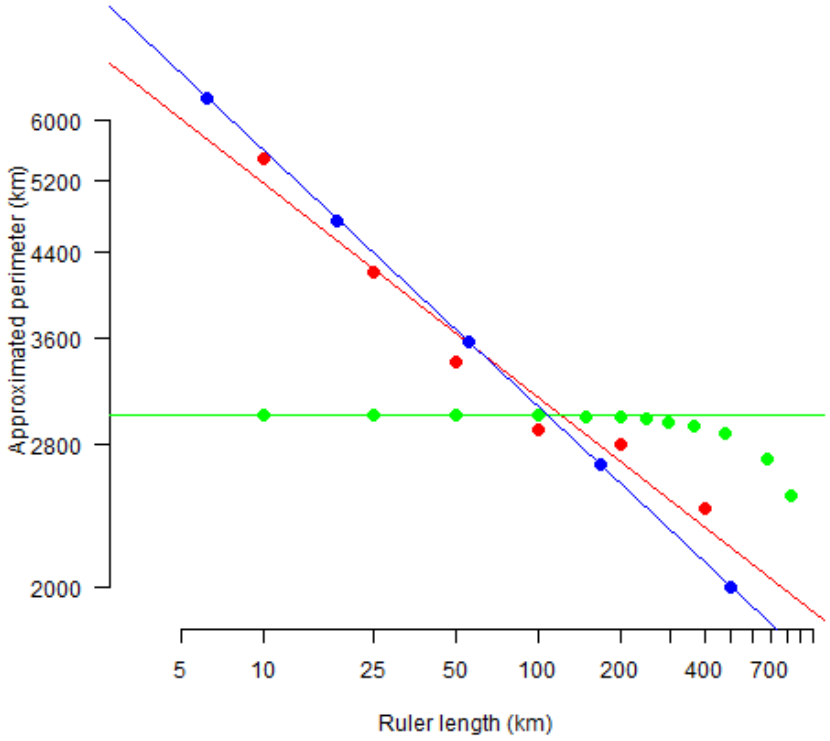


Figure 6.17: This is the same plot of figure 6.10 (page 149) but with Koch island data added. Red points are relative to Britain, green ones to circular island and blue ones to Koch island. Overall size of Koch island was adjusted to roughly fit Britain's value.

The slopes for the Britain and Koch curves are similar. We have seen before that the estimation derived from the statistical fit of the Britain curve gave

$s = -0.22$. Which is the slope of the Koch curve? This can be estimated by means of direct calculations. For each step k, i.e., for each point plotted for the Koch curve, we have the ruler length $l_k = (1/3)^k$ on the abscissa and the perimeter length $L_k = (4/3)^k$ on the ordinate. Now, lets's derive L_k in function of l_k. From these definitions we can write

$$L_k = \left(\frac{4}{3}\right)^k = 4^k \left(\frac{1}{3}\right)^k = 4^k l_k. \tag{6.20}$$

Considered that $D = \log 4/\log 3$ (equation 6.13 at page 155) and using the identity[7] $4 = 3^{\frac{\log 4}{\log 3}}$ we have

$$L_k = \left(3^{\frac{\log 4}{\log 3}}\right)^k l_k = 4^k \left(\frac{1}{3}\right)^k = 4^k l_k = 3^{Dk} l_k. \tag{6.21}$$

Since $l_k = (1/3)^k$, we can write

$$L_k = 3^{Dk} \left(\frac{1}{3}\right)^k = \left[\left(\frac{1}{3}\right)^k\right]^{-D} \left(\frac{1}{3}\right)^k = l_k^{1-D}. \tag{6.22}$$

This result is interesting because it establishes the relationship between the fractal dimension D of the perimeters and the slopes of the respective length-ruler curves, since $s = 1 - D$. This equation allows us to say that the value of the slope in a length-ruler plot is a measure of how *strange* is a curve with respect to regular geometrical shapes: as long as details are increasing, the slope s increases (in absolute value) as well and, consequently, the dimension D becomes larger with respect to 1.

Another interesting point is this: does this result allow us to say that the coasts are fractal? Not exactly. Yes, previous plots suggest that Britain and the island of Koch have something in common. However, we must bear in mind that the two lines are of a very different nature: the approximation of Koch is obtained by means of a well-defined mathematical process obtained by iterative reproduction of a basic form and, at each step, the application of a reductive scale factor, which is the definition of fractal. The coast of Great Britain (or any actual island) is instead shaped over time by a complex combination of many factors. Therefore, in this case the details have a statistical nature. Thus, natural coastlines (and many other natural forms) have a fractal nature, along a wide range of scales perhaps, but they are not fractal in the mathematical sense.

Uncertainty: a crucial ingredient in 20th century science

Since the advent of mass schooling, disciplines are taught separately. And even today, little or no effort is devoted to the big picture, and thus to the connections between the different fields. This means that the areas of intersection, which are often those where knowledge grows fastest, remain completely excluded from the average education of young people. As a result, the scientific picture is almost completely missing a crucial achievement of the 20th century: the intimate interweaving of certainty

[7] To prove the identity: starting with obvious equality $\log a \log b = \log b \log a$, we can write $\log a^{\log b} = \log b \log a$, and therefore $a^{\log b} = b^{\log a}$

and uncertainty that characterizes all areas of science.

In chapter 1, section 1.3, we highlighted this problem with Morin's words, when he is talking about the "School of Mourning". We have mentioned several science areas where unpredictability plays a critical role. But nothing or almost nothing of this comes to school. It can be argued that these topics require too complicated treatment. Apart from the fact that suitable and instructive narrative solutions can almost always be found, there are actually simple examples. The measurement of the perimeter of an island is an example of how trying to solve a seemingly trivial problem one can suddenly find oneself in the domain of uncertainty. It is an interesting example because it can be tackled with mathematical skills within the reach of an upper secondary school student. It is also interesting because it includes various areas and explorations, some of which are only mentioned in this description. An interesting diversion could be done already at the beginning: why did Richardson at some point come to measure the perimeter of Great Britain? The reason for Richardson's interest was not only a yearning for knowledge *per se*, but a combination of causes, including the exceptional historical circumstance, a vision of what humanity should be and an inner emotional impulse. Richardson was not only a proficient scientist, but also a well known pacifist and the tragic events of the World War II prompted him to explore the reasons of many quarrels among nations, for instance about coastlines and border lengths. These kind of considerations could trigger an exploration of the historical and social context. Then, once we have exposed Richardson's measurement method we posed the problem of doing the same with a circular region. This is a problem of geometry, students could be asked to solve it. For example, using Logo (hints could be given on the basis of listing 7.13 at pag. 181 to reproduce figure 6.6). If the students already know the concept of limit, then the limit of a regular polygon perimeter to the circle can be discussed, since the number of sides tends to infinity. (equation 6.10). Dealing with the comparison between the dependence of perimeter estimates from the ruler in the case of Britain and of the circular isle, there is the chance to discuss in depth the different nature of the plotted data: estimates calculated from experimental data versus theoretical values. Data in a plot can be of different sorts: students must learn to realise this, otherwise we will end up with citizens who cannot understand a graph in a newspaper. Then, when we come across a law of power, which appears in countless phenomena, and we reflect on its linearization by means of a log-log plot, students have the possibility to revise the logarithmic function. Guided by Benoit Mandelbrot, the father of fractals, we get a vivid example of what does it mean the reasoning out-of-the-box typical of mathematical thinking. And by tinkering with the Logo approximations of the Koch curve, we have the opportunity of diving into the recursive and the autosimilar nature of fractals. The determination of the Koch curve length entails the application of geometrical series approximations, but, most important, it allows to reflect deeply on the relationship between infinity and finite — crucial insights for future STEM students.

Measuring coastlines: a seemingly trivial problem with profound consequences about the relationship between certain and uncertain, if proposed

in a sensible way at school.

Chapter 7

Appendices

7.1 Appendix 1: Index of powerful ideas and relevant reflections

This is an index of the gray text boxes we added to point out powerful mathematical ideas, concepts or approaches that are evoked by the exercises. Concepts that have not to be explicitly described to the students but that teachers should know, in order to understand the potential of the practices and their crucial points.

Pag. 29 — Syntonic learning: examples of practices

Pag. 30 — Isomorphism

Pag. 32 — Breaking the problem down into smaller parts: *Divide et impera*

Pag. 35 — State of a system

Pag. 37 — The process of building scientific knowledge: the espistemological issue

Pag. 40 — Encapsulating functionality in new commands: modular thinking

Pag. 42 — Again on modular thinking while drawing the house

Pag. 45 — The door to algebra

Pag. 48 — Syntonic learning described through a dialogue

Pag. 49 — Differential equations

Pag. 56 — Linear and exponential growth

Pag. 56 — Self-similarity → fractals

Pag. 62 — Concept of "law"

Pag. 63 — Concept of integration

Pag. 63 — The limits of the machine (and of theory): how can the execution of a program unintentionally become a never-ending story?

Pag. 64 — Trying to find a stopping condition for a (simple) program

Pag. 70 — Successive approximations

Pag. 91 — Randomness in science

Pag. 114 — Multitasking: the computer juggling trick

Pag. 99 — Feedback in nature

Pag. 105 — Scope of a theory

Pag. 111 — Scalar and vector fields

Pag. 74 — Physics — Initial conditions

Pag. 74 — Physics — Computational vs algebraic approach

Pag. 159 — Fractals: infinity in finite $\rightarrow$ towards calculus

Pag. 161 — Uncertainty: a crucial ingredient in 20th century science

7.2 Appendix 2: How to manage graphics in Writer

But what does it mean to use Logo within a *word processor* like Writer, considered that this is a normal word processor while Logo is a kind of a drawing language? Simple: With the LibreLogo toolbar you can produce images that are integrated into the document as if they were imported. It's a brilliant idea, due to Németh László, who reproduced the features of Logo within LibreOffice. In reality, he further improved them, taking advantage of the Python language, with which he wrote the plugin. Using LibreLogo is very simple: you open a document in Writer, you write some code in Logo language, as you would write any other text, and then you run it by pressing the appropriate button in the LibreLogo *toolbar*; if the code is correct, the turtle executes the code drawing a figure in the text, right in the middle of the page.

This graphics can then be managed like any other LibreOffice graphics. The interaction between LibreLogo and Writer is particular for graphics. It may seem cumbersome at first but you actually have to get used to it and learn two or three rules. The probably unique feature of LibreLogo is that by running [1] a script you get a graphic object in the same place where you have written the code, that's it in ODT document page. These objects are of "vectorial" type, that is, they are composed by a set of geometric objects. They are different from *raster* or *bitmap*, that consist of a matrix of pixel[2]. The graphic objects produced by LibreLogo are completely similar to those produced with the handwriting tools available in Writer, accessible through the special *toolbar*, under the menu item **View** → **Toolbars** → **Drawing**:

As such, drawings made with LibreLogo can be moved, copied, or saved like any other graphic object. One useful thing to understand is that such objects are often actually a composition of distinct objects. We will do many of them in this manual. To use them as a single object, use the grouping function, as follows: first, you delimit the region that includes the objects to group, by selecting *pointer*  in the drawing bar and then by outlining the desired rectangular box with the mouse and holding down the left button. Please note that the mouse cursor must be in the shape of an arrow and not the typical you have when inserting text, in the shape of a capital I, because this is where you insert text and not graphics. The fact that the graphic (and not textual) cursor is active is also understood by the fact that, at the same time, another toolbar is activated for controlling the graphics:

[1] In jargon, by "running a program" we intended to execute all its instructions. Today, with modern languages, programs are often called *script*. In general, a program is a complete software and maybe also very complex. A *script* tends to be a smaller, more specific fragment of code but these categories may overlap widely.

[2] A closer look at the distinction between bitmap and vector images can be found at http://https://iamarf.org/2014/02/23/elaborazione-di-immagini-tre-fatti-che-fanno-la-differenza-loptis/

When you select the region containing the graphic objects, icons are activated in this bar, including the icon for the grouping function: . Pressing this will group all graphic objects in the selected region into a single graphic object that can be copied elsewhere or saved.

Another useful trick is to properly "anchor" the graphics to the document, where we have to use them. The key to determine the anchorage in the usual graphic bar is this: . By clicking on the arrow on the right of the anchor, you can select four anchor types: 1) "on page", 2) "in paragraph", 3) "in character" and 4) "as character". In the first case the graphics are associated to the page and do not move from it, in the second to a paragraph, in the third to a character and in the fourth case it behaves as if it were a character. What is the most appropriate anchorage is something that you learn from experience. Most of the graphics in this manual have been anchored "to the paragraph", except for small images that are in line with the text, as in the previous one, these are anchored "as a character".

These concern the management of graphics in Writer in general. Using LibreLogo, the only difference is that the graphics are produced through the instructions we put in the code. LibreLogo places the graphics in the middle of the first page of the document, even if the code text extends on the following pages. It may happen that the graphics overlap the text of the code itself. At first glance the result may be confusing and one can believe something wrong is going on. None of this. The graphics are produced to be used somewhere else. It is simply a matter of selecting it, as we have just described, and taking it elsewhere, in a clean page simply to see it clearly, or in some other document where it must be integrated.

7.3 Appendix 3: Programs listings

The complete commented listings of many programs used in the book are collected in this appendix. Only the pertinent chunks of code are reported within the text in order to make the reading more fluent. The programs can also be downloaded from the accompanying site and they are ready to be run, in Logo, some in LibreLogo and others in XLogo, while the python versions can be run in the TigerYjthon environments.

7.3.1 Modeling smell

This listing refers to the simple model of smell described in section 5.3. The turtle is constrained in a circular garden and its movement is disturbed by a certain degree of randomness.

```
# Turtle smelling model
# confined in circular region with random
# movement perturbations
# 26.2.2020

from gturtle import *
from random import randint
from math import *

# Define a circular region of radius r
# and make it green
def circle(r):
    c = r * 2 * pi
    s = c / 360
```

Listing 7.1: The turtle reach the salad by means of smelling, despite distractions, i..e some randomness.

```
    penUp()
    forward(r)
    right(90)
    penDown()
    setPenColor("green")
    setFillColor("lightgreen")
    startPath()
    repeat(360):
        forward(s)
        right(1)
    fillPath()

# Do one step and one turn if arrival
# point is inside the green area
# Otherwise keep turning right till
# arrival point is inside
def step(data, ll, aa):
    if checkStep(data, ll):
        forward(ll)
        right(aa)
    else:
        while checkStep(data,ll) == False:
            aa = aa + 10
            right(aa)

# Check if, given position and direction,
# arrival point is inside green area
def checkStep(data, ll):
    oldPos = getPos()
    oldHead = heading()
    penUp()
    hideTurtle()
    forward(ll)
    color = getPixelColorStr()
    if color == "white":
        setPos(oldPos)
        heading(oldHead)
        return False
    else:
        setPos(oldPos)
        heading(oldHead)
        showTurtle()
        penDown()
        return True

def arc():
    repeat(60):
        forward(1)
        right(1)

def leaf():
    startPath()
    arc()
    right(120)
    arc()
    right(120)
    fillPath()
```

Listing 7.2: The turtle reach the salad by means of smelling, despite distractions, i..e some randomness.

```
72
73 def salad(sPos):
74     oldPos = getPos()
75     oldHead = heading()
76     setPos(sPos)
77     setPenColor("darkgreen")
78     setFillColor("green")
79     left(90)
80     repeat(5):
81         leaf()
82         right(30)
83     setPos(oldPos)
84     heading(oldHead)
85
86 def dist(p1,p2):
87     return sqrt((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)
88
89 # Setup Python Turtle environment
90 setPlaygroundSize(800, 800)
91 makeTurtle()
92 hideTurtle()
93
94 r = 300
95 sPos = (-150, 180)
96 start = (150, -100)
97
98 circle(r)
99 heading(0)
100 salad(sPos)
101
102 l1 = 1
103 l2 = 5
104 a1 = -90
105 a2 = 90
106 data = (l1, l2, a1, a2)
107
108 # Place the Turtle and start setps loop
109 n = 1000
110 showTurtle()
111 setColor("darkgreen")
112 setPenColor("darkgreen")
113 setPos(start)
114 p1 = getPos()
115 p2 = getPos()
116 p2[1] = p1[1] + 1
117 count = 0
118 while True:
119     count = count + 1
120     d1 = dist(p1,sPos)
121     d2 = dist(p2,sPos)
122     right(randint(1,40) - 20)
123     if d2 > 10:
124         if d1 < d2:
125             aa = 20
126         else:
127             aa = 0
128         step(data, randint(l1, l2), aa)
129         p1 = p2
130         p2 = getPos()
131     else:
132         print(count, "iterations!")
133         break
```

Listing 7.3: The turtle reach the salad by means of smelling, despite distractions, i..e some randomness.

7.3.2 Facing light

This is the simplest of our sight models, where the Turtle is merely trying to keep a constant value of the bearing with respect to a light source (section 5.4.1).

```
# Sighted turtle keeping bearing
# towards(px, py) and bearing(px, py) tested in towards-test.py
# 14.3.2020

from gturtle import *
from random import randint
from math import *

# turns randians into degrees
def r2d(ar):
     return ar / pi * 180

# function towards(px, py)
# gives direction to point
# (px, py) from turtle position
# expressed as angle in degrees
def towards(px, py):
    pcor = getPos()
    dx = px - pcor[0]
    dy = py - pcor[1]
    if dy > 0:
        if dx == 0:
            return 0
        if dx > 0:
            return r2d(atan(dx/dy))
        else:
            return 360 + r2d(atan(dx/dy))
    if dy < 0:
        return 180 + r2d(atan(dx/dy))

# gives bearing angle of direction
# to point (px, py) with respect to
# turtle's heading
def bearing(px, py):
    h = heading()
    if h > 0:
        return towards(px, py) - h
    else:
        return 360 - (towards(px, py) - h)

# lets the turtle facing
# point (px, py)
def face(px, py):
    right(bearing(px, py))

# moves turtle from given position
# and heading keeping fixed
# bearing
def keepBearing(px, py):
    i = 0
```

Listing 7.4: The turtle moves toward a light source by keeping a constant bearing.

```
    while True:
        i+=1
        face(px, py)
        right(86)
        forward(1)
        if i == 5000:
            return

# main program
setPlaygroundSize(600, 600)
makeTurtle()

# set aimed point (px, py)
px = 50
py = 50
setPenColor("red")
setPos(px, py)
dot(10)
setPenColor("darkgreen")

# set intial turtle's
# position and heading

setPos(-230, -90)
label("Turtle started here")
xcor = -100
ycor = -100
a = 0
setPos(xcor,ycor)
heading(a)
hideTurtle()

# go!
keepBearing(px, py)
```

Listing 7.5: The turtle moves toward a light source by keeping a constant bearing.

7.3.3 Two-eye vision

In this example we try to model a two-eyes vision mechanism, simply working on the fields of view. (section 5.4.2).

```
1
2  # Turtle two-eye model
3  # towards(px, py) and bearing(px, py) tested in towards-test.py
4  # 26.2.2020
5
6  from gturtle import *
7  from random import randint
8  from math import *
9
10 # turns randians into degrees
11 def r2d(ar):
12      return ar / pi * 180
13
14 # gives direction to point
15 # (px, py) from turtle position
16 # expressed as angle in degrees
17 def towards(px, py):
18     pcor = getPos()
19     dx = px - pcor[0]
20     dy = py - pcor[1]
21
22     if dy > 0:
23         if dx == 0:
24             return 0
25         if dx > 0:
26             return r2d(atan(dx/dy))
27         else:
28             return 360 + r2d(atan(dx/dy))
29     if dy < 0:
30         return 180 + r2d(atan(dx/dy))
31
32 # gives bearing angle of direction
33 # to point (px, py) with respect to
34 # turtle's heading
35 def bearing(px, py):
36     h = heading() % 360
37     t = towards(px, py)
38     b = abs(t - h)
39     return b
40
41 # check if seen by left eye
42 def leftEye(b):
43     if b > 350:
44         return True
45     if b < 60:
46         return True
47     return False
```

Listing 7.6: The turtle moves toward a light source by keeping it within the eyes field of view.

```
54
55 # check if seen by right eye
56 def rightEye(b):
57     if b > 300:
58         return True
59     if b < 10:
60         return True
61     return False
62
63 # heading for light source
64 def headFor(px, py):
65     while True:
66         b = bearing(px, py)
67         if leftEye(b) or rightEye(b):
68             forward(10)
69          else:
70             r = randint(1,359)
71             left(r)
72
73
74 setPlaygroundSize(1000, 1000)
75 makeTurtle()
76 hideTurtle()
77
78 # Drawing reference system
79 setPos(0,-200)
80 moveTo(0,200)
81 setPos(-200,0)
82 moveTo(200,0)
83
84 # (px, py) light source
85 # (xcor, ycor) turtle initial location
86 # h turtle initial heading
87 px = 100
88 py = -100
89 setPos(px, py)
90 dot(10)
91 xcor = -100
92 ycor = -200
93 a = 315
94 setPos(xcor,ycor)
95 heading(a)
96
97 showTurtle()
98
99 headFor(px, py)
```

Listing 7.7: The turtle moves toward a light source by keeping it within the eyes field of view.

7.3.4 Two-eyes vision with intensity perception

Here we model a two-eyes vision mechanism, trying to keep the intensity detected by the eyes in balance. (section 5.4.3).

```
# Turtle two-eye model with intensity
# towards(px, py) and bearing(px, py) tested in towards-test.py
# At #83 a drop of randomness in this version...
# 27.2.2020

from gturtle import *
from random import randint
from math import *

# turns randians into degrees
def r2d(ar):
     return ar / pi * 180

# gives direction to point
# (px, py) from turtle position
# expressed as angle in degrees
def towards(px, py):
    pcor = getPos()
    dx = px - pcor[0]
    dy = py - pcor[1]

    if dy > 0:
        if dx == 0:
            return 0
        if dx > 0:
            return r2d(atan(dx/dy))
        else:
            return 360 + r2d(atan(dx/dy))
    if dy < 0:
        return 180 + r2d(atan(dx/dy))

# gives bearing angle of direction
# to point (px, py) with respect to
# turtle's heading
def bearing(px, py):
    h = heading() % 360
    t = towards(px, py)
    b = t - h
    if b < 0:
        b = 360 + b
    return b

# check if seen by left eye
def leftEye(b):
    if b > 300:
        return True
    if b < 10:
        return True
    return False
```

Listing 7.8: The turtle moves toward a light source by keeping left and right intensities in balance.

```
58
59 # check if seen by right eye
60 def rightEye(b):
61     if b > 350:
62         return True
63     if b < 60:
64         return True
65     return False
66
67 # gives distance from point (px, py)
68 def dist(px, py):
69     p = getPos()
70     return sqrt((px - p[0])**2 + (py - p[1])**2)
71
72 # intensity perceived from
73 # left eye
74 def intensityLeft(px, py):
75     strength = 1000000
76     b = bearing(px, py)
77     if not leftEye(b):
78         return 0
79     fact = strength / (dist(px, py)**2)
80     a = bearing(px, py) - 45
81     return fact * cos(a*pi/180)
82
83 # intensity perceived from
84 # right eye
85 def intensityRight(px, py):
86     strength = 1000000
87     b = bearing(px, py)
88     if not rightEye(b):
89         return 0
90     fact = strength / (dist(px, py)**2)
91     a = bearing(px, py) + 45
92     return fact * cos(a*pi/180)
93
94 # heading for light source
95
96 def headBySight(px, py):
97     i = 0
98     while True:
99         i = i + 1
100        left(randint(-90,90)) # some randomness here
101        forward(1)
102        iL = intensityLeft(px, py)
103        iR = intensityRight(px, py)
104        if iL > iR:
105            left(10)
106        elif iL < iR:
107            right(10)
108        else:
109            while intensityLeft(px, py) == 0:
110                left(randint(-180,180))
111
112 setPlaygroundSize(1000, 1000)
113 makeTurtle()
114 hideTurtle()
115
116 # Drawing reference system
117 setPos(0,-200)
118 moveTo(0,200)
119 setPos(-200,0)
120 moveTo(200,0)
```

Listing 7.9: The turtle moves toward a light source by keeping left and right intensities in balance.

```
121
122 # (px, py) light source
123 # (xcor, ycor) turtle initial location
124 # h turtle initial heading
125 px = 100
126 py = -200
127 setPos(px, py)
128 dot(10)
129 xcor = 100
130 ycor = 200
131 a = -90
132 setPos(xcor,ycor)
133 heading(a)
134
135 hideTurtle()
136
137 print("Start!")
138 headBySight(px, py)
```

Listing 7.10: The turtle moves toward a light source by keeping left and right intensities in balance.

7.3.5 Managing two turtles simultaneously in LibreLogo

LibreLogo is a monotasking environment so far, i.e. you can control only one turtle at a time. Morever this is meant to produce a graphic, not to model a dynamic behaviour. Here we are forcing these limitations by implementing the basic mechanism used by computers to achieve a multitasking effect. (section 5.5)

```
HOME
CLEARSCREEN
HIDETURTLE

TO SAVERED
  GLOBAL PRED, HRED, PGREEN, HGREEN
  PRED = POSITION
  HRED = HEADING
END

TO RESTRED
  GLOBAL PRED, HRED, PGREEN, HGREEN
  X = PRED[0]
  Y = PRED[1]
  PENUP
  POSITION [X,Y]
  PENDOWN
  HEADING HRED
END

TO SAVEGREEN
  GLOBAL PRED, HRED, PGREEN, HGREEN
  PGREEN = POSITION
  HGREEN = HEADING
END

TO RESTGREEN
  GLOBAL PRED, HRED, PGREEN, HGREEN
  X = PGREEN[0]
  Y = PGREEN[1]
  PENUP
  POSITION [X,Y]
  PENDOWN
  HEADING HGREEN
END
```

Listing 7.11: Managing two turtle simultaneously in LibreLogo

```
36
37 TO EXECBOTH
38   REPEAT 50 [
39     RESTGREEN
40     EXECGREEN
41     SAVEGREEN
42     RESTRED
43     EXECRED
44     SAVERED
45 ]
46 END
47
48 TO EXECRED
49
50   PENCOLOR ‘‘red’’
51   FORWARD RANDOM(10) + 1
52   RIGHT RANDOM(120) - 60
53 END
54
55 TO EXECGREEN
56   PENCOLOR ‘‘green’’
57   FORWARD RANDOM(10) + 1
58   RIGHT RANDOM(120) - 60
59 END
60
61
62 PENCOLOR ‘‘black’’
63 FILLCOLOR ‘‘RED’’
64
65 HEADING ANY
66 SAVEGREEN
67 HEADING ANY
68 SAVERED
69 CIRCLE 4
70
71 EXECBOTH
72
73 PRINT ‘‘Ok!’’
```

Listing 7.12: Managing two turtle simultaneously in LibreLogo

7.3.6 Approximating the circle perimeter

This is the LibreLogo code used to make ghe first figures in section 6.4 to show how a circle can be approximated by regular polygons.

```
1
2  R = 150                        ; radius of circle
3  C = 2*PI*R                   ; circumference
4
5  CLEARSCREEN
6  HIDETURTLE
7  PENSIZE 0.5
8
9  REPEAT 30 [
10    HOME
11    PENUP
12    LEFT 90 FORWARD 150 RIGHT 90 ; transpose the home 150 pt left
13    PENDOWN
14    N = REPCOUNT+2                       ; REPCOUNT iteration counter
15    A = 2*PI/N
16    A2 = A/2
17    L = 2*R*SIN(A2)                         ; polygon side
18    D = PI/2-(PI-A)/2
19    DD = D/PI*180                           ; turtle deviation angle
       in degrees
20    RIGHT DD                                  ; first half deviation
21    REPEAT N [                                ; drawing N sides
22       FORWARD L
23       RIGHT DD*2
24    ]
25 ]
26 HOME                                            ; now draw the ‘‘true
       ’’ circumference
27 PENUP
28 LEFT 90 FORWARD 150 RIGHT 90
29 PENDOWN
30 N = 360
31 A = 2*PI/N
32 A2 = A/2
33 L = 2*R*SIN(A2)
34 D = PI/2-(PI-A)/2
35 DD = D/PI*180
36 RIGHT DD
37 REPEAT N [
38    FORWARD L
39    RIGHT DD*2
40 ]
```

Listing 7.13: Circle radius and number of polygons sides can be adjusted as desired

7.3.7 Fractal L-system stick tree

Script for drawing a fractal stick tree with the L-system formal language (section 6.2.1).

```
# Stick tree "plant"
# with L-system
# 24.5.2022

# Axiom: B
# B -> F[-B]+B
# F -> FF


from gturtle import *

makeTurtle()
clearScreen()
hideTurtle()

def tpush(lp,lh):
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):
        setPos(lp.pop())
        heading(lh.pop())

def B(l, delta, iter, lp, lh):
    if iter == 1:
        pass
    else:
        forward(l)                    # F forward
        tpush(lp,lh)                  # [ beginning right branch
        right(delta)                  # - right rotation
        B(l/2, delta, iter-1, lp, lh) # B recursive branch call
        tpop(lp, lh)                  # ] closing right branch
        left(delta)                   # + left rotation
        B(l/2, delta, iter-1, lp, lh) # B recursive branch call


l = 100
delta = 50
iter = 7

lp = []
lh = []

B(l, delta, iter, lp, lh)
```

Listing 7.14: Stick tree "plant" with L-system.

7.3.8 The impossible task of measuring the length of coastlines

This is the R code we used to download geographical coastline data from the Database of Global Administrative Areas (GADM) and to calculate perimeters of Britain and other imaginary geometrical isles (section 6.4). R is a free software language for statistical calculation and processing of big data. The knowledge of this language is not among the objectives of this book. We show the code here just for reference.

```
# Original code: Spatial Data Science: The length of a coastline
# https://rspatial.org/raster/cases/2-coastline.html
# $\copyright$ Copyright 2016-2020, Robert J. Hijmans
# CC BY-SA 4.0

# Remixed by
# $\copyright$ Copyright 2020, Andreas Robert Formiconi
# CC BY-SA 4.0

# high spatial resolution (30 m) coastline for the
# United Kingdom from the Database of Global Administrative
# Areas (GADM).

library(raster)
uk <- raster::getData('GADM', country='GBR', level=0)
par(mai=c(0,0,0,0))
plot(uk)

# This is a single "multi-polygon" (it has a single feature) and
# a longitude/latitude coordinate reference system.
data.frame(uk)

# Let's transform this to a planar coordinate system.
# That is not required, but it will speed up computations.
# We used the "British National Grid coordinate reference system,
# which is based on the Transverse Mercator (tmerc) projection.

# latitude to the British National Grid.
prj <- "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000
     +y_0=-100000 +ellps=airy +datum=OSGB36 +units=m"

# Note that the units are meters.
# With that we can transform the coordinates of uk from longitude

library(rgdal)
guk <- spTransform(uk, CRS(prj))

# We only want the main island, so want need to separate (
    disaggregate)
# the different polygons.

duk <- disaggregate(guk)
head(duk)

# Now we have 920 features. We want the largest one.

a <- area(duk)
i <- which.max(a)
a[i] / 1000000
b <- duk[i,]
```

Listing 7.15: Remixing of "Spatial Data Science: The length of a coastline" code. See heading of listing. Listing continues on new page.

```
49
50 # Britain has an area of about 220,000 km**2.
51
52 par(mai=rep(0,4))
53 plot(b)
54 # Now the function to go around the coast with a ruler (yardstick)
55 # of a certain length
56
57 measure_with_ruler <- function(pols, length, lonlat=FALSE) {
58     # some sanity checking
59     stopifnot(inherits(pols, 'SpatialPolygons'))
60     stopifnot(length(pols) == 1)
61     # get the coordinates of the polygon
62     g <- geom(pols)[, c('x', 'y')]
63     nr <- nrow(g)
64     # we start at the first point
65     pts <- 1
66     newpt <- 1
67     while(TRUE) {
68         # start here
69         p <- newpt
70         # order the points
71         j <- p:(p+nr-1)
72         j[j > nr] <- j[j > nr] - nr
73         gg <- g[j,]
74         # compute distances
75         pd <- pointDistance(gg[1,], gg, lonlat)
76         # get the first point that is past the end of the ruler
77         # this is precise enough for our high resolution coastline
78         i <- which(pd > length)[1]
79         if (is.na(i)) {
80             stop('Ruler is longer than the maximum distance found')
81         }
82         # get the record number for new point in the original order
83         newpt <- i + p
84         # stop if past the last point
85         if (newpt >= nr) break
86         pts <- c(pts, newpt)
87     }
88     # add the last (incomplete) stick.
89     pts <- c(pts, 1)
90     # return the locations
91     g[pts, ]
92 }
93
94 # Let's call the function with rulers of different lengths.
95
96 y <- list()
97 rulers <- c(10,25,50,100,200,400) # km
98 # rulers <- c(50, 100, 250) # km
99 for (i in 1:length(rulers)) {
100     y[[i]] <- measure_with_ruler(b, rulers[i]*1000)
101 }
```

Listing 7.16: Remixing of "Spatial Data Science: The length of a coastline" code. See heading of listing. Page 2 of listing

```
102
103 # Object y is a list of matrices containing the locations
104 # where the ruler touched the coast. We can plot these on top
105 # of a map of Britain.
106 # plot-maps.png
107
108 png(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/
        POWERFUL-IDEAS/Turtle-book/coastlines/plot-maps.png")
109 par(mfrow=c(2,3), mai=rep(0,4))
110 # for (i in 1:length(y)) {
111 for (i in length(y):1) {
112     # plot(b, col='lightgray', lwd=2)
113     plot(b, col='lightgray', lwd=1)
114     p <- y[[i]]
115     # lines(p, col='red', lwd=3)
116     lines(p, col='black', lwd=1)
117     # points(p, pch=20, col='blue', cex=2)
118     bar <- rbind(cbind(525000, 900000), cbind(525000, 900000-rulers
        [i]*1000))
119     lines(bar, lwd=2)
120     points(bar, pch=20, cex=1.5)
121     text(525000, mean(bar[,2]), paste(rulers[i], '  km'), cex=1.5)
122     # text(525000, bar[2,2]-50000, paste0('(', nrow(p), ')'), cex
        =1.25)
123     text(525000, bar[2,2]-50000, paste0('(', nrow(p)*rulers[i], '
        km)'), cex=1.25)
124 }
125 dev.off()
126
127 # Here is the fractal (log-log) plot. Note how the axes are on
128 # the log scale, but that we used the non-transformed values for
129 # the labels.
130
131 # number of times a ruler was used
132 n <- sapply(y, nrow)
133
134 # Plot of perimeter lengths vs ruler lengths
135 # Britain and circular isle data
136 # plot-lin.png
137
138 png(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/
        POWERFUL-IDEAS/Turtle-book/coastlines/plot-lin.png")
139 # set up empty plot
140 plot(rulers, n, type='n', xlim=c(5,500), ylim=c(2000,6000), axes=
        FALSE,
141     xaxs="i",yaxs="i", xlab='Ruler length (km)', ylab='Approximated
         perimeter (km)')
142 xtics <- c(0,100,200,300,400,500,600)
143 ytics <- c(1000,2000,3000,4000,5000,6000)
144 axis(1, at=xtics, labels=xtics)
145 axis(2, at=ytics, labels=ytics, las=2)
146 # linear regression line
147 m <- lm(log(n)~log(rulers))
148 abline(m, lwd=1, col='red')
149 # add observations
150 # points(log(rulers), log(n), pch=20, cex=2, col='red')
151 points(rulers, n*rulers, pch=20, cex=2, col='red')
152 poly_n <- c(300,120,60,30,20,15,12,10,8,6,4,3)
153 c_rulers <- 3000/pi*sin(pi/poly_n)
154 points(c_rulers, poly_n*c_rulers, pch=20, cex=2, col='green')
155 abline(h=3000, lwd=1, col="green")
156 dev.off()
```

Listing 7.17: Remixing of "Spatial Data Science: The length of a coastline" code. See heading of listing. Page 3 of listing

```
157
158 # Log-log plot of perimeter lengths vs ruler lengths
159 # Britain and circular isle
160 # plot-log.png
161
162 png(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/
    POWERFUL-IDEAS/Turtle-book/coastlines/plot-log.png")
163 # set up empty plot
164 plot(log(rulers), log(n*rulers), type='n', xlim=c(1,7), ylim=c
    (7.5,9), axes=FALSE,
165      xaxs="i",yaxs="i", xlab='Ruler length (km)', ylab='
    Approximated perimeter (km)')
166 xtics <- c(5,10,25,50,100,200,300,400,500,600,700,800,900,1000)
167 ytics <- c(2000,2800,3600,4400,5200,6000)
168 axis(1, at=log(xtics), labels=xtics)
169 axis(2, at=log(ytics), labels=ytics, las=2)
170 # linear regression line
171 m <- lm(log(n*rulers)~log(rulers))
172 abline(m, lwd=1, col='red')
173 # add observations
174 points(log(rulers), log(n*rulers), pch=20, cex=2, col='red')
175 poly_n <- c(300,120,60,30,20,15,12,10,8,6,4,3)
176 c_rulers <- 3000/pi*sin(pi/poly_n)
177 points(log(c_rulers), log(poly_n*c_rulers), pch=20, cex=2, col='
    green')
178 abline(h=log(3000), lwd=1, col="green")
179 dev.off()
180
181 # Log-log plot of perimeter lengths vs ruler lengths
182 # Britain, circular isle and Koch data
183 # plot-log-koch.png
184
185 png(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/
    POWERFUL-IDEAS/Turtle-book/coastlines/plot-log-koch.png")
186 # set up empty plot
187 plot(log(rulers), log(n*rulers), type='n', xlim=c(1,7), ylim=c
    (7.5,9), axes=FALSE,
188      xaxs="i",yaxs="i", xlab='Ruler length (km)', ylab='
    Approximated perimeter (km)')
189 xtics <- c(5,10,25,50,100,200,300,400,500,600,700,800,900,1000)
190 ytics <- c(2000,2800,3600,4400,5200,6000)
191 axis(1, at=log(xtics), labels=xtics)
192 axis(2, at=log(ytics), labels=ytics, las=2)
193 # linear regression line
194 m_b <- lm(log(n*rulers)~log(rulers))
195 abline(m_b, lwd=1, col='red')
196 # add observations
197 points(log(rulers), log(n*rulers), pch=20, cex=2, col='red')
198 poly_n <- c(300,120,60,30,20,15,12,10,8,6,4,3)
199 c_rulers <- 3000/pi*sin(pi/poly_n)
200 points(log(c_rulers), log(poly_n*c_rulers), pch=20, cex=2, col='
    green')
201 abline(h=log(3000), lwd=1, col="green")
202 k_step <- c(1,2,3,4,5)
203 k_n <- 4**k_step
204 k_rulers <- 1500*((1/3)**k_step)
205 # linear regression line
206 m_k <- lm(log(k_n*k_rulers)~log(k_rulers))
207 abline(m_k, lwd=1, col='blue')
208 points(log(k_rulers), log(k_n*k_rulers), pch=20, cex=2, col='blue')
209 dev.off()
```

Listing 7.18: Remixing of "Spatial Data Science: The length of a coastline" code. See heading of listing. Page 4 of listing

Bibliography

[Abelson and diSessa, 1980] Abelson, H. and diSessa, A. (1980). *Turtle Geometry - The computer as a medium for exploring mathematics*. MIT Press, Cambridge, Massachusetts.

[Alighieri, 1321] Alighieri, D. (1321). *Divina Commedia - Paradiso*. La Scuola Editrice, Brescia, 1969 edition.

[Bau et al., 2017] Bau, D., Gray, J., Kelleher, C., Sheldon, J., and Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communication of the ACM*, 60:72–80.

[Bruner, 1960] Bruner, J. S. (1960). *The process of education*. Harvard University Press, Cambridge.

[Byrne, 1847] Byrne, O. (1847). *The first six Books of the Elements of Euclid*. William Pickering, London. Available at: `https://archive.org/details/firstsixbooksofe00eucl/page/n7/mode/2up.Accessed:2020-12-28`.

[Dehaene, 2020] Dehaene, S. (2020). *How we learn*. Penguin, London.

[Eisenbeis, 2006] Eisenbeis, G. (2006). Artificial night lighting and insects: Attraction of insects to streetlamps in a rural setting in germany. In Rich, J. and Longcore, T., editors, *Ecological Consequences of Artificial Night Lighting*, chapter 12, pages 281–304. Island Press, Washington D.C.

[Esseily et al., 2016] Esseily, R., Rat-Fischer, L., Somogyi, E., O'Regann, K., and Fagard, J. (2016). Humour production may enhance observational learning of a new tool-use action in 18-month-old infants. *Cognition and Emotion*, 30:817–825.

[Fisher et al., 2014] Fisher, A. V., Godwin, K. E., and Seltman, H. (2014). Visual environment, attention allocation, and learning in young children: When too much of a good thing may be bad. *Phsycological Science*, 7:1362–1370.

[Fraser, 2015] Fraser, N. (2015). Ten things we've learned from blockly. Available at: `https://doi.org/10.1109/BLOCKS.2015.7369000`. 2015 IEEE blocks and beyond Workshop (blocks and beyond). Presented at the 2015 IEEE blocks and beyond Workshop (blocks and beyond).

[Garcia et al., 2015] Garcia, D., Harvey, B., and Barnes, T. (2015). The beauty and joy of computing. *ACM Inroads*, 6:71–79.

[Goldenberg, 1982] Goldenberg, E. P. (1982). Logo — a cultural glossary. *BYTE*, 7:210–228.

[Harvey, 1982] Harvey, B. (1982). Why logo? *BYTE*, 7:163–193.

[Hromkovič, 2009] Hromkovič, J. (2009). *Algorithmic Adventures - From knowledge to magic*. Springer, Heidelberg.

[Hromkovič and Kohn, 2018a] Hromkovič, J. and Kohn, T. (2018a). *Einfach Informatik - Programmieren - Sekundarstufe I*. Klett und Balmer, Baar.

[Hromkovič and Kohn, 2018b] Hromkovič, J. and Kohn, T. (2018b). *Einfach Informatik - Programmieren Begleitband - Sekundarstufe I*. Klett und Balmer, Baar.

[Hromkovič J., 2017] Hromkovič J., L. R. (2017). The computer science way of thinking in human history and consequences for the design of computer science curricula. In Dagienė V., H. A., editor, *Informatics in Schools: Focus on Learning Programming. ISSEP 2017. Lecture Notes in Computer Science, vol 10696.*, pages 3–11. Springer, Cham.

[Lewis, 2015] Lewis, C. (2015). How programming environment shapes perception, learning and goals: Logo vs. scratch. `http://ims.mii.lt/ims/konferenciju_medziaga/SIGCSE'10/docs/p346.pdf`. Accessed: 2017-08-13.

[LogoTree, 2020] LogoTree (2002-2020). Logo tree. Available at: `https://pavel.it.fmi.uni-sofia.bg/logotree/index.html`. Accessed: 2020-12-30.

[Lorenz, 1972] Lorenz, N. E. (1972). Predictability: Does the flap of a butterfly's wings in brazil set off a tornado in texas? `https://static.gymportalen.dk/sites/lru.dk/files/lru/132_kap6_lorenz_artikel_the_butterfly_effect.pdf`. Accessed: 2021-01-24.

[Mandelbrot, 1967] Mandelbrot, B. (1967). How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, 156:636–638.

[Morin, 1977] Morin, E. (1977). *Il metodo 1. La natura della natura*. Cortina Editore, Milano.

[Morin, 1999] Morin, E. (1999). *I sette saperi necessari all'educazione del futuro*. Cortina Editore, Milano.

[Papert, 1986] Papert, S. (1986). New mindstorms 1 - resonances [video]. Available at: `https://el.media.mit.edu/logo-foundation/resources/onlogo/video/newmindstorms1.mp4#t=6m20s`. Accessed: 2020-10-08.

[Papert, 1993] Papert, S. (1993). *Mindstorms, Children, Computers, and Powerful Ideas*. Basic books, New York, 2 edition.

[Peitgen et al., 1992] Peitgen, H. O., Hartmut, J., and Saupe, D. (1992). *Chaos and Fractals*. Springer, Berlin.

[Persico, 1956] Persico, E. (1956). Che cos'è che non va? *Giornale di Fisica*, 1:64–67.

[Press et al., 2007] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes – The Art of Scientific Computing*. Cambridge University Press, Cambridge, Masssachusetts.

[Prusinkiewicz et al., 1988] Prusinkiewicz, P., Lindenmayer, A., and Hanan, j. (1988). Developmental models of herbaceous plants for computer imagery purposes. *Computer Graphics*, 22:141–150.

[Resnick et al., 2009] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eanstmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, J. (2009). Scratch: programming for all. *Communication of the ACM*, 52:60–67.

[Richardson, 1961] Richardson, L. F. (1961). The problem of contiguity: An appendix to statistic of deadly quarrels. *General Systems Yearbook*, 6:139.

[Sherin, 2001] Sherin, B. L. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *Int. Journal of Computers for Mathematical Learning*, 6:1–61.

[Solomon, 1982] Solomon, C. (1982). Why logo? *BYTE*, 7:195–208.

[Walter, 1950] Walter, W. (1950). An imitation of life. *Scientific American*, 182:42–45.

[Weintrop and Wilensky, 2015] Weintrop, D. and Wilensky, U. (2015). Using commutative assessments to compare conceptual understanding in blocks based and text based programs. `http://dweintrop.github.io/papers/Weintrop_Wilensky_ICER_2015.pdf`. Accessed: 2017-08-13.

[Weintrop and Wilensky, 2019] Weintrop, D. and Wilensky, U. (2019). Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers and Education*, 142:1–17.

[Xu and Vashti, 2008] Xu, F. and Vashti, G. (2008). Intuitive statistics by 8-month-old infants. *PNAS*, 105:5012–5015.