Building knowledge with turtle geometry

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

Andreas R. Formiconi

Version 2.0

January 2024

This work is released under the Creative Commons Attribuzione 2.5 Italia license. In order to read a copy of the license visit http://creativecommons.org/licenses/by/2.5/it/ or write to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
Preface

Overall, this book is the result of two decades of experience in teaching the use of technology in schools. Specifically, parts of this book were originally written for a "Swiss Day for Computer Science Education" workshop held at ETH Zurich in 2020. It was later extended for the benefit of computer science students in the Data Science curriculum of the Department of Statistics, Computer Science and Applications at the University of Florence. It is multidisciplinary in that it transcends the rigid disciplinary divisions that dominate the world of education. Software writing is used as a tool to explore different scientific fields. Many of the examples proposed can be thought of as real virtual laboratories for thinking about particular geometric and mathematical facts, physical or biological phenomena, ranging from simple geometric explorations for children to quite complex simulations in the fields of physics and biology. The text is rich in examples that can be reproduced and developed by the reader. In addition, it is enriched by several specific boxes offering pedagogical reflections and insights on certain "strong ideas" evoked by the examples - mathematical, physical, or computational ideas. Basically, the coded examples available in the book are written in Logo, a well-known programming language created for educational purposes fifty years ago by scientists at MIT's Artificial Intelligence Laboratory, and it is also the same language used to construct Abelson and diSessa’s Turtle Geometry in order to construct geometric shapes based on the metaphor of a turtle drawing lines by moving in the plane or space. It is a conceptual tool used in various scientific fields, as well as in education. The specific implementations of LOGO used in the book belong to the free software categories, so the reader does not have to incur additional costs to perform the explorations. The most common implementation of LOGO is based on Python. The book is accompanied by a MOOC in which iconographic material, videos of dynamic examples, downloadable program listings, questions with feedback, and links to other resources enrich the reader’s experience. Special links have been included in the book that lead directly to the corresponding material in the MOOC. In this sense, the intention is to create a kind of remote dialog with the reader, who can participate by reproducing and developing the examples.
Firenze,

January 2024

Andreas Robert Formiconi
Acknowledgements

I would like to thank Professor Maria Cecilia Verri for reflections on the problem of teaching computer science in primary and secondary schools. There are many primary and secondary school teachers whom I have to thank for discussions on the use of technology in the classroom. These experiences have been crucial in giving a concrete slant to the considerations developed in this book. In particular, I would like to thank the teachers of the Pestalozzi City School in Florence and the Strada in Chianti Primary School, who allowed me to carry out workshops in their classrooms with my students. I must also thank the teachers for the discussions during the workshops organized in their schools in Paderno d’Adda, Marsciano, Pontassieve and Castel Del Piano. Many thanks also to all the students who gave me feedback during their internships, where they applied what was suggested to them.
Contents

1 Learning, emotions, complexity: not just what, but how 1
  1.1 From great teachers of the past to last neuroscience findings 3
  1.2 Fighting the “School of Mourning” . . . . . . . . . . . . . 19

Bibliography 25

2 Software languages 29
  2.1 What we mean by Logo . . . . . . . . . . . . . . . . . . . . . 30
  2.2 The Logo culture . . . . . . . . . . . . . . . . . . . . . . . . . . 33
  2.3 Turtle geometry . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
  2.4 Free software tools . . . . . . . . . . . . . . . . . . . . . . . . . 45

Bibliography 55

3 Basic turtle programming 59
  3.1 Making things smaller and larger . . . . . . . . . . . . . . . 81
  3.2 Doing round things . . . . . . . . . . . . . . . . . . . . . . . . 86

Bibliography 117

4 Simulation: physics dynamic 119
  4.1 Free-falling body . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

Bibliography
CONTENTS

4.2 Free-falling body with constant acceleration and horizontal velocity component ........................................... 126
4.3 Free-falling body with air resistance ........................................ 127
4.4 Body hanging from spring ........................................ 130
4.5 Pendulum ..................................................... 132
4.6 Body free-falling from space ........................................ 134
4.7 Calculate the orbit of Halley’s Comet... ........................................ 136

Bibliography ........................................ 149

5 Simulation: behaviour ........................................ 151
  5.1 Is there a place for randomness in computers? The Turtle plays the turtle... ........................................ 151
  5.2 Shaping the environment ........................................ 156
  5.3 Modeling smell ........................................ 160
  5.4 Modeling sight ........................................ 168
  5.5 Modeling interactions ........................................ 181

Bibliography ........................................ 193

6 Simulation: fractals growth ........................................ 196
  6.1 Fractals ..................................................... 199
  6.2 Fractals and randomness ........................................ 222
  6.3 Mathematical and natural fractals: the impossible task of measuring the length of coastlines ........................................ 224

Bibliography ........................................ 249

A Appendices ........................................ 251
  A.1 Index of powerful ideas and relevant reflections ........................................ 252
  A.2 Program listings ........................................ 254
Chapter 1

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

It is possible to teach a skillful pupil abilities above his actual level, like one can train young children in the arithmetic of fractions without telling them what fractions mean, or older children in differentiating and integrating though they do no know what differential quotients and integrals are.

— Hans Freudenthal,
Mathematics as an educational task

This book is about the use of turtle geometry in primary and secondary schools. The text is largely inspired by the work and vision of Seymour Papert, a mathematician and computer scientist who also received his doctorate from Jean Piaget. Today, Papert is best known as the inventor of the Logo educational programming language. The Logo programming lan-
CHAPTER 1. LEARNING, EMOTIONS, COMPLEXITY: NOT JUST WHAT, BUT HOW

language was designed to promote a better way of teaching science and to encourage self-reflection: learning while reflecting on one’s own learning. We live in a time of very fast change, too fast. Papert’s deep thinking too often goes unnoticed today. Many current “coding” practices look like a kind of educational game, but the connection between such practices and sound scientific concepts seems to be largely absent. This text is an effort to fill that gap. In particular, by presenting a series of exercises ranging from the primary level to the last years of secondary school, we attempt to recover and discuss Papert’s concepts of syntonic learning and powerful ideas [1]. 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 book the reader will find gray text boxes intended to comment on powerful ideas that are evoked by the exercises. In appendix A.1 (pag. 252) 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, or to the simulation of growth dynamics, ending with a simulation of general relativity! The textbook they wrote, Turtle Geometry [2], is very rich in ideas for an educator with a reasonable mathematical background, but too difficult to be used at large in school contexts — 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.

Surprisingly, after almost ten years of teaching Logo, I realized that there is not much difference between children and adults. On average, children do even better the first time around! We often discuss this point with students who claim to have had great difficulty “talking to the turtle”.1. Most of them

---

1The turtle will be the protagonist of our activities. We’ll explain it below.
experience a kind of “resurrection” after an initial struggle: “At first I didn’t understand anything, but then... how cool!” However, a few of them never get rid of the bad mood, at least within the two and a half weeks of the lab. Talking to the more reluctant students is interesting because they are amazed when I tell them that most of the time, nine-year-old kids get along quite well with the turtle. When you try to dig into this paradox, it seems that children naturally grasp the playful, albeit thoughtful, aspect of the situation. On the other hand, adults easily get caught up in their own prejudices: I don’t have a head for numbers, computers and I are very far apart, I’ve never been into technology... The last one in particular is a statement I have heard so often, surprisingly even from older digital natives: people with thousands of Facebook contacts openly claim not to be able to cope with technology! Having emphasized this, please approach the turtle with childlike curiosity, starting from scratch. Since you probably know how kids approach new things, try to do the same, but remember that playful doesn’t mean easy. Learning is very rewarding, but it takes hard work.

1.1 From great teachers of the past to last neuroscience findings

In cognitive science, learning consists of building an internal model of the external 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 it as a sort of magic mirror of reality, but that would be a misleading metaphor. The brain constantly monitors all external and internal sensory stimuli and compares the content received with the internal representation of the world. This is an endless learning process, but it is not just a linear process of accumulating new information. Making sense of the world is a matter of recovering the hidden causes of events in order to reconstruct a general theory that can explain them. It seems that our brains are extremely good at this, and at a very early age. Scientific studies have been done that show how 8-month-old babies
CHAPTER 1. LEARNING, EMOTIONS, COMPLEXITY: NOT JUST WHAT, BUT HOW

behave like budding scientists who think like statisticians [3].

This scientific attitude should be better harnessed in schools. Several authors have had the intuition that exploration should be an important part of school activities — John Dewey, Maria Montessori, Célestin Freinet, Seymour Papert, Emma Castelnuovo, to name a few. There is a strong sense that we need to make much better use of the natural budding scientist attitude of the young brain. The point is particularly striking when it comes to science subjects: the fact that STEM disciplines are taught, in most cases, with very little attention paid to the basic processes of scientific knowledge creation is an amazing oxymoron. Teaching science should be a great opportunity to put such a natural brain talent to work. Unfortunately, this is not the case. STEM, like everything else, is usually taught using a passive knowledge transfer model.

It was precisely for this kind of problem that Papert designed Logo as an environment where students could explore, aim, and ultimately discover. Papert is often associated with “pure discovery learning”, that is, learning without the mediation of a teacher. But this 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. [4, 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 their own devices. Simply exposing students to an educational artifact or context does not create magic.

The four pillars of learning

Papert’s ideas, on the other hand, are particularly interesting in light of current neuroscience. For example, it is easy to find in his words the first three of what Stanislas Dehaene calls the four pillars of learning: attention, active engagement, error feedback, and consolidation.

Regarding the first pillar — attention — it is important to emphasize that only through a strong listening attitude can a teacher hope to cap-
1.1. FROM GREAT TEACHERS OF THE PAST TO LAST NEUROSCIENCE FINDINGS

ture the attention of students. Something that Dehaene himself says very effectively [5, 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 experience that makes it possible to answer these questions. However, when teaching a new course, experience may be lacking. It is also important to identify individual problems, which can be caused by weaknesses, but also by specific talents. In order to get feedback from students, it is necessary to activate their attention and stimulate their active engagement. This is the only way to achieve listening. The right mood can be created by immediately offering small challenges, preferably something fun. The idea is to encourage exploration, but, as mentioned earlier, this does not mean that teachers should be absent and leave students to the mere presentation of some educational artifact or context. On the contrary, teachers need to be more involved, encouraging initiative and personal exploration, but without neglecting to guide students towards meaningful goals:

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.” [1, pag. 83]

And coming to the third pillar of learning, Dehaene’s idea that “error feedback is not synonymous with punishment” [5, 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. [1, pag. 52]

Finally, when it comes to school programming activities:
...bugs, are not seen as mistakes to be avoided like the plague, but as an intrinsic part of the learning process. [1, pag. 71]

The guiding idea, therefore, is to provide examples that can be reused according to Bruner’s concept of the spiral curriculum, which is intended as a form of long-term consolidation, enriched by the deepening of knowledge acquired in previous years. Of course, Dehaene’s consolidation also includes 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.

Dehaene’s four pillars are all essential for learning, but it all starts with the first one, attention: getting students’ attention is the main challenge a teacher has to face, failure in this task means defeating all subsequent efforts. And it is not at all trivial to get and, above all, to keep the attention of the students.

Usually it is taken for granted that students of all ages pay attention in class 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 that is not easy to activate and maintain for a long time. In this case, listening means being aware of the actual state of attention of one’s students and being able to assess how long attention can really be sustained. Everyone knows how powerful video games are at stimulating attention. Video games work because players are highly motivated. The art of teaching is to create contexts that stimulate that motivation as much as possible.

Attention has a lot to do with filtering: our brains see what they expect to see. We all tend to think that what we see is what everyone else can see. The famous “invisible gorilla” experiment [7] and other similar cases, such as simulating large unseen objects in front of a landing airplane, remind us that inattention to the unexpected can be incredibly powerful. When students are distracted, the teacher’s words are completely lost [5, 8]. Attention is also related to the myth of multitasking. The idea that we can do two (or more) tasks at the same time is illusory, unless one of the two tasks has been learned so well that it can be performed in a highly automated fashion, since automatization frees up the conscious workspace. If both tasks require
1.1. FROM GREAT TEACHERS OF THE PAST TO LAST
NEUROSCIENCE FINDINGS

conscious attention, one of them would necessarily slow down. Tasks other than the most important one will act as *distractions*. Distractions can also be caused by the visual environment. For example, overly decorated classrooms have been shown to distract students and reduce their concentration [5, 10, 15, 24].

It is also because of these attention considerations that we have preferred text-based rather than block-based programming environments for the scientific explorations described in this book: the blank page as a free space to maximize concentration and unleash creativity. Too many distractions, on the other hand, dilute concentration and dissipate creativity.

**Emotions**

Today, the role of emotions in learning has been widely documented by neuroscience:

Negative emotions crush our brain’s learning potential, whereas providing the brain with a fear-free environment may reopen the gates of neuronal plasticity [5].

Similarly, good experiences have a positive impact on learning, although with a lower priority than negative memories. Many studies have shown that making children laugh improves certain cognitive skills such as attention, motivation, perception, and/or memory, which in turn improves learning [6].

The crucial role of emotions in our rational thinking has been studied in depth for more than a couple of decades. Damasio, LeDoux, and Dehanene are among the most prominent neuroscientists who have studied the influence of emotions on human thinking from different perspectives. There are evolutionary reasons for the role of emotions in learning. The survival chances of an individual, of any species, are strongly influenced by the ability to remember. But not all memories are the same. The close relationship with the chances of survival determines a scale of priorities that 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 responded to as quickly as possible. In the face of such dangers, all
other brain activity takes a back seat: no matter how promising a courtship
or the smell of delicious food, the appearance of a predator will disrupt any
plan. Similar priorities determine the persistence of memories associated
with different situations, since circumstances involving serious emergency
threats require the most persistent, long-lasting memories.

A long evolutionary process has shaped the brain in its basic functions.
The persistence of memories is closely related to emotions. Memories ac-
quired under conditions of stress or fear always evoke the same negative
sensations each time they are recalled to consciousness. Such associations
are extremely powerful and difficult to erase. The amygdala and the hip-
pocampus, located deep in our “reptilian brain” are the brain structures
involved in the formation of emotional memories and emotional responses.
They work synergistically to form long-term memories of significant emo-
tional events, hard wiring the most negative experiences [11]. But even if a
math problem is not as dangerous as a saber-toothed tiger, the mechanism
activated in both cases is the same, once the first encounters with formal
mathematics have been stored along with traumatic emotions.

Mathematics, the scourge of students

We did not mention mathematics by chance. The subject of mathematics
education has been widely debated around the world for nearly a century.
Yet the issue is still unresolved. [12, 13]

In fact, in the field of neuroscience, the problem of “mathematics anxi-
ety” is a real syndrome, as Dehaene notes: [5, pag. 212]

The effects of school-induced stress have been particularly stud-
ied in the field of mathematics, the school subject most famous
for the all-too-well-known anxiety it induces in so many stu-
dents. 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 anxi-
ety 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.

Papert long ago (1980) described exactly how this suffering turns into an identity problem:

Mathophobia can, culturally and materially, limit people’s lives. Many more people have not completely given up on learning but are still severely hampered by entrenched negative beliefs about their capacities. Deficiency becomes identity: “I can’t learn French, I don’t have an ear for languages;” “I could never be a businessman, I don’t have a head for figures;” “I can’t get the hang of parallel skiing, I never was coordinated.” These beliefs are often repeated ritualistically, like superstitions. And, like superstitions, they create a world of taboos; in this case, taboos on learning. […] In a learning environment with the proper emotional and intellectual support, the “uncoordinated” can learn circus arts like juggling and those with “no head for figures” learn not only that they can do mathematics but that they can enjoy it as well. [1, p. 42]

And Maria Montessori pointed out even earlier (1934) how poor learning can result from too much early abstract teaching:

But the teacher’s concern is to get the child’s mind into abstraction as soon as possible, otherwise the very essence of teaching, which is to elevate the mind into the realms of abstraction, would be lost. So the path is all based on the teacher’s judgment. He judges what is easy and what is difficult, what is to be given and how, and finally, passing from the fleetingly concrete teaching to abstract combinations of numbers
and signs, he thinks he has penetrated and guided the child’s mind. But how often the teacher deceives himself! Only in exceptional cases did he penetrate into the child’s mind: most of the time the master’s work remained outside, because it did not interest the child. And the alleged abstraction was often the forced response of a mere mnemonic faculty subjected to torture.² [15, p. 4]

Several scholars have addressed the problem of mathematics education by focusing on geometry. Among them, Fishbein has been exploring the nature of geometric concepts since the 1960s. This author emphasized the peculiarity of geometric concepts as a combination of mental representations of spatial properties and purely abstract and logically determined concepts. These two categories were already present and widely studied in psychology, as was their interaction. Fischbein’s novelty is that he combined mental spatial representations and pure concepts into an inseparable unit, which he called “figural concept” [16]. Figural concepts are mental entities that are neither images nor genuine concepts. They are actually figures, but their properties are completely fixed within the framework of an axiomatic system. Many mistakes made by students are caused by a dissociation between the conceptual and figural components of figural concepts. Fischbein notes how the development of figural concepts is not a natural process, which accounts for the difficulty of learning geometry. The formation of figural concepts does not occur spontaneously, but must follow a careful progression. This progression was illustrated experimentally by Fischbein in a 1963 study of children aged 7 to 11 (grades 2 to 6). [17]³. In summary, the study shows that it is not until around the age of 11 that geometric interpretations begin to take precedence over perceptual ones, forming real, though not yet fully defined, figural concepts.

Instead, a more operational approach was taken by two Dutch teachers and researchers, Dina van Hiele-Geldof and her husband, Pierre Marie van Hiele. The couple created a model of geometric thinking that could be used

---

²Translated from Italian by myself.
³The study was published in Romanian, but the results of the experiment are also described in detail in the 1993 paper [16] cited a few lines above.
as a guide for teaching and assessing students [18, 19]. The model was the result of their Ph.D. thesis, completed in 1984 at the University of Utrecht. Since Dina died prematurely shortly after completing his doctorate, Pierre devoted the rest of his life to refining and disseminating the theory. The model consists of five levels that describe characteristics of the thinking process [20] – visualization, analysis, informal deduction, formal deduction and rigor – and five properties – sequential, advancement, intrinsic and extrinsic, linguistics and mismatch. Levels and property specifications are described in the corresponding text boxes.

Van Hiele’s levels of geometric thinking

0. **Visualization** At this level pupils are aware of space only as something that exists around them. They use visual perception and nonverbal thinking. For instance they recognize a rectangle as something that “looks like a door”. They don’t yet recognize parts and properties of geometric figures. At this level kids can learn geometric vocabulary, can identify given shapes and can reproduce a figure. However they make decisions based on perception, not reasoning.

1. **Analysis** Students begin to discern the characteristics of figures through observation and experiment. A figure is no longer recognized because “it looks like something” but because it bears certain properties. Language is important at this level. Relationships between properties or between figures are not seen. For instance, a triangle with equal sides is not necessarily one with equal angles, or a square is not a special kind of rectangle. They do not discern which properties are necessary and which are sufficient to determine a figure.

2. **Informal deduction** Students recognize relationships between properties and between figures. For instance equal sides in a triangle imply equal angles; a square is also a rectangle because it has all the properties of a rectangle. Definitions are understood,
deductive reasoning can take place, but the intrinsic meaning of deduction, the role of axioms, definitions and theorems is still not understood. Pierre van Hiele claimed that in his experience as a teacher of geometry “all too often, students have not yet reached this level of informal deduction. Consequently, they are not successful in their study of the kind of geometry that Euclid created, which involves formal deduction.

3. Formal deduction Students are able to understand deductive geometric proofs in the framework of an axiomatic system. They differentiate between necessary and sufficient conditions and they see relationships among properties. They are able to develop a proof, not just memorize it.

4. Rigor At this final level, geometry can be seen purely in the abstract. Students can operate in different axiomatic systems. This level has never been described in much details, so far.

Properties of Van Hiele’s levels of geometric thinking

0. Sequential Students cannot be at one level without having gone through the previous one. They must go through the levels in order.

1. Advancement Progress from one level to the next depends more on instruction than on age; a teacher’s method may enhance progress, but it may also retard or prevent the transition to the next level. One should also be aware of false improvements: it is possible to teach bright students facts above their level, such as memorizing the area formula of a geometric figure when they do not yet know the proper context. In such situations, what has really happened is that the material has been reduced to a lower level, and a thorough understanding has not occurred.

2. Intrinsic and extrinsic At each level, concepts that appear in
an intrinsic way are seen as extrinsic at the next level. At the base level, figures are actually determined by their qualities, but someone thinking at that level is not yet aware of those qualities.

3. **Linguistic** “Each level has its own linguistic symbols and its own systems of relations connecting these symbols” [18, p. 246]. Thus, a relation that is “correct” at one level may be modified at another level. For example, a figure may have more than one name – a square is also a rectangle. At level 1 students do not yet conceptualize the inclusion.

4. **Mismatch** If the level of the student and the level of the teacher do not match, learning will be hindered. This is a very common cause of teaching failure.

The van Hiele’s model is also reflected in the ideas of Hans Freudenthal, author of a powerful work on issues in mathematics education [21, p. 125]:

> It is possible to teach a skillful pupil abilities above his actual level, like one can train young children in the arithmetic of fractions without telling them what fractions mean, or older children in differentiating and integrating though they do not know what differential quotients and integrals are.

The thinking and models of these scholars provide a framework into which activities using Logo can fit well to facilitate the transition from experiential geometry to more formalized levels of knowledge. In a specific study, Clements and Battista found that the use of Logo can enrich students’ geometric thinking [22]. Specifically:

> If Logo does indeed help children become aware of their geometric intuitions, certain Logo environments should facilitate the transition from Level 1 to Level 2 geometric thinking. For example, when children write a procedure to draw a shape that they recognize visually, they must analyze the visual aspects of the shape and how its parts fit together, an activity that seems
to promote Level 2 thinking. Such an activity may also facilitate the development of specific geometric concepts such as angle, angle size, and related arithmetic ideas, especially given the critical role that turtle rotations play in forming geometric figures.

The role of “materials”

The need to take great care in matching method, content, and vocabulary to the level of the students has led other leading scholars to place great emphasis on tools, teaching aids, and work environments. We can mention, among others, Maria Montessori’s well-known teaching “materials”, Emma Castelnuovo’s poor materials for learning mathematics, and Seymour Papert’s Logo. They all offer ways to approach mathematical concepts through concrete experiences that are meaningful to children’s perceptions.

The Montessori method is famous for its use of materials specifically designed to stimulate abstract thinking:

The Montessori Classroom is arranged into areas, usually divided by low shelving. Each area has “materials”, the Montessori term designating educational objects, for working in a particular subject area (art, music, mathematics, language, science, and so on). This contrasts sharply with traditional education, in which learning is derived largely from texts. Books become more important as tools for learning at the Montessory Elementary level, but even there, hands-on materials abound. Dr. Montessori believed that deep concentration was essential for helping children develop their best selves, and that deep concentration in children comes about through working with their hands, hence materials. [24]

Montessory is mainly focused on the preschool and elementary level, although some materials can be used at higher levels, such as the binomial

---

4The levels 1 and 2 mentioned by Clements and Battista correspond to the levels 0 and 1 we described on page 11, because these authors adopted a renumbering where 0 corresponds to a new so-called prerecognition level. The subsequent levels are numbered accordingly.
and trinomial cubes, which can be used as 3D puzzles at the elementary level, but can also be recalled at the secondary level when dealing with the binomial formula.

A similar emphasis on physical materials to support the learning of abstract concepts, especially in mathematics, is found in the work of Emma Castelnuovo, who was a renowned mathematician as well as a secondary school teacher. Emma Castelnuovo was the foremost scholar of mathematics education. She worked and wrote primarily for the Italian school, but her work immediately resonated abroad, winning the esteem and collaboration of leading scholars such as Jean Piaget and Bruno De Finetti. For several years she was president of the Commission Internationale pour l’Étude et l’Amélioration de l’Éducation des Mathématiques.

In her vision, mathematical intuition cannot come from memorizing small rules, but is closely linked to concrete actions: children’s attention must therefore be directed to objects built with their own hands and to the operations that can be performed on them with very simple materials: strings, rubber bands, sticks, paper cutouts, paper clips, ... A material that allows continuous transformations, such as the gradual transformation of a square into a rhombus by changing the angles between the rods. The emphasis on the role of physical materials is similar to that of Montessori. One of her last books, published at the age of 94, is entitled “The Mathematical Workshop, Reasoning with Materials” [25].

Emma Castelnuovo, following Maria Montessori’s vision, made a decisive contribution to breaking the dead end in which mathematics education had been stuck for more than a century. Especially in the case of geometry, the students were directly confronted with the Euclid’s axiomatic order. This was a consequence of the long process of abstraction in mathematics that culminated between the 19th and 20th centuries with the work of Henri Poincaré and David Hilbert. Castelnuovo’s criticism was directed at the mathematical world for being too passive in ignoring the educational problems inherent in the transfer tout court of scientific knowledge in school curricula. There were important exceptions, as in the case of the aforementioned Dina and Marie Pierre van Hiele, Efrem Fischbein, Hans Freudenthal, and many others, but always a minority. The work of these scholars, especially Montessori and Castelnuovo, has indeed influenced the attitudes
CHAPTER 1. LEARNING, EMOTIONS, COMPLEXITY: NOT JUST WHAT, BUT HOW

of educational authorities around the world and has produced a number of success stories, but surprisingly the problem of teaching mathematics still persists [12, 13].

Obviously, Maria Montessori, but also Emma Castelnuovo, could not include digital technologies among their materials, except for a few recent fragments. It would be interesting to know their views on this, although much suggests that they would at least consider giving precedence to manual activities - a view we share. This is not to say that we cannot benefit from the use of technology, as long as we can do it as “hands on” as possible, and without too many intermediaries based on interfaces and applications that are also “too easy” and “too smart”. If you will, this is one of the main themes of this book, and it was certainly the focus of Seymour Papert’s work.

Mindstorms, the famous seminal book in which Papert discusses in detail why and how to use Logo, is largely a book about learning geometry [1]. Seymour Papert’s view was largely identified with that of constructivism, of which Jean Piaget is one of the fathers, but differed in the role assigned to the learning context. A difference that found its place in the variant of constructionism. It is worth referring to Papert’s own words:

I take from Jean Piaget a model of children as builders of their own intellectual structures. Children seem to be innately gifted learners, acquiring long before they go to school a vast quantity of knowledge by a process I call “Piagetian learning”, or “learning without being taught”. For example, children learn to speak, learn the intuitive geometry needed to get around in space, and learn enough of logic and rhetorics to get around parents — all this without being “taught”. We must ask why some learning takes place so early and spontaneously while some is delayed many years or does not happen at all without deliberately imposed formal instructions. If we really look at the “child as a builder” we are on our way to an answer. All builders need materials to build with. Where I am at variance with Piaget is in the role I attribute to the surrounding cultures as a source of these materials. In some cases the culture supplies them
in abundance, thus facilitating constructive Piagetian learning. For example, the fact that so many important things (knives and forks, mothers and fathers, shoes and socks) come in pairs is a “material” for the construction of an intuitive sense of number. But in many cases where Piaget would explain the slower development of a particular concept by its greater complexity or formality, I see the critical factor as the relative poverty of the culture in those materials that would make the concept simple and concrete. In yet other cases the culture may provide materials but block their use. In the case of formal mathematics, there is both a shortage of formal materials and a cultural block as well. The mathophobia endemic in contemporary culture blocks many people from learning anything they recognize as “math”, although they may have no trouble with mathematical knowledge they do not perceive as such. [1, p. 6]

The first environment Papert created to implement a learning environment is Logo and it is also the one he spent the most time on during his lifetime, but he did not stop there. His contributions extended to Scratch and educational robots.

**Logo or Scratch?**

We want to avoid the easy but sterile dichotomy that sets Logo’s text language against Scratch’s block language. The opposition stems from the fact that so-called “coding in school” today is completely identified with Scratch. This is understandable because Scratch is the result of Logo’s evolution into today’s reality of the web, graphical user interfaces, and social environments. In fact, Scratch is not only a visual programming language, but also a web environment for sharing and remixing projects. In addition, Scratch offers a wide range of tools and plug-ins to interface with the most popular microcontrollers and robots on the market, far beyond what Logo allows.

So if it were simply a matter of choosing the environment, there would be no question about choosing Scratch. But that is not our goal. Instead, we are interested in studying the role of programming as a tool for learning
mathematics and exploring various areas of scientific inquiry. From this perspective, we prefer to work in the conceptual context of Turtle Geometry described by Abelsson and diSessa, which is based on a basic version of Logo, with a limited set of commands, but perfectly adequate to propose a wide variety of scientific explorations. Their text is complex; the idea of this book is to bring it to a level and in a form accessible to the school world, primary and secondary, with appropriate adaptations and some expansion. But the basic idea of Turtle Geometry, of a very simple language that is easily portable to other contexts, remains. From this point of view, the choice of the specific software environment is secondary and, if you like, left to the user’s preferences. Since more or less everything that can be done in Logo can also be done in Scratch, the reader is free to code the Logo examples proposed in this book in Scratch. Of course, since the latter is a block language, porting requires rewriting the code in blocks. On the other hand, it is much easier to port the Logo code to Python, for example in environments like TigerJython, or directly to Python and its regular “turtle” module.

In essence, what we are emphasizing here is the use of Logo, the Turtle Geometry language, in any environment: one of the available Logo implementations, Python, Scratch, or other similar programming languages.

Referring to van Hiel’s levels, Logo can be used in school starting from level 1 (analysis), where students begin to recognize the properties of figures through observation and experimentation. The practice of programming in the context of Turtle Geometry can facilitate the transition from Level 0 to Level 1, by consolidating the knowledge of the properties of figures, as well as the transition from Level 1 to Level 2, by recognizing the relationships between properties and between figures.\(^5\)

\(^5\)Although the van Hiele model clearly specifies that the progression from one level to the next depends much more on the type of instruction than on age, it is clear that there is a correlation, albeit one that needs to be interpreted flexibly. Nesimović and Pjanić have suggested the following match [23]:

0. **Visualization** From preschool age to the 2nd grade of primary school
1. **Analysis** From the 3rd grade to the 5th grade
2. **Informal deduction** From the 6th grade to the 9th grade
1.2 Fighting the “School of Mourning”

In the beginning of his Method, Edgar Morin talks about the “School of Mourning” [26, Pag. 8]:

The school of research is a school of mourning. Every neophyte who devotes himself to research is led to a fundamental renunciation of knowledge. They are convinced that the age of Pico della Mirandola belongs to the past of three centuries ago, that it is now impossible to build a vision of man and the world together. They are shown that the growth of information and the increasing heterogeneity of knowledge exceed any storage and processing capacity of the human brain. They are assured not to complain about this, but to rejoice in it. Therefore, they must devote all their intelligence to the increase of that determined knowledge. They are included in a specialized team, and in this expression the underlined term is specialized, not team. Once specialists, researchers are offered the exclusive possession of a fragment of the puzzle, but the global picture eludes everyone.

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 into 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 place the subjects in their context. The omission of connections and intersections between disciplines deprives them, and prevents us 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 schools about the most disruptive and pervasive outcome of twentieth-century scientific inquiry: the intimate interweaving of certainty and uncertainty that pervades all areas of science.

3. Formal deduction Secondary school
4. Rigor Tertiary education
CHAPTER 1. LEARNING, EMOTIONS, COMPLEXITY: NOT JUST WHAT, BUT HOW

The most spectacular presence of uncertainty on the scientific scene is probably that of quantum mechanics, whose laws cannot be “understood” in the sense that we understand them in ordinary life. We understand them in the sense that we accept them, and for a very simple reason: because they work, and as long as they work. 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, as in the case of so-called chaos.

Chaos occurs in a wide variety of contexts, from the simplest, such as the double pendulum, to the movement of celestial bodies. Or, in more complex milieux, such as weather events — the famous butterfly effect — population dynamics and the myriad ubiquitous fractal forms of nature.

In fact, we now know that chaos is the rule in nature, while order is the exception.

Uncertainty has contaminated even fields that seemed absolutely immune, such as that of mathematical logic. As in the case of Kurt Gödel’s theorems on the 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.

6There are moons, such as Saturn’s Hyperion or Pluto’s Nix and Hydra, whose rotational axes have a chaotic orientation in time, which 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.

7In 1972 Edward Lorenz, a professor of meteorology at MIT, presented a paper entitled: “Predictability: Does the Flap of a Butterfly’s Wings in Brazil Set Off a Tornado in Texas?” While performing numerical simulations of meteorological dynamics, Lorenz found that in such complex systems, even tiny errors in initial conditions can diverge wildly, making prediction impossible. His findings changed the course of science forever.

8The logistic equation is a famous example of how chaotic behavior can emerge when 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 when it is close to the maximum possible value allowed by the context.
1.2. FIGHTING THE “SCHOOL OF MOURNING”

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 tracing the causal factors that led to a series of observations. Ill-posed problems are those where very small perturbations in the data cause huge variations in the solutions. Tomographic medical images, such as CT or PET, are solutions to inverse problems\(^9\) that are ill-posed. For example, the PET problem is more ill-posed than the CT problem — the consequence is that PET images are worse because the ill-posedness and the quality of the data do not allow for more spatial resolution.

Many of the examples above may be difficult to suggest in school. But there are ways. For example, chapter 6 (Section 6.3) deals with the problem of measuring the length of coastlines, which is difficult because of their fractal nature. It is a multidisciplinary exploration that secondary school students could undertake with the help of their teachers. The startling difficulty of such a seemingly insignificant task could be a good way to set sail into the ocean of uncertainty.

This is not a book about computer science, strictly speaking. It is a book about science, which can be written because computer science exists. It is a book for exploring some specific aspects of science, using a computer and an appropriate programming language, through personal engagement, learning by doing, and discovery. In the following, we will briefly describe the contents of each chapter.

In the next chapter (2), we will discuss the software language and development environment that the reader will be using while reading this book. It is not just a chapter about specific software tools, but an opportunity to make some important remarks about languages and development environments. They play an important role in this book. Of course, you can just read it to get a general idea. But in order to be able to suggest some of these activities to students, you have to do them yourself. So this is the first point: to make full use of these resources, you have to read and try them yourself,\(^9\)

---

\(^9\)The mathematical representation of tomographic techniques is given by the Radon transform. It relates a distribution in an n-dimensional space to all its possible \((n-1)\)-dimensional projections. The inverse Radon transform gives the original distribution once the knowledge of its projections is given. This is a typical ill-posed problem.
hands on. The second point is related to the kind of tools I’m proposing, which are free software or tools that are made freely available by academic groups. Access to education is a key pillar of modern democracies. All western constitutions state the importance of giving all citizens access to educational resources in order to achieve an adequate cultural level. A key point, especially as far as the lower grades of instruction are concerned. Not in every country, not in every region, and not in every family can access to expensive educational tools be taken for granted. The use of free or low-cost tools plays a crucial role in such a broad perspective. Not only that, but the use of the most widely used open standards allows for a long-term investment in learning.

The beginning of Chapter 3 is about the first steps and how to guide children along them, starting with physical activities. This part is inspired by Papert’s concept of syntonic learning, in which the teacher exploits the link between physical and intellectual activities. It then shows how a fairly complex idea can be hidden behind even a simple physical activity. The rest of the chapter explores variations on the circle, namely spirals, polygonal approximations, and orbits. This chapter can also be read from the perspective of Bruner’s spiral curricula, where the circle is revisited in increasingly complex ways, starting with children’s play.

The following chapters all fall into the world of simulations. In all examples, the computer is used as a tool to explore scientific concepts in a hands-on way. Simulations themselves are an important part of science education because they are a crucial tool of modern science. Nowadays, numerical approaches play a fundamental role in a wide range of problems in all fields of science. The speed and memory of computers make it possible to simulate phenomena that would be far too complex to tackle in any other way. In biology, there is even a specific term to describe experiments conducted using computer simulations: in silico instead of in vitro experiments.

In chapter 4 we explore the computational way of studying and solving classical dynamical problems. This writing was largely inspired by Sherin’s work on comparing programming languages and algebraic notation as expressive languages for physics [30]. The codes are written in the simplest way possible, so as not to intimidate readers and to encourage them to push
themselves into further exploration. The examples are variations on the problem of the free fall of a body, where different 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 a non-zero horizontal initial velocity or a large distance from the Earth’s surface, or a combination of both to compute the orbits of celestial bodies around the Sun.

While chapter 4 sticks to the classical deterministic description of phenomena, in the following two chapters we introduce uncertainty in terms of Morin’s big picture. In chapter 5 we introduce randomness for the simulation of animal behavior. The behavioral models used in the exercises are quite naive, but perfectly adequate to highlight basic features of living systems, such as the power of feedbacks and the deep entanglement between randomness and determinism in complex systems. The chapter begins with an introduction to the random choices in the turtle’s behavior. Then three types of models are discussed: a simple olfactory model, some variations on a binocular vision model, and a multiple-turtle interaction model for simulating phenomena such as the spread of a disease in a population. The multiple-turtle study also provides an opportunity to introduce the concept of multitasking through a simple exercise.

Uncertainty plays a different role in chapter 6, except in some cases where randomness has also been added. We enter this new context through the concept of recursion to explore a variety of fractal shapes. Different fractals can be used to emphasize different aspects. The basic “sticky tree” opens the door to a reflection on infinity and infinitesimal. The Koch fractal allows you to explore the impossible task of measuring a coastline. This is achieved by extending in a relatively simple way the seminal work of Benoit Mandelbrot on the concept of fractional dimension [31], the distinguishing geometric feature of fractal shapes. The blurring of integer dimensions that characterizes fractals is the kind of uncertainty we explore in this chapter. The final examples show how amazing the fractal reproduction of plant shapes can be.

The examples proposed in chapters 5 and 6 show how an appropriate mixture of simple random choices and a few trivial deterministic rules can lead to realistic natural representations, both dynamical and geometric. The whole is intended to give a tangible idea of Morin’s saying that the
CHAPTER 1. LEARNING, EMOTIONS, COMPLEXITY: NOT JUST WHAT, BUT HOW

human condition is about navigating in an ocean of uncertainty through archipelagos of certainty [32].
Bibliography


“When we were little... we went to school in the sea. The master was an old Turtle — we used to call him Tortoise. ”

“Why did you call him Tortoise if he wasn’t one?” Alice asked.

“We called him Tortoise because he taught us,” said the Mock Turtle angrily.

“Really you are very dull!”

— Lewis Carrol, Alice’s Adventures in Wonderland

When it comes to computer programming, languages should be distinguished from development environments. In this book, we have basically used one language, Logo, but two development environments: LibreLogo and TigerJython. Both environments provide a fairly direct implementation of the Logo language.

Logo originated in the 1960s when computer scientists Cynthia Solomon, Wally Feurzeig, and Seymour Papert joined forces in a project to make math and computer science more accessible to children. The project was started
in 1966 at “Bolt, Beranek and Newman Inc.”, but in 1969 the three scientists met at the MIT Artificial Intelligence Lab, where they formed the “Logo Group”. Seymour Papert was the leader of the project because of his unusual background. Born in South Africa in 1928, he studied mathematics in Johannesburg and then at Cambridge. He went on to earn a Ph.D. in psychology and worked with Jean Piaget at the University of Geneva from 1958 to 1964.

2.1 What we mean by Logo

It is good to clarify what we mean by Logo here. Because there are many Logo environments: at the time of this writing, 309 versions of Logo have been counted [1], and of all these versions, 75 are still active. Most of these implementations are in fact a full-fledged computer language in their own right. They may offer a variety of more or less fancy features, but they all have in common a set of basic instructions that we might call “turtle commands”, which are intended to trace a path in a given space - usually two-dimensional, in some cases three-dimensional - directing an imaginary entity, a turtle in fact. It is important to make it clear that when we refer to Logo in this book, we mainly mean this basic set of instructions, which is basically the same as that used by Seymour Papert in his seminal book “Mindstorms” [2], and by Harold Abelson and Andrea diSessa in their book entitled “Turtle Geometry” [3]. Let’s have a look at their version, which they call “Turtle Procedure Notation” (TPN):

Turtle Procedure Notation is meant to be a consistent and readable way to describe turtle programs independent of the quirks, details, and limitations of common computer languages, yet in such a way that the programs can be readily translated into whatever computer language is available. [3, pag. 393]

The number of commands available in TPL is quite small:

Also the syntax is very simple. Let us see how a square of side SIDE is coded in TPL:
2.1. WHAT WE MEAN BY LOGO

Table 2.1: Basic Logo commands

<table>
<thead>
<tr>
<th>Commands</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARD L</td>
<td>Move straight ahead by the length stored in variable L</td>
</tr>
<tr>
<td>BACK L</td>
<td>Move straight back by the length stored in variable L</td>
</tr>
<tr>
<td>RIGHT A</td>
<td>Rotate right by the angle stored in variable A</td>
</tr>
<tr>
<td>LEFT A</td>
<td>Rotate left by the angle stored in variable A</td>
</tr>
<tr>
<td>REPEAT N</td>
<td>Repeat following instructions by the number stored in variable N</td>
</tr>
<tr>
<td>TO NEWCOM</td>
<td>Pack set of instructions in the single command NEWCOM</td>
</tr>
<tr>
<td>. . . END</td>
<td></td>
</tr>
<tr>
<td>PENUP</td>
<td>Doesn’t draw when moving</td>
</tr>
<tr>
<td>PENDOWN</td>
<td>Draw while moving</td>
</tr>
<tr>
<td>IF . . . THEN . . . ELSE</td>
<td>Conditional expressions</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>

Interestingly, Abelson and diSessa’s TPL uses indentation to identify blocks of code in procedure definitions and cycle blocks, just as we do in the modern Python language. The simplicity of the code helps both mathematicians and novices. In the first case, since blocks of code can be treated as mathematical objects in theorems, the notation should be as simple and concise as possible. On the other hand, beginners benefit from straightforward codes because the syntax complexity of full-fledged computer languages is a major obstacle for inexperienced people.

For comparison, and also to give an idea of what Abelson and diSessa mean talking about “quircks” and “details”, I show the same piece of code in the two variants of Logo used in this book. First LibreLogo:
CHAPTER 2. SOFTWARE LANGUAGES

Here we see that the body of a new command definition must end with an END statement, and that chunks of code referenced in a REPEAT statement must be enclosed in a pair of square brackets.

Now let's see how we should write the same code for Python in the Tigerython flavour:

```
repeat 4:
    forward(100)
    right(90)
```

Listing 2.3: Python

In Python, the body of both procedure definitions and repeating cycles are denoted by indentation, as we said before. This is a nice feature, which partly explains the great success of this language, but the notation requires some more control characters, even in this small piece of code, namely parentheses and colons.

Seymour Papert used to say that educational programming languages should have a low floor and a high ceiling, i.e. be easy to start with, yet powerful enough to allow for complex projects as the learner’s skills grow. In fact, as far as we know, the Logo language version in LibreLogo is the closest to TPL, and as such the most suitable for beginners and for first programming experiences in school. For this reason, in chapter 3, which is dedicated to guiding beginners with a smooth progression from their first very basic steps to more complex cases such as creating polygons and spirals, all the code examples are written in LibreLogo. In the following chapter (4), where the code becomes more complex, the first example, concerning the simulation of a free-falling body, is shown in both variants, LibreLogo and Python; then the rest of the examples are written in Python. However, in each of the following chapters, where the use of Python predominates, the first

---

1Here and in the following, Python always means the TigerJython implementation.
basic codes are shown in the LibreLogo version as well and then developed further in Python. This is about getting comfortable with switching between different code types and choosing whichever is best for a given job.

Let us elaborate a little more on the role of TPN in comparison with that of the two variants we use in this text. Seymour Papert did not need any further instruction to thoroughly discuss the use of Logo at school in “Mindstorms” because he focused on the development of mathematical thinking. Logo was intended as a tool to bring children closer to the mathematical world in a natural way. Therefore, it is good to remember that if we want to focus on the development of logical-mathematical thinking, then the TPN command set is sufficient.

2.2 The Logo culture

From the very beginning, Logo was associated with the turtle, a kind of robot that could move and possibly 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. But Papert was also referring to an earlier turtle designed by Williams Greg Walter (1910-1977), a prominent neurophysiologist and cyberneticist. At the time, around 1950, the robots Greg Walter built were known as cybernetic machines. Cybernetics was a multidisciplinary scientific thought movement promoted by a group of eminent scientists including Norbert Wiener, John von Neumann, Claude Shannon, Alan Turing and many others, including William Greg Walter. The basic idea behind cybernetics was the feedback loop. This has been recognised as the basic mechanism for self-regulation in both machines and living systems. We will explore feedback in several contexts, such as the simulation of the sense of smell on page 164. Walter’s turtle was an electromechanical device that “imitates life” [4]:

---

2 See page 164 for an example of feedback.
3 The motto we used in this chapter was also the one Walter put in his paper “An imitation of life” published in the Scientific American in 1950. This is probably the origin of the turtle metaphor for physical or virtual entities capable of imitating life or creating geometric figures.
Figure 2.1: Seymour Papert shows one of the early versions of Logo, when it was some sort of robot for drawing.
2.2. **THE LOGO CULTURE**

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 allows the exploration of their crucial role in living systems. The turtle we simulated on page 176 with binocular vision and light intensity perception, is essentially a replica of Walter’s turtle.

The Logo language is not limited to the world of education, but is also used in other contexts where it is appropriate to generate 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 the development of herbaceous plants. [6]

But beyond that, in the 1970s and 1980s, phrases like “Logo movement,” “Logo culture,” “Logo as an educational philosophy,” 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 that Logo 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. [7, pag. 163]

Everyone agrees that Seymour Papert was the leading influence in the development of such a computing culture. However, it is important to recognise that this culture as such was closely shared by a fairly large group of scholars involved in teaching or researching artificial intelligence. See, for example, the perspective of Cynthia Solomon:

---

4We will see examples of this in Chapter 6.
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. [8, pag. 195]

Or the “Cultural Glossary” by E. Paul Goldenberg [9, pag. 210]:

![Diagram of concepts at stake around Logo in 1982.](image)

The important fact is that all of these people, and many others, although from very different backgrounds, shared a broad vision. A vision that led
2.2. THE LOGO CULTURE

them, starting with the artificial intelligence language Lisp, to create Logo as “a LISP-like language and a laboratory for loose, lifelong learning about learning” [7, pag. 163].

But has the Logo culture permeated the education system? Not quite, if we’re talking about the general use of Logo in schools. On the other hand, one cannot speak of a failure since, as we mentioned earlier, 309 versions of Logo have been counted so far [1], and of all these versions, 75 are still active.

Even programming systems based on visual blocks can be seen as a legacy of Logo. Scratch [10], the best known of these, was developed at the MIT Media Lab under the direction of Mitchel Resnick, who did his PhD under Seymour Papert and Hal Abelson. However, this branch of educational languages did not come from the academy alone, 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 [11] and Snap! [12].

These systems, in various shades, are very powerful and allow the development of multimedia projects. However, the emphasis has shifted from expressions such as “Logo culture”, “Logo as a philosophy of education” or “Logo as a laboratory for lifelong learning about learning” to slogans such as “design, create, remix” or “imagine, program, share”. In short, more social, less cultural. Indeed, there are studies comparing text-based and block-based languages in education. The results show that there is a small advantage 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 significant differences in in-depth understanding of what an algorithm does [13]. In addition, students demonstrated higher levels of self-esteem with Logo [14] and greater confidence and enjoyment in programming [15]. A fairly obvious result of these studies is that as students become more skilled, they take more advantage of text programming. As a result, new mixed systems have emerged that make it easy to switch between text and block languages, such as Pencil [16].

However, this whole line of research focuses mainly on programming skills in the narrow sense, with programming in Java or Javascript being
the ultimate goal. In fact, in the block language culture, the focus is on programming, which is widely referred to as “coding”. A very popular word these days, both on the Internet and in focus groups on technology and schools. The hype is considerable, but the widespread impact in the school world is far from general and homogeneous. And it is usually 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 different areas of science in the new ways made possible by computer science, taking advantage of children’s natural attitude as budding scientists. Finally, the problem is cultural because it concerns the tendency to “teaching in the box”, that prevails in all disciplines, even in teacher training. In this broad sense, the problem of using computers to promote scientific thinking remains unchanged over time: in the era of the Logo movement as well as today. The problem is one of teacher training: teachers trained in the box, teaching in a box-free world. The Logo movement had a sound cultural goal but the school world was not ready to embrace it. The logo movement had a good cultural aim, but the world of education did not go along with it. The hype of visual programming is a successful Internet fact but the school world is disoriented, missing the meaningful point.

There is another problem. At the beginning of this chapter, I wrote that programming languages must be distinguished from development environments. With visual programming systems, this is no longer true. Each system has its own web interface, and each has its own features, such as its own export format. For example, you cannot port a program from one system to another using a simple text editor, as in the case of Logo dialects. So to talk about a visual language is to talk about a specific web environment. How long can a web environment last? Will it be possible to recover and recycle all the scripts that have accumulated in the system when it is shut down one day? Abelson and diSessa wrote “Turtle Geometry” some forty years ago. Today, the wealth of logo examples in the book are still easy to copy, even with voice typing and a few tweaks. It would be difficult to do the same with a collection of block-code scripts, unless there is a system for
exporting them to a standard text format such as Logo.

2.3 Turtle geometry

Logo is a programming language designed to create geometric figures by drawing them. The turtle metaphor allows you to imagine drawing by walking along paths. It is important to realise that when we use Logo we are actually dealing with a particular kind of geometry, turtle geometry. Actually, there are different kinds of geometry. The best known, at least at school level, are Euclidean and Cartesian geometry. The axiomatic Euclidean structure is the basis of geometrical knowledge. Cartesian geometry combines Euclidean geometry with algebra, so that geometric shapes are determined by knowing the coordinates of the points that make them up. Turtle geometry is also a geometry, but in this case the shapes are defined by the Logo commands used to create them.

While the study of Cartesian geometry leads to graphs and equations, turtle geometry introduces other mathematical ideas, some of which are very suitable for introducing children to mathematics.

In Euclidean geometry, the point is the unique dimensionless entity, or the “object without parts”. In Cartesian geometry a point in a plane is given by an ordered pair of numbers \(x, y\), in three-dimensional space by three numbers \(x, y, z\), and so on, in n-dimensional spaces by \((x_1, x_2, \ldots, x_n)\) numbers. In turtle geometry, the point is the turtle itself, which differs from the Euclidean or Cartesian case in that the turtle is fully described once we know not only its position but also its direction.
The expression “fully described” brings to mind the notion of state, a fundamental concept in science\(^5\). The state of the turtle consists of position and direction. The fact that the fundamental object is characterised by such a state is what makes turtle geometry peculiar. It is also what makes it suitable for teaching geometry to children, because working with the turtle allows us to think in terms of movement and, in the case of children, allows them to identify with it, giving rise to Papert’s concept of \textit{syntonic learning}.

\textbf{Equations vs Procedures}

Euclidean geometry consists of postulates and theorems. Cartesian geometry consists of sets of coordinates or equations. The basic elements of turtle geometry are procedures. Let’s take a couple of examples, starting with a square.

In Euclidean geometry the square is a plane figure with four equal sides and four equal angles. In Cartesian geometry a square is defined by four pairs of numbers such as \([(x_1,y_1),(x_2,y_1),(x_2,y_2),(x_1,y_2)]\). In turtle geometry, the square is a code like the one you have already seen on page 42:

\(\)\(^5\)The concept of a system’s state is one of the powerful ideas mentioned in this book. We will discuss it in more detail on page 69
2.3. TURTLE GEOMETRY

REPEAT 4
   FORWARD 100
   RIGHT 90
END

Listing 2.4: A square in TPL

Just a piece of code, the logo code you have to write to get a square. But how can we compare an expression like \([(x_1, y_1), (x_2, y_1), (x_2, y_2), (x_1, y_2)]\) with this piece of code? That’s the beauty of math: the capability of working with the most strange objects, once a perfectly set of rules to work with have been established. Do not worry about understanding that code right now. Just focus on the fact that, in the realm of turtle geometry, that piece of code represents a square, and even in a more general way with respect to the \([(x_1, y_1), (x_2, y_1), (x_2, y_2), (x_1, y_2)]\) expression, because it is perfectly adequate to represents a square in any possible orientation. We will come back on this later on.

![Comparison diagram](image)

Figure 2.4: Comparing Euclidean, Cartesian and Turtle representation of a square.

In Euclidean geometry, the circle is the set of all points at a given distance from a given point, called the centre of the circle [5, 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 the set of points whose coordinates \((x, y)\) satisfy the equation \(x^2 + y^2 = r^2\), where \(r\) is the radius of the circle. In turtle geometry, the circle is again represented by a piece of code. And if we compare it with that of the square, we see that there are similarities. We will keep working on this.

```
REPEAT 360
FORWARD 1
RIGHT 1
```

Listing 2.5: A circle in TPL

![Figure 2.5: Euclidean, Cartesian and Turtle geometries.](image)

**Extrinsic vs intrinsic**

A major difference between turtle geometry and coordinate geometry lies in the notion of intrinsic properties of geometric figures. An intrinsic property is one that depends only on the figure in question, not on the figure’s relation to a reference frame. The fact that a rectangle has four equal angles is an intrinsic property of the rectangle. But the fact that a given rectangle has two vertical sides is extrinsic, because an external frame of reference is needed to determine which direction is “vertical”. Let’s say we have commands that can make the turtle draw by sending it to certain positions, given their coordinates.
2.3. *TURTLE GEOMETRY*

To explain the difference, let’s look at a command that is not part of Turtle’s basic set, but is available in all programming systems. In LibreLogo the command `POSITION [X,Y]` sends the turtle to the point with coordinates X and Y. You can use the `POSITION` command instead of using combinations of `FORWARD`, `RIGHT` and `LEFT` commands, but the coordinates of the arrival points must be known beforehand.

Suppose we want to draw a segment of length L, first in the vertical position, then rotated through the angle $\alpha$. Assuming that the turtle starts at position (X₀,Y₀), we obtain a vertical segment of length L by means of this command:

```
POSITION [X₀, Y₀+L]
```

Listing 2.6: Drawing a vertical segment in the Cartesian way

```
XR = X₀ + L * SIN(\alpha)
YR = Y₀ + L * COS(\alpha)
POSITION [XR, YR]
```

Listing 2.7: The same segment rotated by the angle $\alpha$ in the Cartesian way

So if we want to draw the same segment, starting from the same point but rotated by $\alpha$, we must first calculate the new coordinates of the segment end point. And of course, if we want to rotate more complex figures, we have to calculate the new coordinates of all the vertices. A task that requires knowledge of some trigonometric functions and the equations of the Cartesian reference system. With turtle geometry, the situation is quite different and much simpler, since only the initial state of the turtle is required, that is, a position in the plane and a direction.
CHAPTER 2. SOFTWARE LANGUAGES

FORWARD L
Listing 2.8: Drawing a vertical segment in the turtle way

RIGHT A
FORWARD L
Listing 2.9: Same segment rotated by the angle \( \alpha \) in the turtle way

In this case, even the most complex figures can be easily rotated by simply rotating the turtle before drawing the figure.

Global vs Local

The local/global question is especially interesting in the case of small turtle moves, such as the circle. If we try to put ourselves in the turtle’s shoes as it executes the commands to draw a circle, we realize that it is acting in a strictly local scope, with no knowledge of what is happening in the far reaches of the plane. In a sense, the turtle draws the circle “with its eyes closed”, repeatedly making a small step and a small turn, always in the same way, which could be described as moving by small differences.

This is in sharp contrast to the Cartesian representation of the circle, which requires the explicit notions of center and radius, as we saw in Fig. 2.5. Equally global is the Euclidean description, according to which the circle is the locus of points equidistant from a given center, whereas the turtle representation does not need to refer to this particular “distant” point.

This concept of constructing a figure by strictly local means is fundamental to what we call differential calculus, which is usually covered at the undergraduate level. From this perspective, turtle geometry is a differential geometry, since figures are constructed by means of differences. We are
going to explore this concept more deeply in Sect. 3.2 when discussing how to get children involved in creating circles through physical activities.

2.4 **Free software tools**

A programming language for use in schools should be easy to use initially, but powerful enough to allow the development of increasingly complex projects as skills develop. It must be of a standard that is as widely used as possible, so that projects can be easily shared and maintained over time. It must be cost-effective to maximise the inclusion of individuals in socially fragile contexts.

Both are text-based programming languages with a syntax very close to Abelson and diSessa’s basic Turtle Procedure Notation. This makes project sharing very easy, even using elementary communication systems such as email. The similarity to TPN facilitates porting to other programming contexts: for example, it is very easy to translate the examples published in Abelson and diSessa’s 1980 book Turtle Geometry from TPN to LibreLogo or TigerJython. Both systems consist of applications that can be downloaded to one’s own computer and are released in the form of “free software”, which is a crucial point because it involves both no cost and an important accompanying educational message. To understand this last point, let us focus on the concept of free software.

Free software is defined by four kinds of freedoms: [17]:

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 have highlighted two sentences in Freedom 2 and Freedom 3, respectively, because they express an ethical rather than a technical concept. This is what free software is all about. It should be noted — and this is often misunderstood — that the concept of open source is different in that it does not include the ethical aspect: software is said to be open source if its source code is made available along with the executable version of the program, but no mention is made of ethical aspects. Free software is developed by individuals, often in free communities of developers or non-profit organisations that share a common vision. Open source, on the other hand, can be developed by private economic actors who adhere to the collaborative development paradigm because it fits well with their marketing strategies. Indeed, there are companies that develop open source projects alongside their traditional proprietary products.

Free software is good in an educational context because it implicitly teaches to work together for the benefit of all, not to crack proprietary software because it is against the law, to save money for the public good in a perfectly legal way, and to value tools that everyone can afford. This last point is important in the context of public education, as most national constitutions contain articles establishing the right of access to education for all citizens, without exception. This right is enshrined also in Article 26 of the Universal Declaration of Human Rights [18]. It’s worth remembering:

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
2.4. **FREE SOFTWARE TOOLS**

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 question of skills, but also as a fundamental tool for raising citizens who are aware of the basic principles of democratic coexistence in an increasingly complex world, then the choice of tools and technologies used at school also plays an important role.

**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. Both LibreOffice and LibreLogo are free software. LibreOffice is widely used because of its wide range of features, comparable to those of the most popular commercial competitors. It also has an unprecedented level of internationalisation, thanks to the contribution of developers from all over the world, including ethical and linguistic minorities.

LibreLogo was essentially created as a tool for creating graphics with Logo. The winning idea is to have both the editing of the code and the graphic result in the same environment, which is a normal page of a word processor. With LibreLogo, figures are created as vector images, just as

---

6 LibreOffice can be downloaded from https://it.libreoffice.org/. When you start LibreOffice for the first time, the LibreLogo toolbar is not active. You must therefore activate it using the corresponding menu command: **View → Toolbars → Logo**. Once this has been done, you will need to close and restart the program. You will then see the LibreLogo toolbar among the other toolbars.

7 LibreOffice is licensed under the free software Mozilla Public License v2.0 (impressum in https://www.libreoffice.org/get-help/frequently-asked-questions/) and LibreLogo under Mozilla Public License v1.1 (https://en.wikipedia.org/wiki/LibreLogo).
they can be created with the graphics tools of the LibreOffice word processor Writer. For this reason, LibreLogo offers a considerable number of commands designed to describe the graphic appearance of the signs drawn with the basic turtle commands, such as width, continuity and colour of the turtle trace, colouring of the inside of the figures, etc.

Examples of graphics produced by primary education students with LibreLogo

The accompanying site shows various examples of graphics produced by primary education students in a workshop dedicated to learning Logo as a teaching tool in the classroom. The aim is to put the students in the same situation as their future pupils and to encourage them to pursue a goal of their own choosing. After adequate preparation, many students become enthusiastic and produce very interesting and often surprising results, even for the teacher.

LibreLogo is an exception; normally, in all languages, programs are developed in an integrated development environment (IDE). These are environments that facilitate the editing of programmes and provide a set of tools for the development of complex projects.

LibreLogo is oriented towards the production of graphical objects that are included in a text file in ODT format. Despite this vocation, we will explore ways to produce something else, for example, we will produce simulations of physical and biological phenomena.

TigerJython

TigerJython is an integrated development environment (IDE) for writing Python programs. Like LibreLogo, it is also free software. Python is a notoriously powerful and widely used language. Apart from its clean syntax, Python’s success is certainly due to its very high modularity, which means that the language can be used for an extremely wide variety of tasks. TigerJython is an IDE specifically designed for learning Logo programming and is intended for non-experts. It therefore avoids all the complications

---

*TigerJython is licensed under the Mozilla Public License v2.0 (https://github.com/TigerJython/TigerJython/blob/master/LICENSE). It can be downloaded at https://tigerjython.ch/en*
associated with setting up a standard Python environment or using a professional IDE, with workspaces, project management tools and so on. It has just a simple editor and allows users to write Python code and run it in a straightforward way. On the other hand, there is a rather sophisticated error message system, which is important for programming beginners. In addition, TigerJython makes it possible to exploit the great potential of Python, for example to realise simulations with several interacting turtles — we will see examples — and it has a large set of libraries for interacting with external devices and robots. For these reasons, we have chosen to present the examples in this book in the TigerJython version.

**In which chapters we will use LibreLogo and in which TigerJython**

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 WebTigerJython 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 A.2 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. 9

Alternative programming environments

Precisely because the examples proposed in this book are developed using very simple text languages, the reader can reproduce them in other systems with little effort. I will briefly mention some of the most important ones.

Examples made with the alternative programming systems The accompanying website shows various examples of codes and relative results in all systems.

Plain Python with “turtle” module: the safest Plan B

Anything that depends on software is subject to a degree of volatility, often very high. This applies to both downloadable software and software in the form of web services. In the case of LibreOffice, longevity should be enough, as it is a very popular software used worldwide, provided that the LibreLogo component is maintained in the standard distribution. The TigerJython IDE is probably a growing reality, although it is aimed at a much more limited audience. If we want to think in terms of an alternative plan B, the Python turtle module is probably the most stable alternative imaginable so far.

Usually those who develop projects in Python use very complex environments, but for writing small (and sometimes not so small) educational programmes, one can proceed in a much more direct manner, simply using a minimal text editor, such as Notebook in Windows or Gedit in Linux. For the examples proposed in this book, nothing more is needed, except to include some special instructions in the program, which we will show in the following example for drawing a square.

*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/*
2.4. FREE SOFTWARE TOOLS

# First import the turtle module
from turtle import *

# Set Logo angles
measurement mode:
# initial turtle heading:
upward (north)
# positive angles: clockwise
mode("logo")

# This is how the repeat(4) instruction is written
# in turtle Python
for i in range(4):
    forward(100)
    right(90)

# This is to maintain the graphics window once
the program is run
done()

Listing 2.10: Drawing a square in Python with turtle module

Xlogo

There are alternatives that deal with the transition from block- to text-based languages. I will mention two that use different approaches. The first, Xlogo, is developed and maintained by the Training and Advisory Centre for Computer Science Education (ABZ\textsuperscript{10}). It is a web service\textsuperscript{11} in which four programming environments of increasing level are offered according to Bruner’s spiral curriculum concept [19]. In fact, the four environments make it possible to revisit programming experiences by taking advantage

\textsuperscript{10}ABZ stands for Ausbildungs- und BeratungsZentrum für Informatikunterricht. It is directed by Prof. Juraj Hromkovič at the ETH (Swiss Federal Institute of Technology) in Zurich.

\textsuperscript{11}Xlogo is accessible at https://xlogo.inf.ethz.ch/
of the increasingly advanced possibilities offered by the subsequent levels. The first two levels are block-based, while the second two are text-based.

The first, mini — *programming in a grid*, is a block-based language where the available movements are of fixed length, similar to the popular BeeBot and BlueBot robots. In fact, the system can be connected to a BlueBot\(^{12}\), which can move in fixed steps of 15 cm. The turtle can also turn 90° left and right, and commands can be grouped into repetition cycles of variable length, with a default of 4. This first level is appropriate before children are able to master numbers.

The second level, midi — *blocks with parameters, distances and angles*, is similar to the previous one except that step lengths and angles can be specified explicitly. Programs can connect to robots, except the BeeBot, which only accepts commands of fixed-length.

The third level, maxi — *programming in logo, use your own command*, is the full version of Xlogo. It is text based and is the closest to the original Logo, even in the variables naming conventions: for example “make ”a 100” to define the variable “a” and assign it the value of 100 and “:a” to reference it in subsequent commands. It allows you to use of the most important software constructs, such as the ability to create new commands, i.e. procedures. A short but clear reference manual is available.

The final level, mega — *programming in python, with or without the turtle*, is actually a web version (WebTigerJython) of the TigerJython software we use in this book. In fact, most of the examples shown here can also be run also in this system, without having to download any software to your computer. Even if WebTigerJython can be seen as an entry level general purpose Python environment, it presents itself with a default script which initialize a turtle.

WebTigerJython is therefore Logo-based, so to speak. In fact, the general idea of the Xlogo web system is to provide a smooth transition from block to text programming, where the common thread is the Logo language.

---

\(^{12}\)At the time of writing, it can also be connected to the Root and XBot robots.
Pencilcode

Pencilcode is also a web system based on the turtle metaphor, designed to soften the transition between block and text-based programming, but in a different way from XLogo. At first glance, the layout is similar to other block programming systems such as Scratch or others, and everything is set up to encourage the user to fetch blocks and drag them around a workspace to compose scripts. The distinguishing feature, however, is a central handler that acts as a toggle between the block and text versions of the code. However, the common thread here is the turtle metaphor and not the Logo language, as the text version is based in JavaScript.

If in Xlogo programming is a means to develop logical and mathematical thinking, Pencilcode is more adapted to introduce learners to professional programming, given the choice to base the text version on JavaScript.

Beetle Blocks

Purely block-based languages such as Scratch or Snap! are not included here, as they are difficult to translate into text once the scripts get a bit longer. It is worth mentioning Beetle Blocks, which is a pure block language, but with an important feature: the beetle, which is an alias for the turtle, can move in three-dimensional space. Conceptually, this is nothing new. Abelson and diSessa developed the mathematics of three-dimensional turtle geometry in the 1970s. [3]. About a decade later, a three-dimensional turtle was used in theoretical biology to simulate plant growth. [6]. But these contributions have not been widely used. The three-dimensional generalisation is exciting, and amazing effects can be achieved, but programming the turtle (or beetle) to move in three dimensions is not so easy compared to the two-dimensional case, where only a position (two coordinates) and a heading (one angle) need to be considered. Adding a third dimension is not a big problem in the turtle’s way of thinking (three coordinates), since you don’t have to manage the coordinate explicitly, but the heading is another story. Basically you have to deal explicitly with three angles instead of one. In Beetle Blocks you can rotate the beetle around three axes, called the yaw, pitch and roll axes, with reference to the possible rotations of a ship in the sea or an
aeroplane in the air.
Bibliography


Chapter 3

Basic turtle programming

The Buddha, the Godhead, resides quite as comfortably in the circuits of a digital computer or the gears of a cycle transmission as he does at the top of a mountain or in the petals of a flower. To think otherwise is to demean the Buddha—which is to demean oneself.

— Robert Maynard Pirsig, Zen and the Art of Motorcycle Maintenance

We should always remember that our brains evolved to optimize a multisensory, istic way of learning. We have evolved to learn through our bodies. Not, as Sir Ken Robinson used to joke: — Professors use their bodies mainly to get their heads to conferences!

And this kind of holistic learning is especially important for children. Thus, various kinds of “unplugged activities” involving physical play or practice 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.

Below are some examples of syntonic activities. They were provided by Maestra Antonella Colombo, a former student of mine who now teaches at the primary school of Paderno d’Adda, Lecco, Italy. She has done a great job in her classroom based on our discussions about the concept of syntonic learning. The first activity was to have the children walk along a curved line drawn on the floor:

Figure 3.1: Pay attention to the path...
Or along a hula hoop:

Figure 3.2: One small step, turn a little bit, again and again...
Or by playing games that leave a trail, such as walking barefoot on flour:

![Figure 3.3: “Drawing in the flour”...](image_url)
The next step may then be to have a puppet move along paths on the floor instead of moving itself. This step is in preparation for programming the turtle on the screen.

Figure 3.4: If I would be the puppet...
CHAPTER 3. BASIC TURTLE PROGRAMMING

Very powerful mathematical idea — Isomorphism  We go back 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. Put simply, an isomorphism is a correspondence between two different worlds that preserves sets and relations between elements. The fact that one sees the correspondence between a square “drawn with one’s own body” — for example, by walking along a square figure visible on the ground — and the square drawn by the turtle on the computer screen, is a significant step towards abstraction, a step towards the concept of isomorphism.

First steps

With Logo, students start with a blank page. Of course, you must help them in the beginning. But refrain from giving instructions as much as possible:

— Today we’re going to make something new . . . let’s open this program (LibreOffice for instance).

Then, everyone is presented with a blank page with some fairly familiar text formatting commands at the top. Give them time (always give them time...) to realize where they have landed. Do not explain that this is a word processor. Just suggest that they try something, like typing something, and let them emerge and realize that this is nothing more than one of those programs for writing text, possibly homework or similar stuff:

— So what? Why are we here?

Try to create a sense of mystery. At a certain point, you might suggest typing a certain word: for example¹: FORWARD. Discuss the meaning of this word in English, remembering the translation in your native language. Then give this little statement:

— Go over there, to the left, and press the green arrow . . .

Two things will happen: a green object will appear, could it be an animal, with head and legs? Some kind of turtle? Yes, a turtle, actually, the protagonist of our story, and we will call it “the turtle” in the following. But

¹I used uppercase just for clarity. LibreLogo is “case-insensitive”. In TigerJython you must use lowercase instead, because the environment is case-sensitive.
also a small window appears, telling you that there is an error. It is good to stumble upon an error right away. It's a way to see the problem in a different light: the error as an opportunity to get valuable information:

— Well — you may admit — I forgot something!

Then tell them to add a number, FORWARD 100, and to press the green play button again...

There is a drawing: the turtle at the top of a vertical segment. When working with children, it is good to present the turtle as a friend who can help them to do something new. So the turtle is a kind of creature, not so much because it looks like a turtle, but because it's able to speak, albeit in a peculiar way. We write FORWARD 100 and the turtle draws a segment, i.e. we write a command and the turtle responds by doing something related to the command — a dialog takes place. What should be done is to let
the children feel like they are playing with the turtle, following their innate ability to set their own goals and explore what amazes them. Of course, this takes time, but true learning deserves time — there are no shortcuts. And, of course, again, to achieve this simple drawing, you have to give the children the command to write, but you do not have to say something like

— Now I will tell you how to draw a segment.

Just tell the command and wait...

Probably someone will suggest something:

— Let’s change the number... what else can we do...

Wait for ideas or questions like these, always wait for exploration. Then add something new, like RIGHT 90. At this point, however, you can introduce two handy commands to erase previous drawings and send the turtle home on each new attempt:

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

Listing 3.2: A 100 points step forward then turn right
We got the same drawing, but the turtle turned to the right by an angle of $90^\circ$. Do not explain what 90 means, let them figure it out for themselves.

| Powerful idea — *Divide et impera* This is one of the commonly cited features of so-called “computational thinking”. There is a lively debate about the exact definition, but to tell the truth, some of its aspects have always been good habits of a good scientist. This is the case of “breaking the problem into smaller parts”. I’m quoting it here because of how the problem can arise when teachers are “doing coding” with devices like the Bee-Bot or similar. Sometimes, when planning the coded path with text or cards, they interpret turn commands as steps, thus confusing two different actions: turning should not involve movement, and stepping should not involve turning. This will become clearer when we comment on the powerful idea of “state” later. Here it’s worth pointing out the importance of breaking a problem or process into smaller, clearly defined parts. |

The reason I insist so much on what kids can do is that I’ve had several opportunities to play turtle with them. Very few say it’s too hard, most just play and get excited about letting the little animal do funny things. Too often we forget how kids crave hard. Once a girl tried to draw a house and came up with a bizarre castle. I praised her and told her how wonderful the drawing was, and she was so proud. Then, when I returned to her after looking at the work of other children, I found her in tears, sobbing desperately: she had accidentally erased the instructions in a way that I could not recover. I felt guilty that I had not taught her how to save her work.

By the way, when you start creating some more complex stuff that you like, don’t forget to save the document once in a while. It’s that easy.

**Getting started with geometric shapes**

Most likely someone will come out and say:

— Let’s make a square!
or
— Let’s make a triangle...

or something else. Just go and follow their suggestions, even though we are going to show the example of a square here. If no one has an idea, you can help:
— Why don’t we try to make a...?

Let’s continue with the square. Often children can do this easily and end up with something like this:

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

Listing 3.3: The square, first version

The first of your children to achieve this result will be happy, of course, but is the task really over? At first glance, yes, since the square is there, but the point is more intriguing than that. In order to work properly, that is, to use the principle of dividing a task into smaller parts, the turtle should always end up in the initial state after drawing a geometric figure. What we mean is that if we call the drawing of a square twice, the turtle should draw the same square, superimposing the second on the first. Instead, in this case, see what happens when we try to repeat the same code twice:
Listing 3.4: Two not quite overlapping squares...

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 think about the turtle’s landing position in order to predict where the next square will be drawn. These considerations lead us to define the important concept of state of a system. If a process — for example, for drawing a square — has the property of leaving the turtle in the initial state, it is said to be state-transparent.

These considerations lead us to define the important concept of state of a system. If a process — for example, drawing a square — has the property of leaving the turtle in its initial state, it is said to be state-transparent. So our code is not state-transparent.

**Powerful idea — State of a system** It is appropriate to focus on the concept of “state of the turtle” because it is not enough to know the position: we also need to know where the turtle is looking. That is, to know everything about the turtle, we need to know both its position and
its 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 use in a practical way the concept of “state” of a system, which is a crucial concept in science. For example, in classical mechanics, the state of a system is given by knowing the spatial coordinates and velocity components of all its particles. In the world of turtle geometry (which we will talk about later), the state of the turtle is given by its two spatial coordinates and the directional angle at which it is pointing.

Returning to our square, it is easy to see that what is missing is a last turn to the right (instructions N. 10 and N. 19):

```
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: Two overlapping squares drawn with the final state of the turtle equal to the initial one
That is, each time we repeat a block of instructions such as 1-10, the turtle draws another square, perfectly superimposed on the previous one.

**And why not a triangle?**

This is a typical goal, often aimed at building the roof of a house on top of the previous square. Usually, both children and adults are quite 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 — quite easily — and, of course, the angle, to obtain an equilateral triangle. Often, regardless of age, people decide to replace 90° with 60°, because the interior angles of an equilateral triangle are equal to 60 degrees. So what’s the result?

Try to think about this before moving on to the next page. Perhaps you could use this page to make a pencil drawing...
CHAPTER 3. BASIC TURTLE PROGRAMMING

```
1  CLEARSCREEN  
2       HOME  
3  FORWARD 100  
4  RIGHT  60    
5  FORWARD 100  
6  RIGHT  60    
7  FORWARD 100  
8  RIGHT  60    
```

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

Quite disappointing, isn’t it?²

Reflection — Scientific knowledge The process of building scientific knowledge is iterative: first imagine a theoretical description, then try to devise an experimental confirmation, and finally go back to refine the theory if necessary. The triangle example is an example of how the process is naturally activated with Logo. In this case, we know the theory that says that the interior angles of an equilateral triangle are equal to 60 degrees. This is correct, of course. What’s wrong is how we’ve applied this knowledge: the angle we specify in the RIGHT statement is the deviation angle from the turtle’s current direction, not the interior angle of the figure we’re trying to build. The angle of deviation is supplementary to the internal angle we want to create, which is $180^\circ - 60^\circ = 120^\circ$. Once they see the disappointing result,

²This error is so common that even Abelson and diSessa quoted it in their Turtle Geometry [1, pag. 7]
people immediately realize the error and go to write the correct code.

Making things easier...

Repetitions

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

```
1  CLEARSCREEN
2  HOME
3  REPEAT 4 [
4     FORWARD 100
5     RIGHT 90
6  ]
```

Listing 3.7: Execute a sequence of instructions multiple times with REPEAT
For example, if the turtle finds a REPEAT 100 command, it will execute all the commands listed between the two square brackets “[” and “]” one hundred times. Instead of typing the same sequence of commands a hundred times, it is sufficient to enclose them in 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" them or, if you prefer, while "listening" to them. However, the turtle not only executes the commands one after the other, like an automaton, but it is also capable of memorizing 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: Define the "SQ" new command

When the turtle finds a command that starts with the word TO followed by a name, it stops executing successive commands as usual. Instead, it "sits and listens to the commands", learning them one by one without doing anything else, until it encounters the END command. What happens when you try to run this code in LibreLogo? Nothing happens! Actually, if we think about it, there is nothing to execute in this code, except for the first
two commands, CLEARSCREEN and HOME.

The following code shows how to take advantage of the turtle’s ability.

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

Listing 3.9: Define and apply the new ”SQ” command

Command number 1 has 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 when the turtle finds this new command, it will remember all the commands it has learned between TO and END and execute them in order. That’s why we put the SQ command at the end of the code. More specifically, between commands 1 and 6, the turtle sits and listens - nothing happens. Starting with instruction 7, the turtle executes all commands, one after the other.

Since we put the code on page 73 between the TO and END commands, we got a square. Probably sooner or later someone will try to define other new commands, like the triangle.
Reflection — Encapsulating functionality in new commands: modular thinking

Once students grasp the meaning of the TO...END construct, they feel a sense of empowerment. First, because you can encapsulate even a very complex sequence of instructions in a single command, and second, because you can extend and customize the Logo command set indefinitely. This perception easily leads to a modular way of thinking, which means trying to simplify a large process into a limited number of blocks that can be easily combined and possibly even reused in other contexts. This is an extremely useful step toward a scientific way of thinking that can simplify more complex tasks, reduce the incidence of errors, and improve communication.

As a result of this achievement, the level of self-determined goals is raised, for example, drawing a new figure by combining previously made pieces. A typical example is a house composed of a square and
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 on top. Very often people write a piece of code like this...

```plaintext
1 TO SQ
2   REPEAT 4 [
3       FORWARD 100
4       RIGHT 90
5   ]
6 END
7 TO TR
8   REPEAT 3 [
9       FORWARD 100
10      RIGHT 120
11   ]
12 END
13 CLEARSCREEN
14 HOME
15 SQ
16 TR
```

Listing 3.11: The “usually wrong” first house...

But this is not what was intended. You realize that some more thinking is needed and that something has to be done between the two new commands, SQ and TR. To solve the problem, you have to put yourself in the turtle’s
shoes: where is it and in which direction is it pointing after the square has been drawn? And from this state, which is the first move to make when tackling the triangle? Thinking about these questions, it is clear that before drawing the triangle, the turtle should move to the upper left corner of the square, using a FORWARD 100 instruction. This will start the drawing of the triangular roof at the top of the square, which is good. However, this only produces a translation of the triangle and our house will look as if it were opened like a can. For this reason, before we start drawing the roof, we should rotate the turtle to the right by an angle of... well, you should discover it for 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 CLEARSCREEN
16 HOME
17
18 SQ
19
20 FORWARD 100
21 RIGHT 30
22
23 TR
```

Listing 3.12: The correct first house...

In instruction 18 the turtle draws the square, then with 20 and 21 it reaches the correct state (position and orientation) to start drawing the triangle with instruction 23.

**Reflection — Again on modular thinking while drawing the house**

When you find out that 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 functionality and combine them freely. Once the house is there, there is a spontaneous desire to go one step further: perhaps we could put it all in a new command, HOUSE, for example. Try to consider the following code, which draws the same house:

```
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 TO HOUSE
16   SQ
17   FORWARD 100
18   RIGHT 30
19   TR
20 END
21
22 CLEARSCREEN
23 HOME
24
25 HOUSE
```

Listing 3.13: The “HOUSE” command

The names of new commands are emphasized: SQ, TR and HOUSE. Now we can send the turtle around and spread houses everywhere!
3.1 Making things smaller and larger

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:

Listing 3.14: “SIDE” is a parameter that has value 100

We used a new name here, SIDE, and assigned a number to it. This way the turtle knows that every time it encounters the name SIDE, it must use the number 100 instead. This is very useful if you need to use the number
100 many times in different statements. For example, going back to our first way of drawing a square, we can write the code like this

```logo
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

So if we want to make a smaller square, all we have to do is change the value of SIDE.

**Reflection — The door to algebra** As is well known, algebra is based on symbolic representation of numbers. Algebra is about generalization. Using symbolic names instead of numbers in Logo creates the mental tool for manipulating symbolic representations of numbers. This is a crucial step toward mathematical reasoning. We have used the term “symbolic names” so far, in the following we will call them variables or parameters, depending on the context.
The Turtle is even smarter: passing parameters to new commands

We have seen how to create new commands, such as the HOUSE command. Let’s say we want to draw houses of different sizes. Here is how to do it:
CHAPTER 3. BASIC TURTLE PROGRAMMING

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

The possibilities are endless. Look what a student of the Degree in Primary Education was able to do by exploiting the use of parameters in new commands:
The possibilities are even greater when you consider that as many parameters as we want can be passed to a new command. Try playing 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 3.17: A new command with two parameters
3.2 Doing round things

Syntonic learning

Sooner or later, someone will come along and say: — Why not make something round, like a circle? Remember the figures 3.4, 3.1, 3.2, and 3.3 on page 61? The playful exercises shown in these figures were followed by lively discussions about making the turtle produce the same geometric shapes. Maestra Antonella recounted some of these discussions among her eight-year-old children:

...  
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 turn!  
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 refine the concept of 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 — Oh that’s really easy: the right number is 360.  
Boy 1 — Mmh... no wonder... you’re a grown-up!

3That day, some older girls were placed in Antonella’s classroom because their teacher was temporarily absent.
Reflection — Syntonic learning described through a dialogue The story of Antonella, shown here through her photographs and the dialogues between her children, is a remarkable example of Papert’s pedagogical ideas: activating syntonic learning through physical activities and promoting self-determined goals, freedom of exploration. It is all about discovering the rules to achieve a personally meaningful goal, rather than teaching rules to solve problems that make no sense to the learner.

Now let’s rework the figure 3.2 in this way:

Figure 3.5: A small step and turn a little bit, a small step and turn a little bit...
This is a kind of “syntonic image” in that it associates 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 take a small step to stay on the yellow ring. At the same time, for the same reason, she has to turn her front feet a little. This helps her to imagine that the turtle should receive commands with small numbers, FORWARD 1 and LEFT 1, to make it draw something circular.

**Powerful mathematical idea — Differential equations** “Powerful ideas” is a typical expression of Seymour Papert, along with the concept of learning environments as “incubators of powerful ideas”. By this we mean ideas that are pre-existent, not part of the learning environment, but activated by it. In the light of today’s neuroscientific knowledge about learning, we could define the concept even more precisely: the activation of a powerful idea by an appropriate environment results in the creation of a neuronal configuration, a kind of first path in the bush, which can host a new mental device in the future. The powerful idea is now activated in an unconscious way, but in the future it may blossom into a thoroughly definite concept. This is exactly the case with the syntonic experiences of drawing circles just described. The key point here is that in the FORWARD 1 LEFT 1 code there’s no reference to any global attributes that characterize the concept of a circle: no mention of a center, no mention of a radius. The turtle is just given 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. Here is the differential equation of the circle:

\[
\left\| \frac{dT}{dS} \right\| = k,
\]

where \( k \) is called the *curvature* of the circle and is given by \( k = 1/r \), where \( r \) is the radius. From a mathematical point of view, the equation
3.1 is a *differential-equation* because it’s about variations, relating them to other quantities. In our case, speaking in intuitive terms, the 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 the curvature $k$, which is constant in the case of a circle. The following diagram shows the “small spatial displacement”, $d\vec{S}$, and the “small change of direction”, $d\vec{T}$.

![Figure 3.6: “Syntonic math”](image)

The important concept is that in a circle, the change in direction is constant as you move along the trajectory. This constancy is expressed by the constant value of the curvature $k$. In the following picture, the relevant elements of the diagram are superimposed on the girl’s step. Here, the constancy of the curvature is maintained by the girl’s effort to take regular steps and to bend the front foot regularly.
Of course, our girl is not supposed to know anything about differential equations and vectors\(^a\). Through this activity, however, she has discovered a new way of looking at a geometric figure, a way that lends itself to the concept of a differential equation. A perspective in which a figure (eventually a physical phenomenon) is constructed by means of a precise behavior that is “purely local”, without the need for 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 in a spiral curriculum of successive explorations of the circle. Let’s explore some of the successive steps that lead to other powerful mathematical ideas.

\(^a\) In exact mathematical terms, \(\vec{T}\) and \(\vec{S}\) represent vectorial quantities with direction and intensity. \(\frac{dT}{dS}\) 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
3.2. **DOING ROUND THINGS**

finite quantities *per se*. They can be used in expressions involving the concept of *limit*, the basic tool of mathematical analysis for embedding the infinite or the infinitesimal in finite quantities. For example, the so-called derivative $\frac{dT}{dS}$ is the result of a *limit* operation where the imaginary very small step, *at limit*, approaches 0. Thus, we obtain a finite quantity expressing the local rate of change of one variable, $T$, with respect to another, $S$. Later, when we talk about successive approximations, the same kind of concepts will be evoked.
Spira mirabilis

Figure 3.8: Jacob Bernoulli’s headstone, in Basler Münster.
This is the tombstone of Jakob Bernoulli (1654-1705), which can be seen in the cathedral of Basel. Jakob Bernoulli studied the spiral at length. He was fascinated by its mathematical properties, which he described in a treatise entitled *Spira Mirabilis*. He was so impressed that he had the spiral carved on his tombstone with the inscription *Eadem Mutata Resurgo*: Though changed, I keep arising the same.

Figure 3.9: The wrong spiral engraved in the tombstone: Archimede’s instead Bernoulli’s one...
CHAPTER 3. BASIC TURTLE PROGRAMMING

Listing (3.18)
SPIRA: Archimedean
SPIRB: Bernoulli

Unfortunately, the sculptor carved the Archimedes spiral, whereas it was the logarithmic spiral, not the Archimedes spiral, that made Bernoulli wonder. The difference is crucial because the two spirals grow in fundamentally different ways. Let’s go back to figure 3.5 on page 87: By taking a small step and turning a little regularly, we approximate a circle. But what happens if, for example, we reduce the step size? Well, we get a kind of spiral, the shape of which depends on how we reduce the step size. These effects are easy to explore in Logo.

Both programs, SPIRA and SPIRB, differ from a circle in that they have an instruction (No. 5 in SPIRA and No. 13 in SPIRB) in which the STEP parameter is increased by a certain amount at each step. The parameter
3.2. DOING ROUND THINGS

values are adjusted to show the fundamental difference between the two spirals. The two programs differ in instructions 5 and 13, in which the STEP parameter is increased, but in SPIRA (Archimedean) it is increased by a fixed amount of 0.075, while in SPIRB (Bernoulli) it is increased by \( \text{STEP} \times 0.035 \), that is, by an amount proportional to \( \text{STEP} \). This means that in the first case the loops grow by a constant amount, while in the second case the loops get larger and larger with each turn. The Bernoulli spiral is called logarithmic and can be found in a variety of natural forms. Among them are ammonite fossils.

<table>
<thead>
<tr>
<th>Listing (3.19) SPIRB with parameter adjusted to approximate the ammonite curvature.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 TO SPIRB STEP ANGLE</td>
</tr>
<tr>
<td>2 REPEAT 300 [</td>
</tr>
<tr>
<td>3 FORWARD STEP</td>
</tr>
<tr>
<td>4 LEFT ANGLE</td>
</tr>
<tr>
<td>5 ( \text{STEP} = \text{STEP} + \text{STEP} \times 0.005 )</td>
</tr>
<tr>
<td>6 ]</td>
</tr>
<tr>
<td>7 END</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9 RIGHT -90</td>
</tr>
<tr>
<td>10 SPIRB 1 2.5</td>
</tr>
</tbody>
</table>

(a) Bernoulli’s spiral superimposed to an Ammonite fossil.

---

**Powerful mathematical idea — Linear and exponential growth** By playing with these spirals students get in touch with the essence of linear and exponential variation. The equation of the Archimedean
spiral, expressed in polar coordinates, is

\[ r = a + b\theta \]  \hspace{1cm} (3.2)

This means that the radius \( r \) grows proportionally to the angle \( \theta \). The equation for Bernoulli’s logarithmic spiral is:

\[ r = ae^{k\theta} \]  \hspace{1cm} (3.3)

In this case \( r \) grows exponentially with the angle \( \theta \). How much of these explanations should be given to students depends on which point in the spiral (!) curriculum they are at. The important thing is to introduce the linear-exponential dichotomy early.

---

**Powerful mathematical idea — Self-similarity ➔ fractals** One of the mathematical properties of the logarithmic spiral that fascinated Bernoulli is its self-similarity. Let’s add a shift \( \theta_s \) to the angle \( \theta \), that is, rotate the spiral by an angle \( \theta_s \).

\[ r = ae^{k(\theta+\theta_s)} = ae^{k\theta}e^{k\theta_s} \]  \hspace{1cm} (3.4)

This means that by rotating the spiral we only change the scale, by a factor \( e^{k\theta_s} \), while the shape remains exactly the same, i.e. \( e^{k\theta} \).

Conversely, if we multiply the spiral by a scale factor, \( S \), we get

\[ r = Sa e^{k\theta} = ae^{k\theta+\ln S} = ae^{k\left(\theta + \frac{\ln S}{k}\right)} \]  \hspace{1cm} (3.5)

we get 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.

---

**Golden spiral**

The Bernoulli logarithmic spiral is related to the so-called *golden spiral*. More precisely, the *golden spiral* is a special type of logarithmic spiral,
where the growth rate of the spiral depends on the golden ratio, one of the magic numbers of mathematics, ubiquitous in nature. Its value can be calculated with $\phi = \frac{1 + \sqrt{5}}{2}$. Its approximate value is 1.618\(^4\). There is a nice approximation of the golden spiral that can be constructed with paper, pencil, compass and scissors—it’s good to mix technologies. You can proceed as follows:

1. Cut a square of paper, e.g. with a side of 10 cm
2. Draw a quarter of circle inscribed in the square, from one corner to its opposite
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 next to the previous one so that the first quarter of the circle continues into the second without interruption.
6. Repeat steps 3-5 until you can manage the progressively smaller squares

You will get something like that:

\[ \frac{1 + \sqrt{5}}{2} \]

\(^4\)The golden ratio is an irrational number, so it can’t be represented exactly by a finite number of digits.
This is a good approximation of a golden spiral, which in turn is a logarithmic spiral, or a spira mirabilis, in Bernoulli’s words. This figure was created with Logo. As usual, the result can be achieved in many different ways. You can try yourself to reproduce the physical construction of the spiral with Logo.
POLY

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

```
1 TO TR SIZE
2    REPEAT 3 [ 
3       FORWARD SIZE
4       RIGHT 120
5    ]
6 END
7
8 TO SQ SIZE
9    REPEAT 4 [ 
10       FORWARD SIZE
11       RIGHT 90
12    ]
13 END
```

Listing 3.20: 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 they have in common? The differences are:

- the command names, TR and SQ
- the number of repetitions, 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 like mathematicians, by making things as simple as possible in order to squeeze out the essence:
CHAPTER 3. BASIC TURTLE PROGRAMMING

Listing 3.21: The POLY command

```
1 TO POLY SIZE ANGLE
2   REPEAT [
3       FORWARD SIZE
4       RIGHT ANGLE
5   ]
6 END
```

The program POLY is an important object of turtle geometry. By playing around with POLY, you may discover some nice and relevant facts about turtle geometry, geometry in general, and computers. But more importantly, you will learn to explore, formulate questions, set goals, and achieve them on your own.

For example, when you run the following code...

```
1 TO POLY SIZE ANGLE
2   REPEAT [
3       FORWARD SIZE
4       RIGHT ANGLE
5   ]
6 END
7
8 POLY 100 120
```

Listing 3.22: Running the POLY command

the turtle is going to draw a triangle, which is not a big news since we already learned that by drawing sides with 120° rotations you get an equilateral triangle. The first observation is that once the triangle is drawn, the turtle keeps drawing it, over and over again. We’ll come back to this later. For now, to stop the turtle, click on the red STOP button in the LibreLogo toolbar⁵. Instead, let’s just play, taking advantage of the super concise writing, where

⁵Similarly, you can stop the process in TigerY’thon by clicking on the red STOP button at the top left of the window.
the POLY code is reduced to its essential pattern. By playing with different values of the SIZE and ANGLE parameters, think of POLY as a kind of magic box for creating new objects. Try to think about the different effects you get by changing SIZE or ANGLE. Which is more interesting? What kind of shapes do you get? Just polygons? Or something else? Those who wish to experiment can stop here and try it out for themselves before proceeding to the next page.
CHAPTER 3. BASIC TURTLE PROGRAMMING

Listing (3.23)
The SIZE parameter is just a scale factor

```
1 TO POLY SIZE ANGLE
2    REPEAT [
3        FORWARD SIZE
4        RIGHT ANGLE
5    ]
6 END
7
8 POLY SIZE ANGLE
```

This is just an example, we invite the reader to experiment. It is fun to explore the variety of shapes and how they grow. You can also tinker with the contents of POLY by inserting other instructions to see what happens. Like this, for example:
3.2. DOING ROUND THINGS

1 TO POLY SIDE ANGLE
2   REPEAT [
3     FORWARD SIDE/3
4     RIGHT 60
5     FORWARD SIDE/3
6     LEFT 120
7     FORWARD SIDE/3
8     RIGHT 60
9     FORWARD SIDE/3
10    RIGHT ANGLE
11  ]
12 END
13 POLY 100 144

Listing (3.24) An example of POLY variation, with ANGLE = 144

In all of these cases, it is interesting to note that some figures require only a few cycles to complete, while others require many more iterations. The question naturally arises: how many cycles in POLY are needed to “close” the figure, for a given value of ANGLE? You may have already noticed that for certain figures the answer is quite easy to find. Basically, in the previous examples we found two kinds of objects: regular polygons and some kind of stars. With 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 are all 360, we can conclude that to draw a polygon with N sides, the turtle must make 360 / N turns. This is an empirical rule, which means that we cannot be sure that it is universally true. However, the Total Turtle Trip Law is true [2, p. 76:]

**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 have used the word “law” because it reminds us of something that is always true under all conditions, like the laws we must obey in our everyday lives. In mathematics we have axioms, or postulates, on which whole theories are based, and theorems, statements that are proven to be true from axioms. In physics we have principles, of energy conservation, of dynamics, and so on. In turtle geometry there are some theorems that are true: the Total Turtle Trip Law is the first.

**Powerful idea: Concept of “law”** By playing with the constancy of a result, independent of the path, we develop 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 of giving structure to the narrative of science. With older children, you can try to build a strange figure and see if the law is still followed.

For example, here we first draw a square, accumulating the deviation angles in the variable T, and printing the final value at the end, which of course turns out to be 360. Then a new figure is drawn, which is the same square but with strongly distorted sides, again 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.

```plaintext
1 CLEARSCREEN
2 HOME
3 HIDETURTLE
4
5 T =0
6 REPEAT 4 [ 
7 FD 100
8 RT 90
9 T = T +90 
10 ]
11
12 PRINT T
13
14 CLEARSCREEN
```
3.2. *DOING ROUND THINGS*

Listing 3.25: A distorted square, to show the invariance of total turn

```plaintext
15 HOME
16 PENUP
17 LEFT 90
18 FORWARD 100
19 PENDOWN
20 RIGHT 90
21
22 T = 0
23 REPEAT 4 [
24   REPEAT 90 [
25     FORWARD 1
26     LEFT 1
27     T = T - 1
28   ]
29   REPEAT 180 [
30     FORWARD 1
31     RIGHT 1
32     T = T + 1
33   ]
34   REPEAT 90 [
35     FORWARD 1
36     LEFT 1
37     T = T - 1
38   ]
39   RIGHT 90
40   T = T + 90
41 ]
42
43 PRINT T
```

Listing 3.25: A distorted square, to show the invariance of total turn
Powerful idea: Concept of integration In the previous example, we accumulated a quantity along a path by summing up all the deviation angles as the turtle traveled. This is the basis of a fundamental operation of mathematical analysis: integration. In the previous example, we proposed a simple code using, for example, a command like $T = T \pm 90$ to update the current value stored in the variable $T$. However, with children who are not yet sufficiently familiar with programs, this work can be done in parallel, by reading the code and at the same time noting the values with paper and pencil and summing them by hand.

Of course, we do not have to talk about integration explicitly with the students — that would be nonsense. It is the process, the direct experience that counts. The possibility of creating a new mental tool that will one day help to accommodate the formalization of the concept.

More examples of integration along the pathways of the turtle can be found in the companion website.

Reflection — The limits of the machine (and of theory): How can the execution of a program unintentionally become a never-ending
3.2. **DOING ROUND THINGS**

story? This is the title of a section (2.4) of Juraj Hromkovič’s book, “Algorithmic adventures” [3, p. 62]. It is perfect here. It may seem strange: according to common sense, computers are associated with deterministic computing. Sort of complicated technology, but clearly defined and predictable. In reality, they are complex and therefore tricky machines, both from a practical and theoretical point of view. Now, we are already aware of the absence of a halting condition in our POLY program, this is not a bug. It is a known imperfection, but imperfections (and errors) are useful for learning. The POLY “defect” of not stopping itself may turn out to be an opportunity to think about machine behavior. For example, what does it mean when the computer (the turtle) does nothing? Or, what’s going on when it seems to get stuck?

When we ran 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, from the time when the turtle starts walking over lines that have already been drawn. But try to hide the turtle before starting the POLY by using the HIDETURTLE command. At the beginning, the invisible turtle draws the figures and you see it clearly, but once the figures are finished, there is no way to see what the turtle is doing because it is invisible! You know it’s there because you used the HIDETURTLE command, so you know what’s going on, but imagine you didn’t. What would you conclude? That the task is finished? Not exactly. At first glance, yes, but after a while you will notice that something is wrong. For example, if you try to run the code again, you will not be able to, and the reason is that the invisible turtle is still busy. If you want to start a new run, you have to stop the current one first. This is exactly what we did by playing with POLY. But what is worth noting here is that once the figure is finished the system seems to be stuck. So what we’re learning is that when we say the computer is stuck, it’s actually working, we just don’t see it. In our case, the turtle continues to do something useless, we now see it perfectly. But
it is important to know that when running a program, false conditions that cause a similar condition are quite common, even in very famous (and expensive) software.

This is not to say that software designers are incapable, they are generally very good. Rather, there are serious reasons for this state of affairs. A first problem is that software systems today are extremely complex. It’s often impossible to anticipate all potential problems.

But the reality is even harsher: there is no way to build automatic testing methods (algorithms, technically speaking) to evaluate the correctness of a program. This is a strong statement, in the mathematical sense, meaning that it has been shown theoretically that there is no way to answer questions like [3, 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 are not computer scientists or software engineers, and few of our students will pursue those careers. We don’t have the opportunity to explore this further, although it would be very interesting. We have raised the issue here to encourage a thoughtful and critical approach to technology. An attitude that is aware of the limitations and the inevitable risks of failure or misbehavior. A more vigilant attitude toward technology is crucial in education today if we are to create conscious citizens rather than passive consumers.

---

For the curious — Trying to find a stop condition for a (simple) program It is probably clear by now that if ANGLE is a submultiple of 360, the figure is a regular polygon with \(N=360/\text{ANGLE}\) sides. Therefore, in these cases, the number of repetitions of POLY needed

---

In TigerJython use the hideTurtle() command
to close the figure is $N$. But in all other cases? When $\text{ANGLE}$ is not a submultiple of 360 or $\text{ANGLE} > 360$?

Some years ago, a student in the Elementary Education program sent me a letter — it was not an assignment — in which, starting with a description of an exploration with Logo, written “as if I were a child”, she ended with a mathematical conjecture: is there a way to determine \textit{a priori} how many iterations are needed to close these figures? What she was really exploring was the behavior of a kind of POLY program. More specifically, she was playing with a POLY structure with the construction of a house inside it!

The solution to this problem comes from turtle geometry. It is based on several theorems, the most important of which is a generalized version of the Turtle Total Trip Theorem we saw earlier. It is the \textit{Closed-Path Theorem} [1, pag. 24]:

\textbf{Theorem 1} The total turning along any closed path is an integer multiple of $360^\circ$.

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 multiplied by 360. This integer is called the \textit{rotation number} of the path. It is interesting to try to evaluate the rotation numbers for different paths. Here are some examples:
The general result is that the number of cycles needed in POLY to complete a figure is given by

\[ n = \frac{\text{LCM}(\text{ANGLE}, 360)}{\text{ANGLE}} \]  

(3.6)

where ANGLE is the input angle to POLY and LCM(ANGLE, 360) is the least common multiple between ANGLE and 360.
So, this was for the curious. But some of the readers may find material here to suggest some interesting explorations for slightly older kids.

**Examples of syntonic learning activities** The exact derivation of equation 3.6 and the answer to Marta’s conjecture can be found in [1, pp. 24-32] and [1, pp. 32-36], respectively; synthetic derivations are available in the companion website.

**Successive approximations**

We have seen that with POLY we can draw regular polygons and “stars”. Let’s write a version of POLY to draw only polygons. It’s easy at this point:

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

Listing 3.26: A new version of POLY with SIDE and N, number of sides, as input parameters

This is a kind of “educated” POLY that knows how to stop. This version is limited to regular polygons\(^6\), but this is exactly what we want right now. In this version, we have changed one input parameter: instead of ANGLE, we put N, the number of sides. Then, in the body of the program, we calculate the turning angle as $360/N$.

Now we can play with POLYG and focus on regular polygons. The game is simple: try to draw polygons with an increasing number of sides. Well,

---

\(^6\)It would be possible to write an “educated” general POLY program, but we would have to use the result given in the equation 3.6, so we should write the code to calculate the least common multiple.
from time to time you will have to readjust SIDE, otherwise the figure will exceed the page with a large number of sides. What will you see from a certain point?

For example, let’s try to superimpose the decagon (10 sides) with the icosagon (20 sides):

![Figure 3.16: 10-sided and 20-sided polygons](image)

See how the icosagon is “rounder” than the decagon? Now let’s compare the icosagon (20 faces) with the triacontagon (30 faces).
3.2. *DOING ROUND THINGS*

Here we see that the 30 faces polygon is again “rounder” than the 20 faces polygon. Now let’s exaggerate and compare the 30-sides polygon with the 360-sides polygon — 360 1° turns!

Figure 3.17: 20-sided and 30-sided polygons
The differences are minimal, and even when you zoom in deeply, the 360-sided polygon appears “round”. This is because the length of the sides is comparable to the size of the pixels, so we can no longer see them. Therefore, in Logo we can assume that a circle can be drawn using a command like \texttt{REPEAT [FORWARD 1 RIGHT 1]}. Recently, a bright 16-year-old student said to me: “Professor, you are claiming that we are drawing a circle this way, but it cannot be, it is still a polygon, even with many very small sides! She was absolutely right. The discussion that followed was extremely fruitful, and led to a crucial consideration: we can choose an extremely high number $N$ to build an $N$-sided polygon, and it will be “rounder”, even with respect to our last 360-sided polygon, but it will still not be a true circle. No one is stopping us from choosing an even larger number to draw an even rounder polygon, but still a polygon! So can we get as close to a circle as we want? Yes, we can! Will we be able to reach the circle? No, unless we have eternity! Or the mathematical concept of limit.
Powerful mathematical idea — Successive approximations Such considerations are at the root of the reasoning behind the treatment of infinity and infinitesimality in mathematical analysis. Limits, derivatives, integrals, approximation by infinite series, and other mathematical devices allow us to solve a wide variety of problems in science by dealing with the concept of infinity. 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. This, by the way, is one of the possible definitions of a circle, once put in rigorous formal terms.

The idea of a thought process composed of perfectly defined steps but “requiring eternity” falls within the realm of the considerations made in the note a on page 90. It is by means of the limit tool of mathematical analysis that infinity and infinitesimal can be “packaged” into 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 [3, pag. 73].
Bibliography


4.1 Free-falling body

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

Let’s consider a free-falling body near the surface of the Earth. When you were introduced to these topics, you were probably told that there is a formula, probably you were told that there is a formula describing the motion of the falling body, which can be written as

\[ y(t) = y_0 + v_0 t + \frac{1}{2} g t^2 \]  

(4.1)
where $y$ is the position of the body at each time $t$, $y_0$ is its initial position, i.e. 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 gravitational acceleration constant near the Earth’s surface, which is $9.8 \, \text{m/s}^2$. Depending on the curriculum, the teacher’s method and your age, you may be told that this formula is derived from the second principle of dynamics, which states that

$$F = ma$$

(4.2)

where $F$ is the force acting on the body (gravity in this example), $m$ is the mass of the body and $a$ is its acceleration\(^1\). To understand this derivation, you need to master the basic tools of mathematical analysis, in this case integration. You have probably memorised these formulae in order to do the exercises and pass the exams. Unfortunately, this is not enough for a thorough and deep understanding of the subject. Logo can help, let’s see how.

In the following figure, on the left, we show the relevant part of the Logo code for simulating the free fall of a body, and on the right, the output of the program, where the position of the body at successive equal time intervals is marked with a small green circle.

\(^1\)Later in your studies you’ll learn that the equation 4.2 is a second-order differential equation because it involves variations of variations: acceleration is the variation of velocity, while velocity is the variation of space. This is written as $\frac{d}{dt} \frac{dy}{dt} = mg$. By integrating this equation twice with respect to time, we obtain the equation 4.1.
4.1. FREE-FALLING BODY

There are a couple of new commands here. Firstly, we made a loop using a WHILE statement instead of a REPEAT statement. In particular, we used:

```logo
1 YPOS = 0.0 ; initial position
2 VEL = 0.0 ; initial velocity
3 ACC = 9.8 ; constant acceleration
4 DT = 0.5 ; time interval

6 WHILE YPOS < 500 [
7 VEL = VEL + ACC*DT ; update velocity for next point
8 YPOS = YPOS + VEL*DT ; update position for next point
9 PENDOWN
10 CIRCLE 5
11 PENUP
12 FORWARD VEL*DT
13 ]
```

Listing 4.1: Logo code for a free falling body experiment
When using a REPEAT, we need to specify a number of repetitions, for example REPEAT 10. Here we have the expression YPOS < 400 instead of the number of repetitions. What does this mean? The expression is a so-called condition: the WHILE command will repeat the sequence of statements between the square brackets, [...], until the condition is satisfied, i.e. until the value of the variable YPOS is less than 400. The first time YPOS is greater than or equal to 400, the turtle will come out of the loop and execute the following instructions. The second new command is CIRCLE. This is a ready-to-use LibreLogo command: CIRCLE R, where R is the radius of the circle. We only used it for brevity. In LibreLogo there are such ready-made commands for the most common figures. We have not used them yet, because for the first basic explorations of turtle geometry it is better to do things “by hand”. Nobody is 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 straight to the physics concepts, but before we discuss them, let us switch to Python, in the TigerJython environment.
4.1. FREE-FALLING BODY

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

```python
from gturtle import *
makeTurtle()
clearScreen()

setPos(0,200)
setHeading(180)

YPOS = 0.0  # initial position
VEL = 0.0   # initial velocity
ACC = 9.8   # constant acceleration
DT = 0.5    # time interval

while YPOS < 400:
    VEL = VEL + ACC*DT  # update velocity for next step
    YPOS = YPOS + VEL*DT  # update position for next step
    penDown()
dot(5)
penUp()
forward(VEL*DT)
```

Let's comment this program in detail. Command N. 1 imports all the resources that Python needs to manage a turtle. It’s a common practice in Python to load only the necessary resources. Instruction n. 3, makeTurtle(), creates a so-called turtle instance, essentially a new turtle that listens to your commands. Instructions Nos. 6-7 tell the turtle at the top of the
page to drop itself 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 in solving any physics problem. In the algebraic formulation of the problem, namely the equation 4.1, the initial conditions consist of the values of the initial position $y_0$ and the initial velocity $v_0$. You will (perhaps) realise the importance of these values when you have to solve some problems, but here it is more direct: you have to decide immediately where to place the turtle and what initial velocity to give it. You also have to decide which unit of measurement to use. In other words, you have to build up a thorough knowledge of the concept by using it immediately. Similarly, you will need to find the correct value for the acceleration of gravity in instruction N. 11.

**Powerful concept in physics — Initial conditions**

Setting initial conditions is the first essential step in solving any problem in physics. Initial conditions are embedded in formulas taught in secondary schools, but at this age the ability to understand all the implications that are synthesised in formulas is rare — building the mental apparatus for understanding mathematical formulas takes time, especially when physical meaning is involved. The problem is not trivial and not new: a famous physicist, Enrico Persico, master of Enrico Fermi, wrote an interesting paper in 1956 [1], wondering about the good formal preparation of some students but their poor understanding of the core issues:

Why does this girl, who is not stupid but finds it so difficult to describe a capacitor, run like a locomotive once she has started writing formulae [Maxwell’s equations]?  

The computational formulation of physical problems favours the dynamic perception of physical phenomena over the conventional algebraic formulation, as has been pointed out in a very thoughtful paper by Sherin [2]. It’s not about abandoning the algebraic description of physical phenomena. It’s about integrating both approaches.
4.1. *FREE-FALLING BODY*

In instruction No. 12 we set the *time interval*. This is a necessary parameter because continuous quantities become discrete when solving problems numerically. This means that when we use time, we have to divide it into many equal intervals and evaluate all the quantities — position, velocity... — once for each interval of time, for example every second. The length of the interval has to be chosen as a compromise: many short intervals describe better what’s going on, but the computational cost may be too high.

**Powerful concept in physics — Computational vs algebraic approach**

When faced with problems from a numerical point of view, students are forced to re-enter the anatomy of phenomena. The appreciation of the numerical face of problems, alongside the algebraic one, is essential for the proper formation of a student’s scientific culture. Today, the computational side of all sciences is as important as the classical descriptions. Moreover, completely new and crucial fields are entirely numerical. All technologies are based on numerical solutions to mathematical problems. For example, the ubiquitous computed tomography images represent the numerical solution of the tomographic problem based on the inversion of the Radon transform. The technologies are based on a fine blend of analytical formulations and numerical methods. It is important to give students the opportunity to immerse themselves in numerical exploration.

Instructions 14-20 solve the problem. What does it mean to “solve a problem” here? From a classical point of view, it means finding the function \( y(t) \) given the physical circumstances. The equation 4.1 is the solution of the problem 4.2, given the acceleration \( g \) and the initial conditions \( y_0 \) and \( v_0 \). In numerical terms, solving the problem means calculating the value of the position YPOS for a sufficient number of time points and, in our case, plotting it graphically. In the numerical way, the phenomenon is actually simulated. Instruction No. 14 starts a loop that continues until the position YPOS is less than 400. For each cycle, the velocity for the next time point is first evaluated with \( \text{VEL} = \text{VEL} + \text{ACC} \times \text{DT} \) (Instr. N. 15), based on the definition of acceleration. Once we have the new value of VEL, we can find
the new value of the position: \( \text{YPOS} = \text{YPOS} + \text{VEL} \times \text{DT} \), according to the definition of velocity (Instr. N. 16). Finally, \text{dot(5)}\ draws a point at the current position and \text{forward(VEL*DT)}\ sends the turtle to the next point. The figure shows the expected uniformly accelerated motion.

The reader is invited to play around with this code, trying to calculate more data points or changing the acceleration, for example, how 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 will show some variations on the previous case. We leave it to the reader to discover the differences. We will only quote the added effects.

Here we show the same free falling situation but with a constant horizontal velocity component. Three trajectories are shown for three different values of the gravitational acceleration constant. This physics exercise allows us to see how a mathematical object — the parabola — is derived from a natural phenomenon. It is interesting to explore the effect of the parameter \( a \) in the parabola equation \( ax^2 + bx + c = 0 \), which in this context is equal to the gravitational acceleration constant \( G \).
4.3. **FREE-FALLING BODY WITH AIR RESISTANCE**

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

```python
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)
```

4.3 **Free-falling body with air resistance**

Here we have a component of air resistance, RES, which we express as a force proportional to the speed VEL by means of the constant KA. The acceleration, or rather the deceleration, caused by this force is RES/M,
where $M$ is the mass of the body. For example, try to simulate a thicker atmosphere...
4.3. **FREE-FALLING BODY WITH AIR RESISTANCE**

Listing 4.4: Python code for a free falling experiment with air resistance. The steps become constant because of the balance between gravity and air resistance.

```python
\begin{programcode}

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)

\end{programcode}
```
4.4 Body hanging from spring

Python code for a free falling experiment with air resistance. The steps become constant because of the balance between gravity and air resistance. We keep the air resistance component proportional to the body’s speed, 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 strength of this force is given by the constant $K$. In this version we also add a horizontal displacement to obtain the effect of plotting the movement against time. In order to simulate only the motion, the last three commands can be set as comments by placing a `#` character at the beginning of the lines.
4.4. **BODY HANGING FROM SPRING**

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

```python
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()
donote(5)
pennUp()
forward(VEL * DT)

# set as comments the following instructions
# if want to see oscillating along y-axis
left(90)
forward(3*DT)
right(90)
```
CHAPTER 4. SIMULATION: PHYSICS DYNAMIC

4.5 Pendulum

A bit of trigonometry is needed here. There is some work to be done on this code, but it is a good starting point...
4.5. PENDULUM

```python
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

L = 500 # Pendulum length

# Where are we starting?

PHY = -pi/20 # initial angle

# With 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: Python code for a body hanging from a spring in the presence of air resistance.
4.6 Body free-falling from space

This is the same case as for the free-falling body, except that we have to drop the assumption of constant gravitational acceleration, because as we move away from the Earth the force of gravity decreases. This means that we have to use Newton’s law of gravity explicitly. Can you find out where in the code we use Newton's law?
4.6. BODY FREE-FALING FROM SPACE

Listing 4.7: Body free-falling from space.

```python
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)
```
CHAPTER 4. SIMULATION: PHYSICS DYNAMIC

4.7 Calculate the orbit of Halley’s Comet...

To tell the truth, I was reluctant to include this section for fear of frightening the reader. Actually, it is a little outside the scope of this text, but I think it could also be fascinating.

A note on the term "program", which we have used throughout the list. In all “Einfach Informatik” volumes published by ABZ, for example in “Einfach Informatik — Programmieren — Sekundarstufe I” [3], the word “command” (Befehl) is used to design sequences of instructions that can be called by a single name to be executed as a whole. This choice makes sense, because when you start with Logo it is very appropriate to give the children the idea of creating their own self-made commands, since with the turtle these encapsulated pieces of code are usually new commands for the turtle. However, in this case it is a bit strange to call the instructions used to do an interpolation, for example, “command”. Hromkovič and Tobias [4, pag. 45] raise the point about the variety of names used to call these objects, for example “routines”, “procedures”, “subroutines”, “methods”, “functions”. They do not mean exactly the same thing, depending on historical context and specific behaviour. For example, in Python jargon they are usually called “functions”. However, there is also a tendency to assume that “functions” calculate a value and render it. Well... in this list we have used the term “subroutines”.

But another reason for going through this example is to see how similar the structures of the simple algorithm we have seen for drawing a circle are to that for drawing the orbit of a celestial body, despite the remarkable difference in complexity. The reason for this similarity lies 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:
4.7. **CALCULATE THE ORBIT OF HALLEY’S COMET**

**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;  
3. 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;  
3. end

**Algorithm 2:** Calculating Halley’s comet orbit

---

The previous example of a body falling out of space opens the way to evaluating the orbits of bodies in space. The addition of an initial horizontal velocity component allows us to simulate orbits. Let us sketch the problem in the context of Newton’s gravity, first in algebraic notation and, of course, restricting ourselves to the two-dimensional problem. As far as the algebraic approach is concerned, here we just pose the problem. The analytical solution of the differential equations is a problem for higher education. However, the algebraic approach serves as a starting point for the programmed numerical solution.
To solve the problem we need to start from the second principle of dynamics

\[ \mathbf{F} = m \mathbf{a} \]  

(4.3)

where the force is given by Newton’s law

\[ \mathbf{F} = -\frac{GMm}{r^2} \frac{\mathbf{r}}{r}. \]  

(4.4)

\( G \) is the universal gravitational constant, \( M \) is the mass of the larger of the two bodies, \( m \) is the mass of the smaller, \( r \) is the distance between the two masses, between the barycentres of the two bodies to be precise. Symbols in bold are vectors, the others are scalars. First, we have to derive the acceleration of the smaller body — the larger one will remain stationary, assuming that its mass is negligible with respect to that of the Sun, i.e. \( M \gg m \).

\[ \mathbf{a} = -\frac{GM}{r^2} \frac{\mathbf{r}}{r}, \]  

(4.5)

and then one should solve the differential equation

\[ \frac{d}{dt} \frac{d\mathbf{r}}{dt} = -\frac{GM\mathbf{r}}{r^2 \cdot r}. \]  

(4.6)

Of course, this is beyond our aims here. However, we will use this relationship as a basis for the numerical approach. Let us look at the list.

```
1 # Halley-RK-4-AU-90-sharable.py
2 # Copyright 2020 Andreas Robert Formiconi (arf@unifi.it)
3 # Program distributed under the terms of the GNU
4 # General Public License
5 # This program is Free Software: it can be redistributed
6 # and modified under the terms of the GNU General Public
7 # License published by Free Software Foundation, in
8 # or one of the following ones. The text of the license
9 # is accessible in
10 #
11 #
```
4.7. **CALCULATE THE ORBIT OF HALLEY’S COMET...**

```python
# Calculation of the orbit of a celestial body around the sun
# by numerical integration of the equations of motion according
to Newton’s law of gravitation. The problem is posed in two
# dimensions and assumes that there are no perturbations from
# other bodies. The code is adapted to solve the case of a highly
# eccentric orbit, such as that of Halley’s comet.
# The turtle plays the role of the comet. The sun is in the
# center, which in TigerJython has coordinates (0,0), where
# the unit of measurement is the "point" (p).
# Warning: if you run this code as it is, be patient: the
# trajectory takes about a minute or so to appear, on the
# right,
# because a lot of points have to be calculated, to get a
# reasonable accuracy...

from gturtle import *

makeTurtle()
clearScreen()
showTurtle()

# Global variables, i.e., variables that are "visible" both
# in the main program and in each subroutine, such as
# NEWTON, STP and so on.

global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS,
# XVEL, YVEL, XACC, YACC

# The following are subroutines in which certain
# functions
# have been encapsulated: NEWTON calculates the
# acceleration
```
CHAPTER 4. SIMULATION: PHYSICS DYNAMIC

# at a given point, STP evaluates the next step, WRITEPOINT
# plots the point as long as they are calculated
# subroutine NEWTON: calculates the acceleration at a point of
# coordinates X, Y and returns two acceleration components
# ACX and ACY

def NEWTON(X,Y):
    global GG, Dt, DX, DY, XPOS0 , YPOS0 , XPOS , YPOS , XVEL , YVEL , XACC , YACC
    DX = (X-XPOS0)
    DY = (Y-YPOS0)
    R2 = (DX**2 + DY**2)
    R = sqrt(R2)
    XACC = - GG / R2 * DX / R
    YACC = - GG / R2 * DY / R

# Subroutine STP: calculates the next step using fourth order Runge-Kutta interpolation. This interpolation is a rather sophisticated way calculating the trajectory points.
# It returns the coordinates of the new position, XPOS and YPOS, and the velocity at that point, XVEL and YVEL.

def STP():
    global GG, Dt, DX, DY, XPOS0 , YPOS0 , XPOS , YPOS , XVEL , YVEL , XACC , YACC
    NEWTON(XPOS, YPOS)
4.7. CALCULATE THE ORBIT OF HALLEY’S COMET...

70 \( KX1 = Dt \times XACC \)
71 \( XVEL1 = XVEL + KX1 / 2. \)
72 \( KY1 = Dt \times YACC \)
73 \( YVEL1 = YVEL + KY1 / 2. \)
74 \( XPOST = XPOS + XVEL1 \times Dt / 2. \)
75 \( YPOST = YPOS + YVEL1 \times Dt / 2. \)
76
77 \textbf{NEWTON}(XPOST, YPOST)
78 \( KX2 = Dt \times XACC \)
79 \( XVEL2 = XVEL + KX2 \)
80 \( KY2 = Dt \times YACC \)
81 \( YVEL2 = YVEL + KY2 \)
82 \( XPOST = XPOS + XVEL2 \times Dt / 2. \)
83 \( YPOST = YPOS + YVEL2 \times Dt / 2. \)
84
85 \textbf{NEWTON}(XPOST, YPOST)
86 \( KX3 = Dt \times XACC \)
87 \( XVEL3 = XVEL + KX3 \)
88 \( KY3 = Dt \times YACC \)
89 \( YVEL3 = YVEL + KY3 \)
90 \( XPOST = XPOS + XVEL3 \times Dt / 2. \)
91 \( YPOST = YPOS + YVEL3 \times Dt / 2. \)
92
93 \textbf{NEWTON}(XPOST, YPOST)
94 \( KX4 = Dt \times XACC \)
95 \( XVEL4 = XVEL + KX4 \)
96 \( KY4 = Dt \times YACC \)
97 \( YVEL4 = YVEL + KY4 \)
98 \( XVEL = XVEL + (KX1 + 2 \times KX2 + 2 \times KX3 + KX4) / 6. \)
99 \( YVEL = YVEL + (KY1 + 2 \times KY2 + 2 \times KY3 + KY4) / 6. \)
100 \( XPOS = XPOS + (XVEL1 + 2 \times XVEL2 + 2 \times XVEL3 + XVEL4) \times Dt / 6. \)
101 \( YPOS = YPOS + (YVEL1 + 2 \times YVEL2 + 2 \times YVEL3 + YVEL4) \times Dt / 6. \)
102
103 # **************************************************
104 # subroutine WRITEPOINT. It does two things:
105 # 1) it sends the \texttt{turtle to} the next point of the trajectory
119 # with a POSITION command - that’s how the trajectory is  
120 # drawn.
121  
122 # 2) it writes to a file all the relevant values of each  
123 # point of the trajectory: two position, velocity and  
124 # acceleration components - this is the solution to the motion problem  
125 # This data can be used in turn to create graphs with  
126 # software or for further processing.
127
128 def WRITEPOINT():
129     global GG, Dt, DX, DY, XPOS0, YPOS0, XPOS, YPOS, XVEL, YVEL, XACC, YACC
130
131     moveTo(XPOS, YPOS)
132
133     # **************************************************
134     # **************************************************
135     # Main program
136
137     # First of all, the physical constants involved are  
138     # calculated in the M.K.S. system (metre, kilogram, second), with  
139     # exception of the distances of the orbits, because of the  
140     # enormous values involved. The AU (Astronomical Unit) is  
141     # more practical, where 1 AU = 1.495978707 x 10^11 metres  
142     # One AU is the average distance between the Earth  
143     # and the Sun. So, for example, if Halley’s aphelion  
144     # is found that to be 35.08 AU, this means that the  
145     # maximum distance of the comet from the Sun is equal to  
146     # about 35 times the distance between the Earth and the  
147     # In the end, however, all distance measurements are  
148     # converted to "points" for the purposes of graphical  
149     # representation.
150
151     G = 6.67E-11  # (N*m^2/Kg^2) Gravitation
4.7. **CALCULATE THE ORBIT OF HALLEY’S COMET**...

constant

\[ M_s = 1.99 \times 10^{30} \text{ (Kg)} \] # Mass of the sun

\[ D_p = 200.0 \] # Aphelion expressed in points

\[ r_{Af} = 35.08 \] # Aphelion (AU)

\[ D_t = 0.001 \] # Integration interval (time)

\[ K = D_p / r_{Af} \] # Scaling factor: number of points/AU

\[ G_{AU} = G / 1.496 \times 10^{11} \] # Gravitation constant expressed in AU

\[ G_p = G_{AU} \times K^2 \] # (N*p^2*Kg^2)

\[ G_G = G_p \times M_s \] # Gravitation constant inclusive of the solar mass (to reduce the number of multiplications in cycles)

\[ \varepsilon = 0.967 \] # 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 centre of the page, which we assume to be at the origin of the reference system and which we make coincident with the position of the Sun.

XPOS0 = getX() # origin coordinates (centre of page)
YPOS0 = getY()
hideTurtle() # better hiding the turtle: too slow...
```

# Determining initial conditions
# Let us put the comet at its initial position

```python
setPos(XPOS0 + rAf*K, YPOS0)
```

XPOS = getX()
YPOS = YPOS0

# The initial velocity of the body, given by Kepler’s law II,
# when the eccentricity and the solar mass are given

XVEL = 0.0
YVEL = sqrt(GG/(rAf*K)*(1-eps))

setPenColor("blue")
setPenWidth(1)

# This is where the trajectory drawing cycle begins.
# The points of the trajectory are calculated by the STP
# subroutine, then they are drawn by the WRITEPOINT subroutine,
# which also take care of writing data (position, speed,
# acceleration) to a file. However, WRITEPOINT is only
called once, as the trajectory is calculated in a
# large number of points, in order to evaluate the path
# with reasonable accuracy and this number would be far too
# large for drawing purposes.

# This implementation, tuned to reproduce highly
eccentric orbits, such as that of Halley’s comet,
# uses only one point out of 10000 to draw the orbit.

# This feature is controlled by means of the nWrite counter.

# A flag (yes/no variable), yIsNegative, is then used to
# check that the orbit is only drawn once.

nWrite = 0
yIsNegative = False  # Flag one orbit only

while not ( yIsNegative and (YPOS-YPOS0) > 0):
    nWrite = nWrite + 1
    if not yIsNegative and (YPOS-YPOS0) < 0:
4.7. **CALCULATE THE ORBIT OF HALLEY’S COMET...**

Listing 4.8: Calculating the orbit of Halley’s comet.

Here is the result, plotted in TigerYjthon’s graphics output window. Of course, you can also print out the point coordinates or save them to a file for further processing or other graphical representations. The point on the left is the sun.

Let’s go through the list of programmes. There are three sub-programs: NEWTON (Inst. 53-60) for calculating the acceleration based on the equation 4.4, STP (Inst. 76-113) for calculating the position (XPOS, YPOS) and velocity (XVEL, YVEL) at the next point\(^2\), WRITEPOINT moves the turtle to the next point and draws the path at the same time. Let’s rewrite the pseudocode of the page 146 to show the role of the different subroutines:

\(^2\)In this program we use a special interpolation to reduce the approximation errors inherent in calculating a continuous function in a discrete set of points. We use here the Runge-Kutta interpolation of the fourth order [5, pag. 704-708]. We do not need to go into the details of this interpolation here.
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 programme starts at instruction 137. Instructions 153-166 set the problem data: Sun mass (or whatever), aphelion (maximum distance from the Sun) and eccentricity of the 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 of the figure, where the velocity vector is vertical and upwards and vertical as far as the velocity module is concerned, it can be determined using the Kepler II law and the properties of ellipses:

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

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

where \(a\) is the length of the semimajor axis, \(\epsilon\) is the eccentricity of the ellipse and \(r_a\) is the aphelion.

---

\(^3\)This is true for the TigerYjthon environment, where the Cartesian origin is in the centre of the image space, the \(x\)-axis is positive to the right and the \(y\)-axis is positive to the top. In the case of LibreLogo, the same condition applies, except that the \(y\) axis is positive towards the bottom of the page.
4.7. *CALCULATE THE ORBIT OF HALLEY’S COMET*

Finally, we have the core of the main program between instructions 221 and 229. It’s just a loop to call STP and WRITEPOINT, with some controls to stop when the orbit is closed.

By changing the appropriate astronomical data, this program can be used to simulate orbits for other two-body systems, provided that the mass of the smaller body is negligible with respect to the other. It might be interesting to find such data and experiment with the code.
Bibliography


Chapter 5

Simulation: behaviour

We watch an ant make his laborious way across a wind- and wave-molded beach. He moves ahead, angles to the right to ease his climb up to a steep dunelet, detours around a pebble, stops for a moment to exchange information with a compatriot. Thus he makes his weaving, halting way back home.

— Herbert Simon, The science of the Artificial

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

So far we have presented a kind 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
how complex, the drawing is frozen in the code. So there is no place for randomness in the computer?

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

Basically, to create randomness you need to be able to generate random numbers, using what are 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 is to use algorithms that produce extremely long periods, so that successive extractions never reach the end of the sequence.

So they appear as real random numbers. In LibreLogo the Turtle understands the RANDOM command:

- $X = \text{RANDOM} 100$ ; returns a random float number$^1$ ($0 \leq X < 100$), that is equal or greater than 0 and smaller than 100.
- $X = \text{RANDOM} \text{“abcde”}$ ; returns a random letter from a, b, c, d, e
- $X = \text{RANDOM} \{1, 2, 3\}$ ; returns a random element from 1, 2 and 3

You can also mix different items, for example

$X = \text{RANDOM}\{1, \text{“pippo”}, 3.14\}$

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

\[ \text{PENCOLOR ANY} \]

you are going to use a random pen color.

$^1$In computer science a float number is a number with decimal places. For example, 3.14 is a float number, 18 is an integer number.
5.1. IS THERE A PLACE FOR RANDOMNESS IN COMPUTERS? THE TURTLE PLAYS THE TURTLE...

Let us take our original POLY program again (pag. 100) and modify it by randomly choosing the direction of the steps and the angle of rotation instead of passing them as parameters:

```
1   TO RAND
2       REPEAT [  
3           FORWARD RANDOM(10) 
4           RIGHT RANDOM(360) 
5       ]
6   END
```

Listing 5.1: POLY in random version

The structure is the same as POLY, except for the lack of parameters, but the behaviour is completely different! What’s the turtle doing here? Well, the turtle is doing the turtle, albeit a bit crazy. It’s a game, a simulation. You can play with it. In the previous code, the steps are randomly chosen between 0 and 10, while the turning angles are between 0° and 360°. These values make the turtle’s behaviour hectic, let’s give it a tranquilliser...
154  

CHAPTER 5. SIMULATION: BEHAVIOUR

Listing 5.2: RAND with different random choices

With FORWARD RANDOM(1) + 1 we choose step lengths between 1 and 2. With RIGHT RANDOM(90) - 45 we limit the deviations to angles between -45° and 45°—after all, who has ever seen a turtle turn 180°?

You can try the same code, but I could easily bet that your turtle didn’t follow exactly the same pattern as mine. To be honest, the chances of you getting the same picture aren’t exactly zero, but they’re extremely low: I’m pretty sure.²

---

²Further considerations and examples can be found in lesson 7 of the MOOC https://federica.eu/l/the_turtle_does_the_turtle
5.1. IS THERE A PLACE FOR RANDOMNESS IN COMPUTERS? THE TURTLE PLAYS THE TURTLE...

the contrary, the more complex the phenomena, the more simulations may prove to be the only possible investigative tool. In the context of biological disciplines we speak of \textit{in silico} experiments, referring to the fact that they are carried out through calculations performed by digital computers running on silicon chips - the expression is an allusion to the Latin phrases \textit{in vivo}, \textit{in vitro} and \textit{in situ}, that are commonly used in biology.

Today, \textit{in silico} experiments include molecular biology, genetic testing, tumour growth, dermatology, bone remodelling, organ failure, clinical trials, to name but a few. In medicine, for example, \textit{in silico} studies are used to discover new drugs because they are faster and much cheaper. In biology, they are used to formulate behavioural models of cells, and in genetics to analyse gene expression (in molecular biology, “gene expression” means the way in which a set of genes determines the functioning of the cell at the macromolecular level).

Beyond simulations, understanding the role of randomness in nature is one of the most important achievements of science. Since the 18th century, mathematicians have been building the edifice of statistics, the branch of mathematics needed to deal with randomness. In the 20th century, randomness burst into several fields of physics, for example in statistical mechanics to relate the macroscopic properties of gases to their microscopic particle nature, in quantum mechanics to describe the behaviour of subatomic particles, in nonlinear dynamics to describe the chaotic degeneration of deterministic systems. The overwhelming complexity that characterises biological, atmospheric and social systems makes dealing with randomness a daily business in these fields. The merging of technologies with social life is what drives us in the field of data mining. Last but not least, computer scientists find in randomness a great tool to deal with otherwise unsolvable problems. [1, pag. 201].

The philosopher and sociologist Edgar Morin describes human condition as about \textit{navigating in an ocean of uncertainty through archipelagos of certainty} [2]. In general, school curricula do very
little to give a correct perspective of our contemporary knowledge of
the world, in which uncertainty plays a crucial role. As a result, the
scientific vision that is given is skewed towards a false deterministic
description dominated by one-answer problems. Which is wrong.

5.2 Shaping the environment

Let’s give some context to our simulated turtle. In the previous examples,
the turtle walked around the page with no idea of the environment. Now
we are ready to let the turtle walk in a confined space. So we need a
shape for the allowed space and a behaviour to determine the turtle’s actions
once it reaches the boundary. Let’s switch to TigerYjthon, which is more
appropriate for the following considerations. Here we will use the “function”
name instead of the “new command” we used when talking about Logo. First
of all, we try to create an area, for example a circular one. We make it green,
turtles like grass.

The following examples are written in Python. They are a bit more
complex than the corresponding versions in LibreLogo, but later devel-
opments will be easier. Comments have been added to make the code
self-explanatory.

```python
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")
```
Listing 5.3: A round garden for the Turtle but no fence
The allowed area is defined here using the circle(r) function. This creates a bitmap that can be used as a mask to check if the turtle is inside or outside the allowed area. It is very easy to create arbitrary shapes in this way, a feature that will come in handy later.

The turtle’s behaviour is the same: apart from the presence of a circular green area, it is free to go anywhere. Now let us make the fence effective. The current version of step(data, ll, aa) is very simple:

```
1   def step(data, ll, aa):
2       forward(ll)
3       right(aa)
```

Listing 5.4: Function step(data, ll, a)

To prevent the turtle from getting over the fence, it is necessary to complicate this function considerably. Basically, the function checkStep(data, ll) moves the turtle “invisibly” before executing the usual forward right combination to check if the arrival point is on the green area. If not, the turtle is turned 10° to the right until the arrival point is inside the area. Then the execution continues as before.
# Do one step and one turn if arrival point is inside the green area. Otherwise keep turning right till arrival point is inside

```python
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 5.5: Function step(data, ll, a) with fence

Here is the result:
Of course, this code is by no means the best. On the contrary, it only sketches the possibilities. There are many possible improvements, even in these minimal versions. For example, are the contacts between the turtle and the fence described correctly? Try running the program several times and observing the collisions. Or what changes if we select different parameters? Or different angle increments in the function checkStep(data, ll)? There are many opportunities for exploration and reflection.

Now we have given our turtle a garden of almost any size, but life is getting a bit boring.

5.3 Modeling smell

We could imagine that there is some food in the garden and design some mechanisms to allow the turtle to find it “by sense of smell”. A very interesting aspect of computer simulation is that phenomena can be modelled in countless ways. Modelling is about trying to read reality and guessing the essential facts that determine behaviour. It is both a game and a thoughtful experiment.
Let’s assume that there is some lettuce somewhere and that the turtle is able to pick up variations in smell when it moves. To do this, we need to provide the following code:

1. define the garden boundary
2. plant a salad somewhere
3. provide the Turtle with the sense of smell, which depends on her distance from the salad
4. place the Turtle somewhere
5. start the game

To manage the odour, we will use a very simple model: if the turtle detects that the odour is getting stronger, it will keep going in the same direction, otherwise it will turn in a different direction. We build the code on top of this, which is given in the program listing on page 156.

The salad is provided as follows.

```python
1  def salad(sPos):
2      oldPos = getPos()
3      oldHead = heading()
4      setPos(sPos)
5      setPenColor("darkgreen")
6      setFillColor("green")
7      left(90)
8      repeat(5):
9          leaf()
10         right(30)
11     setPos(oldPos)
12     heading(oldHead)
13
14  def leaf():
15      startPath()
16      arc()
17      right(120)
18      arc()
19      right(120)
20      fillPath()
```
This is a very simple function, written just to place a salad in the garden. The “sPos” parameter of the “salad(sPos)” function represents the two coordinate position where we want to place the salad. The function uses “leaf()” to draw the individual leaves and this in turn uses “arc()” to draw the individual arcs that define a leaf. It was written with an example quoted by Seymour Papert [3] in mind. It could, of course, be quite different, and could be parameterised to adjust the dimensions, for example.

Then, to give the Turtle the power of detecting 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:

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

Listing 5.7: Calculation of distance

The loop that generates the turtle’s random walk is very simple, consisting of just a few instructions (instructions 50 and 51 on page 156):

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

Listing 5.8: Simple random walking loop

When the turtle wandered randomly, we had to enter the number of cycles. But if we can give the turtle a specific destination, we don’t need to predict the number of cycles.
5.3. MODELING SMELL

```python
p1 = getPos()
p2 = getPos()
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 5.9: Walking loop with food search

The value returned in “d2 = dist(p2,sPos)” is the distance at the last step and “d1 = dist(p1,sPos)” is the distance at the previous step. A “while” loop is used instead of the “repeat” loop so that we can let it run forever. The cycle ends when the turtle gets close enough to the food — “d2 < 15” in this version. The smelling is modelled in instructions 10-19: if the distance has increased in the last step, then call “step(data, randint(l1, l2), aa)” with a rotation angle of “aa = 20”, otherwise go straight on.

It may seem like too simple a model, but let’s see how it works...
Figures show the turtle path for six different combinations of turtle starting point and light source — turtle is shown at its arrival position. Left-right, top-bottom, respectively:

\((-100, -100) \rightarrow (100, 100), (100, -100) \rightarrow (-100, 100), (100, -100) \rightarrow (-100, -100), (-100, -100) \rightarrow (100, -100), (-100, -100) \rightarrow (100, 100), (100, -100) \rightarrow (-100, 100).\)

Our olfactory model may seem unrealistic because it gives the turtle stereotypical behaviour: if the distance increases, just turn 20 degrees to the right. However, regardless of the initial position of the turtle and the lettuce, the turtle always manages to get the food.

This is the power of feedback. Feedback mechanisms are ubiquitous in the dynamic processes of nature.

**Powerful concept — Feedback**

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

Natural systems grow and evolve through continuous adaptation to the context, which is also dynamic. Feedback systems are the engine
of these processes. The very basic experiment proposed here helps to focus on this concept. We have given the turtle the ability to receive feedback from the context at every step, according to the following scheme:

The turtle step, coded by the sequence of commands “forward(d)” and “right(\(\alpha\))”, causes the position coordinates to be incremented, 
\[ x_{n+1} = x_n + d_x \quad \text{and} \quad y_{n+1} = y_n + d_y, \]
where \(d_x\) and \(d_y\) are the Cartesian components of Turtle’s \(d\) step, and changing the heading 
\[ \alpha_{n+1} = \alpha_n + \alpha. \]

The distance from the food is sent back to the turtle as feedback, so that it can be taken into account in the next step.

The turtle is only able to turn 20°, but it has achieved its goal: the power of feedback, despite the poor model.

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

```python
1 p1 = getPos()
```
CHAPTER 5. SIMULATION: BEHAVIOUR

Listing 5.10: Casual walk with a little extra 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 each step:
The figures show the turtle’s path for six different combinations of turtle start point and light source — the turtle is shown at its arrival position. Left-right, top-bottom, respectively: 

\((-100,-100) \rightarrow (100,100), (100,-100) \rightarrow (-100,100), (100,-100) \rightarrow (-100,-100), (-100,-100) \rightarrow (100,-100), (-100,-100) \rightarrow (100,100), (100,-100) \rightarrow (-100,100).\)
5.4 Modeling sight

The sense of smell is based on a kind of local measurement of intensity. In the previous example we used the distance from the food location as the olfactory measure. This depends on the turtle’s position, but not on its orientation. The turtle had “no idea” about the direction of the salad. It only compared the intensity of the odour — measured by distance — in the current position with the previous one. Modelling vision is different because to “see” a light source you have to turn your head. Therefore, to model vision, we need a way to evaluate the orientation to 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 the turtle must turn to face a given object. First we need the direction of the line connecting the Turtle’s position and the object, say a light source.
5.4. MODELING SIGHT

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

The angle $\theta$ is calculated (Listing at page 258) with the function “towards(px, py)”. The parameters “px” and “py” are the coordinates of the light source. The “pcor” tuple holds the two current coordinates of the turtle. The function returns the $\theta$ angle, taking into account the domain of the “arctan” mathematical function.

```python
1  def towards(px, py):
2      pcor = getPos()
3      dx = px - pcor[0]
```
CHAPTER 5. SIMULATION: BEHAVIOUR

4 \[ dy = py - pcor[1] \]
5 if \( dy > 0 \):
6 \hspace{1em} if \( dx == 0 \):
7 \hspace{2em} return 0
8 \hspace{1em} if \( dx > 0 \):
9 \hspace{2em} return r2d(atan(dx/dy))
10 else:
11 \hspace{2em} return 360 + r2d(atan(dx/dy))
12 if \( dy < 0 \):
13 \hspace{2em} return 180 + r2d(atan(dx/dy))

Listing 5.11: Function towards(px, py)

Facing light

The bearing command can be used to point the Turtle directly at the light source, or to move the Turtle while maintaining a fixed bearing.

The function “face(px, py)”, which is called by the function “keep-Bearing(px, py)” (complete listing on page 258) makes the turtle face the light source, then turns the turtle to the right by a fixed amount (86° in this example) and sends it one step forward. The parameters px and py are the coordinates of the light source.

2 def face(px, py):
3 \hspace{1em} right(bearing(px, py))
4 5 def keepBearing(px, py):
6 \hspace{1em} i = 0
7 \hspace{2em} while True:
8 \hspace{3em} i+=1
9 \hspace{2em} face(px, py)
10 \hspace{2em} right(86)
11 \hspace{2em} forward(1)
12 \hspace{2em} if i == 10000:
13 \hspace{3em} return

Listing 5.12: Function face(px, py)
So what is the effect of such a model? If we try, we see that the turtle spirals around the point.

Figure 5.2: The fixed bearing model causes the Turtle to spiral around the point.

Why would such an unlikely behaviour be worth modelling? Well, there is actually a theory about the night flight of some insects: it seems that they fly along straight paths by keeping sky lights, such as the moon, at a constant bearing as they fly. This happens because the moon is very far away, but if they mistake the moon for a nearby light, the fixed bearing mechanism causes them to spiral. In this way, artificial lights become traps, and entomologists speak of the “fixation or trapping effect” [4, pag. 281]. Knowing this, our programme could be used to explore the concept of distance in this context. For example, at what distance does the trajectory
begin to approximate a straight line? You can do this by moving the light source out of the field of view...

**Powerful concept — Scope of a theory**

Here we have an opportunity to get an idea of what it means for a theory to be true, albeit in a somewhat simplistic way. Theories arise from specific experiences of the physical world. If we start from the context of night-flying insects, we come up with the idea that insects are able to fly in a straight line using fixed lights as reference points. And this minimal theory is true, as long as new experiments confirm it. Suppose we start experimenting with closer lights. Soon we will discover that our theory is no longer true, because by maintaining a fixed bearing to a close light, the insects will spiral. Does this mean that the previous theory was wrong? No. Very often in ordinary life, answers to these sorts of questions are seen as dichotomous. That’s a misconception about theories. A theory makes sense in a given context. If the context changes, we have to make sure that the theory still holds. In fact, in our minimal example, the first theory is somewhat “constrained” by the second. Insects always spiral around lights, but in the case of the moon they would spiral on an extremely large path in space. Within short distances, however, a curved path is very well approximated by a straight one.

This is what happens, for example, with Newton’s and Einstein’s theories of gravity. Newton’s theory is perfect for sending someone to the moon, but if you want to explain the effects of gravity on light, you need Einstein’s, but the latter does not contradict the former, it includes it.

**Two-eye vision**

Vertebrates have two eyes and so do turtles. Let’s give our turtle a mechanism for seeing with two eyes. We start by creating a separate field of view for each eye:
Here we define two functions, “rightEye(b)” and “leftEye(b)”, to determine whether the light source is within the field of view of each eye, given the bearing, which, as we have seen, is the angle by which the turtle would have to turn to face the light source.

```python
# 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 False
```
Once we have given the turtle the sense of sight, we need to model its behaviour. As in the case of smelling, we define a very simple model: if the source is seen by one or both eyes, the turtle moves 10 points forward, otherwise it chooses a random angle between 1 and 359 an turn by that angle.

The following code works with a loop that stops when the distance to the light source is less than or equal to 10 points. At each iteration, the bearing is calculated and then the presence of the light in the field of view is checked.

```python
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 5.14: Function headFor(px, py) drives the Turtle in the search of the light source
The figures show the turtle’s path for six different combinations of turtle start point and light source — the turtle is shown at 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 selection of a new direction by narrowing the range of possible angles. Or experimenting with a different field of view, or introducing an asymmetry between the left and right eye. Or introduce some randomness into the motion. Again, as with the previous olfactory model, we can see the effectiveness of the feedback mechanism, despite the extreme simplicity of the visual model.
Two-eye vision with intensity perception

A further refinement of the visual model is the intensity of the light stimulus perceived by each eye. This is a bit more complex, as it takes into account the strength and distance of the light source, as well as the angle of incidence relative to the viewing plane. Let’s see how this can be coded.

```python
1 def intensityLeft(px, py):
2     strength = 1000000
3     b = bearing(px, py)
4     if not leftEye(b):
5         return 0
6     fact = strength / (dist(px, py)**2)
7     a = bearing(px, py) - 45
8     return fact * cos(a*pi/180)
9
10 def intensityRight(px, py):
11    strength = 1000000
12    b = bearing(px, py)
13    if not rightEye(b):
14        return 0
15    fact = strength / (dist(px, py)**2)
16    a = bearing(px, py) + 45
17    return fact * cos(a*pi/180)
```

Listing 5.15: The functions intensityLeft(px, py) and intensityRight(px, py) return the perceived intensity

The light source variable, “strength”, has been set to an arbitrary value to provide a suitable scale for drawing. If the source is not in the field of view, the value returned is 0. Otherwise the value returned is “strength” multiplied by the inverse of the square of the distance and by a factor \( \cos(\alpha) \), where \( \alpha \) is the angle of incidence: for perpendicular rays \( \alpha = 0 \) and \( \cos(\alpha) = 1 \). Hence intensity = strength \times \cos \alpha / \text{distance}^2.

When calculating the angle \( \alpha \) we have to take into account the fact
that both eyes are 25° away from the turtle’s direction, the right one in one direction, the left one in the other — in other words, the bearing has to be calculated with respect to the direction of the eyes instead of the direction of the turtle. In the following listing, the headBySight(px, py) function controls the turtle’s path by means of a while loop that stops (in this example) when the turtle is 3 points away from the light source.

```python
1 def headBySight(px, py):
2     i = 0
3         while dist1(px, py) > 3:
4             i = i + 1
5                 left(random.randint(-90, 90)) # some randomness here
6                 forward(1)
7             iL = intensityLeft(px, py)
8             iR = intensityRight(px, py)
9                 if iL > iR:
10                     left(10)
11                 elif iL < iR:
12                     right(10)
13                     else:
14                         while intensityLeft(px, py) == 0 and
15                             intensityRight(px, py) == 0:
16                             left(random.randint(-180, 180))
Listing 5.16: The functions intensityLeft(px, py) and intensityRight(px, py) return the perceived intensity
```

In this version, a bit of randomness is added by instruction N. 5, where the turtle turns left at a random angle between -90° and 90°. If this instruction is removed, the algorithm works in the same way, the only difference being that the turtle moves more directly to its destination. At each cycle, the turtle makes a one-point step (instruction N.6), 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 its left, it turns slightly to the left (instructions N.9-12). In this way the turtle moves forward, trying to keep the amount of light received by each eye the same. If the intensities are the
same, but both equal to zero, then the turtle will keep trying new random left turns until at least one of the eyes sees something. If the intensities are equal but not zero, the turtle does nothing.

The following figure shows the paths chosen by the Turtle with this model for six different combinations of starting point and light source position.

The figures show the turtle’s path for six different combinations of turtle start point and light source — the turtle is shown at 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, the possibilities for exploring different situations are endless. For example, what happens if one eye sees less light than the other? What if you masked one of them? Or you could observe how the turtle behaves in the presence of one or more additional light sources.
Powerful concept — Scalar and vector fields The examples we have explored so far in this chapter all implicitly invoke the concept of field. In physics, the concept of field underlies a wide range of topics. In this context, it is sufficient to describe it as a physical quantity, represented by a number or vector, that has a value for every point in space. For example, a weather map that gives a temperature value for each point describes a scalar field. In physics, the term scalar means a quantity described by a simple number. The graphical representation of a scalar field, for example in two-dimensional space, may consist of numbers printed at certain points on a map. Instead, a vector is a physical quantity that must be described by an intensity, a direction and a sense. Graphically, vectors are represented by arrows, where the length represents the intensity and the arrowhead indicates the direction. A weather map showing the distribution of winds will have vectors represented as arrows superimposed on the map.

The lettuce search tasks we’ve seen so far are not strictly physics problems, but they easily evoke the concept of field. In the case of the smell model, the turtle was only concerned with the intensity of the smell and was 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 the odour was proportional to the distance.
Figure 5.3: The sense of smell is modelled by estimating the distance to the food at each step (page 160). Numbers represent distances expressed in pixels. Odour intensity is proportional to the inverse of the distance.

Instead, in the case of the vision models, where the turtle was trying to reach a light source, we were concerned with direction and sense, not just intensity, and thus involved the concept of vector field.
5.5 Modeling interactions

Multitasking

A “handmade” example

The previous examples pave the way for modelling interactions between turtles. But first we need a way to handle multiple turtles. This is possible in the TigerJython environment, not in LibreLogo. Basically, what we need to make multiple Turtles work at the same time is parallel processing, which means executing multiple sequences of instructions independently at the same time. The Python language used in the TigerJython environment has good support for multithreading. We’ll see some examples, but first we’ll try to build a multithreaded program with LibreLogo, even though it’s strictly a single-Turtle environment. Such a ”handmade” example allows us to delve a little into the concept of multitasking that we are so used to today.

The trick is to create two separate sequences of instructions, one for each turtle, in which only a single step is executed. Suppose we have a red
turtle and a green turtle:

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

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

Listing 5.17: 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 alternately, provided that the state of each turtle is restored before its next move and saved after completion.

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

Listing 5.18: Managing two turtles at once

The states of the two turtles are saved and restored using the functions “SAVERED”, “RESTGREEN”, “SAVERED”, and “RESTRED”. For example, for the red turtle we have (complete program list at page 269):

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

7 TO RESTRED
  GLOBAL PRED, HRED, PGREEN, HGREEN
  X = PRED[0]
  Y = PRED[1]
```
5.5. Modeling Interactions

Here is an example of three runs.

```
11 PENUP
12 POSITION [X,Y]
13 PENDOWN
14 HEADING HRED
15 END
```

Listing 5.19: Turtle state management

Three runs of LibreLogo EXECBOTH program (page 182).

Multitasking: the computer juggling trick

Computers are designed according to the von Neumann model, i.e. as sequential machines which, based on a programme, execute instructions one after the other in small successive time intervals. True multitasking is only possible when different processors perform different tasks at the same time. Today there are massively parallel computers made up of 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 of today’s computers, which tend to have multicore processors — for example, the laptop I’m writing on has a four-core processor. However, this feature is invisible to the user because it is used by some application software to speed up processing in a peculiar way. In fact, we take it for granted in our daily lives 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 does one thing after another, very quickly. The example we have shown is very simple indeed, but it can be used to develop some ideas about the behaviour of computers when it comes to managing simultaneous tasks. The concept of multitasking can be considered at different levels, for example as concurrent threads in a single application — in this case we speak of multithreading — or as competing processing within the operating system. In any case, this exercise can be used to show that there needs to be some kind of software hierarchy to manage concurrent activities. For example, the EXECBOTH procedure runs at a higher level because it is responsible for allocating tasks to the EXECRED and EXECGREEN procedures.

Interacting Turtles

Predator prey

Let’s turn to the TigerJython environment to tinker with interacting turtles. TigerJython is more suitable as it has true multithreading support. In section 5.4 on page 176 we modelled a two-eye vision system based on intensity perception and used it to simulate a turtle’s path to a light source. Here we take advantage of multithreading to let two turtles run simultaneously. In our simulation, turtle 1 plays the role of the predator, while turtle 2 plays the role of the prey. Turtle 1, the predator, simply spins in a circle, oblivious to everything else — this could be the situation of a rainforest bird, intent on performing its complex dance moves, unaware of the approaching predator. Turtle 1, the predator, is tracking its prey using its own vision, based on the same model described in section 5.4, where the turtle had to reach a light source. Of course, in this case the target is moving, so the predator’s paths can vary greatly 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.

There is a lot to experiment and think about here. For example, one could study the effects of different choices for the relative starting positions, the prey’s path — in this case, for example, the radius of the trajectory circles — and the relative speeds. You could also change the parameters of the prey’s vision model. Or give the prey a chance to sense the presence of the predator, for example through the olfactory model we discussed in section 5.3 (pag. 160).³

Contagion dynamics

During the COVID-19 pandemic caused by the SARS-CoV-2 coronavirus, many analyses of the dynamics of COVID-19 infections were reported in the media. Thanks to TigerJython’s turtle geometry and multithreading, it is possible to simulate the dynamics of infection in a community. In the following example, we have 100 turtles moving randomly in a circular region. At the beginning, all but one are healthy. As the turtles move around, every time a healthy turtle stumbles over a sick turtle, it gets sick.

³Recently, it has been discovered that cheetahs are excellent night hunters thanks to their extraordinary night vision, while their prey must rely primarily on their sense of smell to prevent their attacks.
and becomes contagious. In this version, the level of contagiousness of the virus can be adjusted by the threshold distance below which a sick turtle can infect a healthy one.

Progression of contagion in a population of 100 Turtles of which only one is initially ill.
In this setting, two factors have an important influence on the dynamics: the contagiousness of the disease and the closed environment. The typical trend is therefore a sudden increase in the number of cases, followed by a more or less rapid decline. The first part is due to the exponential nature of the contagion phenomenon, as the number of new infections is proportional to the number of sick turtles. The second is due to the saturation of the population.
The course of contagion in 12 different runs of the simulation. The histogram shows the number of new infections in each iteration.

But even in this small example, we can see how the infection progresses in bursts that suddenly involve a large number of subjects. The contagion then declines because healthy subjects are no longer available. Of course, the simulation can be extended to larger samples, possibly changing other parameters such as infection distance or starting conditions. Here we limit ourselves to highlighting how these outbreaks represent the seeds of the 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 carrier of the disease.

Reflection — Are we sure we "feel" the vertigo of exponential growth?

Exponential growth should be anxiogenic because of what it implies: essentially an irreparable loss of stability. But the expression $2^n$ hardly evokes this feeling, except in those who have some mathematical insight. Perhaps 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 [5, 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 sparks from Beatrice’s words multiplies like the flames of iron in the forge, more than the doubling of chess.

Legend has it that the inventor of chess asked the king of Persia, who wanted to compensate him, to put one grain of wheat on the first chessboard, two on the second, four on the third and to double away. This seemed to the king to be nothing, but when he did the maths it turned out that not even the cultivation of all the known lands with wheat would have been able to satisfy this request!

Let’s try to get an idea of this amount. First we need to work out how many grains of wheat there are. We place one grain in the first square. In the second we put 2, so there are 3 in all. In the third we put 4, making a total of 7. It is easy to see that if the number of squares we fill is $n$, the number of grains is $2^n - 1$. So for 64 squares, the total number of grains is given by $2^{64} - 1$.

This little script seems harmless, but 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
5.5. MODELING INTERACTIONS

\[
10^x = 2^{64} \quad (5.1)
\]

Using logarithms you get \(x\):

\[
x = 64 \log_{10} 2 = 19.3 \quad (5.2)
\]

So the number of seeds is about \(10^{19.3}\). In other words, there would be more than 10 billion billion seeds. What does this mean? If you use a scale, you can easily see that a handful of about 100 seeds weighs about 5 grams. So a seed weighs \(5 \times 10^{-2}\) grams, so the amount the mathematician would need would be \(5 \times 10^{11.3}\) tonnes. That’s a lot, since the world’s average wheat production is “just” 35 million (\(35 \times 10^6\)) tonnes per year: the amount of wheat the mathematician needed would therefore last more than 28000 years for all of humanity! Beware of exponents!
Bibliography


Chapter 6

Simulation: fractals growth

Why is geometry often described as "cold" and "dry"? One reason lies in its inability to describe the shape of a cloud, a mountain, a coastline, or a tree. Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line. More generally, I claim that many patterns of Nature are so irregular and fragmented, that, compared with Euclid — a term used in this work to denote all of standard geometry — Nature exhibits not simply a higher degree but an altogether different level of complexity. The number of distinct scales of length of natural patterns is for all practical purposes infinite.

— Benoit B. Mandelbrot, The Fractal Geometry of Nature
Everyone knows that two mirrors facing each other create an incredible fugue of images. Mirror number 1 can only do one thing: reproduce the scene in front of it. Mirror number 2 can do 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. This is an amazing phenomenon because it allows us to peek into infinity, which is normally inaccessible to human experience. This is recursion.

What’s going on here? Nothing at all! The RECURSION program keeps calling itself without doing anything else — a perfect example of self-referentiality! In this code fragment, the turtle executes only one command: RECURSION. Actually, if you try to write this code in LibreLogo and run it, nothing will happen except that 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 can get the computer into trouble if there is no stopping criterion. Each time a procedure is called, the system allocates some memory for its activities. If you let the machine continue with this process, an infinite amount of memory will be allocated, which is obviously not possible.

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

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

Listing 6.1: Minimal recursion program

```
1    TO RECURSION D
2       CIRCLE D
3       RECURSION D+1
4       END
5
6    RECURSION 1
```

Listing 6.2: Slightly less minimal recursion program
This is more fun, try it: what do we get? You should see a balloon growing. However, you need a stop rule, because this code will make the balloon go over the page: Recursive programs need a stop rule.

```
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 stop rule we need a statement that evaluates a given condition. In LibreLogo we can do this with the IF command together with a condition, which in this case could be “D < 100”. Instruction No. 2 says that if D is less than 100, then instructions 3 and 4, inside the square brackets, will be executed, otherwise not. In this case, we leave the program RECURSION and instruction n. 10 is executed.

In TigerYjthon this program would look like this:

```
from gturtle import *
makeTurtle()
clearScreen()
hideTurtle()
def rec(d):
  delay(100)
  if(d < 100):
    dot(d)
    rec(d+1)
rec(1)
print("Done!")
```

Listing 6.4: Recursion Python program with stopping rule
Apart from syntactical differences, we added a delay at statement N. 8 because the program runs much faster in TigerYthon and the growth of the balloon would not be visible.

What could these simple and unhelpful examples be used for so far? Why bother with these fanciful but strange constructs when 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 recursive description, for example fractals.

## 6.1 Fractals

A fractal is a never-ending geometric pattern, that is, an infinitely complex pattern that is self-similar across scales. Fractals became popular, even in mathematical circles, in the 1980s, thanks to the work of Benoît Mandelbrot. In fact, fractals were known before they had a name. For example, the Cantor set, published by Georg Cantor in 1883 [1, pag. 67] — an example of which we will give later. However, before Mandelbrot, Cantor’s set and other strange entities were regarded as exceptional objects, as “mathematical monsters”. Mandelbrot’s merit was to have shown that these “extreme shapes”, now called fractals, actually represent the normality instead of the exception, showing that there are countless examples of fractal structures in nature. Hence the title: *The Fractal Geometry of Nature* [2].

Fractals look the same on different scales, that’s what self-similarity means. We have already encountered self-similarity in Bernoulli’s spiral, *spira mirabilis*, i.e. the logarithmic spiral (pag. 92 and following), another overrepresented shape in nature. To get an idea about growing fractals, a very simple example may be appropriate.

```
1  TO TREE LL
2  IF LL > 2 [ 
3    FORWARD LL
4    LEFT 50
5    TREE LL/2
6    RIGHT 100
```

\(^1\)The topic is not trivial, experts can discuss it at length. In our context, “everything” is ok.
The TREE procedure is recursive because it calls itself. The fact that there are two calls has to do with the recurring bifurcation structures that are the “basic idea” of this tree. To tell the truth, there is nothing but bifurcations in this tree, the only differentiation being the spatial scale. Let’s look at some details. First, the program is called in line N. 13: TREE 200. Therefore, when TREE is entered, the value of LL is checked to see if it is less than 2 (Instruction N. 2). Since this time it’s value is 200, the execution continues. With instructions N. 3 and 4, the turtle moves forward by LL=200 points and turns to the left by 50°, drawing the trunk and turning itself to the left.
Then, instead of continuing to draw, it calls TREE (Instruction N. 5), but passes it a value of LL/2. Let us not delve into this TREE call and assume that we have done it. In instruction N. 6 the turtle turns 100° to the right, in N. 7 calls TREE LL/2 again, then in N. 8 it turns 50° to the left and comes back along the last branch (Instruction N. 9).

The study of fractals makes it easier to understand the concepts of infinity and infinitesimal through the often fascinating graphical representations. Let’s play a little with the stick tree to explore the interplay between infinity and infinitesimal. To do this, we will rewrite the code in a slightly more complex way. We are using a Python version here because the program runs faster.

```python
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)
        left(aa)
        back(ll)

setPenColor("dark green")
setPos(0,-100)
ll = 200 # First branch length
```

---

1We are actually jumping between the Logo and Python programs. It’s a good exercise, try translating some small programs from one language to another.
Two things have changed since the last Logo version: 1) we added the parameter “aa” (angle) to the “tree” subroutine and 2) instead of generating two branches at each recursion level, five branches are now generated. Instead we kept the scaling factor $1/2$, i.e. at each recursion the tree command is called with the branch length parameter “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 reflections on the concepts of infinity and infinitesimal. A stopping rule has been added to the “tree” command in this version,
requiring that the length of the branches is not less than 10. Starting with \( l_l = 200 \) we get four levels of recursion with values \( l_l = 100, 50, 25 \) and 12.5.

It is obvious that if we had set the threshold to a higher level, we would have obtained a shorter tree with fewer branches, such as the following one, where we set the threshold to 30 and obtain only two additional levels: \( l_l = 100, 50 \).

Figure 6.3: Same tree but with only 2 levels of recursion.

On the other hand, by lowering the threshold and allowing the creation of smaller branches, the tree grows taller and its crown thicker. But what happens if we allow the recursion process to continue indefinitely? Will the tree become infinitely tall by adding more and more branches, or will the progressive reduction in branch size compensate for the explosion of the tree? Which of these two opposite tendencies will prevail?

You can try it yourself, it’s very easy: all you have to do is change the stop rule at statement No. 9 in the program listing on page 201. For example, we now try “if(\( l_l > 0.1 \)):”
In this case, the recursion levels are ten: \( l_1 = 100, 50, 25, 12.5, 6.25, 3.125, 1.56, 0.78, 0.39, \) and \( 0.19 \). The vegetation over there is really dense! But is that tree really taller? It might be. Let’s try to compare them by placing a 400-point ruler in the middle of each tree.
6.1. FRACTALS

Figure 6.5: Transition from four to ten recursion levels. The distance between the two horizontal bars is 400 pixels. The vertical bars mark the beginning of successive branches, proportional to 1/2, 1/4, 1/8...

In fact, the tree continues to grow with the number of recursions, but at an increasingly slower rate. Will this growth remain below a certain limit as the recursion goes to infinity? Computer experiments could be refined, but only mathematics can provide a definitive answer. For this reason, I have chosen a structure of our tree that is suitable for obtaining the answer by using a well-known mathematical relationship. Each recursion generates five equidistant branches, but the central branch has no angle of deviation from the original branch. Thus, at the center, we have all the subsequent branches aligned on a vertical line that reaches the top of the tree, which coincides with the height of the tree. We see that the successive branches are reduced by a factor of 2 with each recursion. Thus, in the case of the first tree, the length of the central vertical line is given by

$$H = d + d/2 + d/4 + d/8 + d/16 + d/32 \quad (6.1)$$

where $H$ is the total height and $d$ is the length of the first branch, which is the trunk ($d = 200$ in our code). Mathematics gives us the tools to deal
with infinity: we cannot make our computer do infinite recursions, but we can write:

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

This is a geometric progression for which

$$\sum_{n=0}^{\infty} a^n = \frac{1}{1-a}$$  \hspace{1cm} (6.3)

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

$$H = d \frac{1}{1 - \frac{1}{2}} = 2d$$  \hspace{1cm} (6.4)

This is exactly the value we used to draw the red rulers. Now we know that the recursion process will create an increasingly dense crown that approaches, but never exceeds, the height of 400 points.

If we look at Fig. 6.5, we see that as we continue with the recursions, the tree will grow no more than $2d$ in height. We know this from our knowledge of mathematics — especially geometric progression. What we are doing by pushing the process to infinity is filling more and more parts of the plane closer and closer to the upper limit of 400 points. Even though we are only drawing segments, the recursive process leads us to fill the plane to some extent. But to what extent? It will be the mathematician Mandelbrot who will give us this answer when we tackle the problem of measuring the coastline of Britain (page 224).

**L-systems: the turtle language for growing plants**

Turtle geometry was used to model plant growth [1, 3]. The stick tree algorithm (Page 199) we wrote in the previous section actually represented the growth of a plant, albeit an extremely simple one. We chose this example to make the recursion mechanism as understandable as possible. To facilitate the creation of more complex structures, scientists use various methods,
including a special formal language called L-system. The advantage is that once the method is understood, it is much easier to tinker with different and more complex forms. So let us rewrite the stick tree algorithm using the L formal language.

A so-called L-tree system is specified by three components: a alphabet, a axiome and a set of production rules. Each production rule is a string composed of labels belonging to a predefined alphabet. The alphabet determines the category of plants we want to model. The axiom and production rules determine the specific plant we want to simulate. Square brackets are used to delimit branches: items in square brackets define a specific branch. The brackets [ and ] are encoded as push and pop operations on the turtle state stack. The following table shows the alphabet we will use to model herbaceous plants.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>move forward by a certain fixed step length drawing a line</td>
</tr>
<tr>
<td>f</td>
<td>move forward as above for F but do not draw the line</td>
</tr>
<tr>
<td>+</td>
<td>turn left by a fixed angle</td>
</tr>
<tr>
<td>−</td>
<td>turn right by a fixed angle</td>
</tr>
<tr>
<td>[</td>
<td>new branch opening...</td>
</tr>
<tr>
<td>]</td>
<td>... current branch closing</td>
</tr>
<tr>
<td>B</td>
<td>do nothing (i.e. turtle does not go anywhere)</td>
</tr>
</tbody>
</table>

How are these instructions translated into turtle commands? Tree L-systems are set up by defining forward steps and rotation angles once for all, so that $F$ could be coded as forward($l$), “+” as left($d$), and “−” as right($d$), where, for example, $l = 100$ points and $d = 50^\circ$. The $B$ command is a kind of no operation command, which results in a recursion call without any further action. From a general point of view, an L-system generates a sequence of stages, where the production rules tell how to get from one stage to the next, 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, consider the simple example of the stick tree\(^3\), which we saw on page 199.

\[
\begin{align*}
& \text{Axiom: } B \\
& \text{Production rule: } B \rightarrow F[+B][−B] \\
& \text{Production rule: } F \rightarrow FF \\
\end{align*}
\]

(6.5)

And this is the corresponding program:

```python
from gturtle import *  # import turtle library

makeTurtle()  # create a turtle instance
clearScreen()  # clean previous drawings
hideTurtle()  # hide Turtle while drawing for higher speed

def tpush(lp, lh):  # store current Turtle status
    lp.append(getPos())
    lh.append(heading())

def tpop(lp, lh):  # recover previous turtle status
    setPos(lp.pop())
    heading(lh.pop())

def B(l, delta, iter, lp, lh):  # production rule of B
    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)  # recursive 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)  # recursive call
        tpop(lp, lh)  # ] closing right branch

l = 100
delta = 50
iter = 7
```

\(^3\)As the shape is reminiscent of a simple tree, we will continue to refer to it as such. However, the simple growth scheme is that of a herbaceous plant.
6.1. FRACTALS

There are three functions in this program: tpush(lp, lh), tpop(lp, lh) and B(l, delta, iter, lp, lh). The first two are used to store the current state of the turtle and retrieve it later if needed. The B(l, delta, iter, lp, lh) function implements the $B \rightarrow F[+B][-B]$ production rule of the equation 6.5.

Concept of stack Stacks are a fundamental construct in computer science. Basically, a stack is a collection of elements that can be accessed using two operations: put to add new elements and pop to retrieve elements by removing them from the stack. Stacks have a LIFO (Last In First Out) data structure: the last element stored is retrieved first. Think of a stack of sheets of paper: the last sheet added to the stack is on top, so it pops out first. Adding a sheet is called “pushing” and removing it is called “popping”.

In addition to stack, there is the concept of queue, which is also about storing information, but the access is different. Queue has a FIFO (First In First Out) data structure: the first item stored is retrieved first. Imagine a line in a supermarket. The first person in the queue is the first to leave. A person joining the queue is “enqueued” and a person leaving the queue is “dequeued”.

In our case, we need a LIFO data structure to build branch structures. In this case, the data are turtle states, another powerful concept we mentioned on page 69. Each turtle state element is composed of three numbers: two coordinates for the turtle’s position in the plane and an angle for the turtle’s pointing direction. State stacks are useful because each time the turtle finishes drawing a branch, it has to

```python
32 lp = []
33 lh = []
34 B(l, delta, iter, lp, lh)
```

Listing 6.7: Basic recursive function for the L-system stick tree
Remember the state it had just before it started drawing the branch.

The recursive process is thoroughly defined by the B(l, delta, iter, lp, lh) function (statements 15-27). This function takes 5 arguments: “l” is the length of the forward F and f steps; “delta” is the deviation angle of the “+” and “-” rotations — these are parameters that influence the final aspect of the plant; “iter” is the number of recursions, “lp” and “lh” are the positions and headings accumulated in the stack, respectively. Depending on the number of iterations, two kinds of actions can take place in this function. If the iteration counter is equal to 1, i.e. we have reached the bottom of the recursive process, then one last F step is performed and the function is exited. It is important to realize that this is the only way to exit the function, i.e. each call to B() will remain pending until the bottom of the recursion is reached. If the iteration number is not 1, the production rules are applied. The nine statements between lines 19 and 27 uniquely correspond to the nine characters $F[-B][+B]$ of the first production rule, $B \rightarrow F[-B][+B]$. Finally, the second production rule, $F \rightarrow FF$, translates to halving the step size on each function call: B(l/2, delta, iter, lp, lh) (statements 22 and 26); this makes the branches in our tree smaller and smaller.

Let us now see how different shapes can be obtained by playing with the tree L-system. The following is a variation of the previous stick tree, which is the one shown on the cover of Turtle Geometry [5, p. 85].
The following specifications were used to obtain this tree:

\[
\begin{align*}
\text{Axiom: } & B \\
\text{Production rule: } & B \rightarrow F [ +FB ] [ -B ]
\end{align*}
\]

which results in the following code:

```python
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)  # recursive call
        tpop(lp,lh)  # ] closing right B
        tpush(lp,lh)  # [ beginning right B
        right(delta)  # - right rotation
        B(l, delta, iter -1, lp, lh)  # recursive call
        tpop(lp, lh)  # ] closing right B

l = 20
delta = 20
iter = 8
lp = []
lh = []
B(l, delta, iter, lp, lh)

Listing 6.8: Asymmetric variant of the stick tree

This asymmetric version of the stick tree is given by a slightly different
first production rule: \( B \rightarrow F[+FB][−B] \) instead of \( B \rightarrow F[+B][−B] \);
in addition, in this variant the second production rule is simply omitted,
meaning that we have no branch length reduction throughout the recursive
process.

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

from gturtle import *

Options.setPlaygroundSize(600, 1000)  # set drawing area
makeTurtle()
clearScreen()
6.1. **FRACTALS**  

```
6 hideTurtle()
7
8 def tpush(lp, lh):
9     lp.append(getPos())
10    lh.append(heading())
11
12 def tpop(lp, lh):
13     setPos(lp.pop())
14     heading(lh.pop())
15
16 def F(l, delta, iter, lp, lh):
17     if iter == 1:
18         forward(l)
19     else:
20         F(l, delta, iter-1, lp, lh) # F recursive forward
21         tpush(lp, lh)  # [ beginning right F
22         left(delta)  # + left rotation
23         F(l, delta, iter-1, lp, lh) # F recursive forward
24         tpop(lp, lh)  # ] closing right F
25         F(l, delta, iter-1, lp, lh) # F recursive forward
26         tpush(lp, lh)  # [ beginning left F
27         right(delta) # - right rotation
28         F(l, delta, iter-1, lp, lh) # F recursive forward
29         tpop(lp, lh)  # ] closing right F
30         F(l, delta, iter-1, lp, lh) # F recursive forward
31
32 l = 4
33 delta = 25.7
34 iter = 6
35 setPos(0,-490)

37 lp = []
38 lh = []

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

Listing 6.9: A grass plant
This kind of grass is obtained by means of the following tree L-system:

\[
\begin{align*}
\text{Axiom: } & F \\
\text{Production rule: } & F \rightarrow [+F][F[-F]F
\end{align*}
\]

(6.7)

Also in this case we have only 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.

What is the difference between using \( F \) or \( B \) as an axiom? To understand better, let us look at the role of the axiom for a moment. In this example, the axiom is \( F \), which means that the substitutions specified by the production rule must be applied to all occurrences of \( F \), at every iteration.

This is better illustrated in the following picture.
Figure 6.6: The first four recursions of the grass fractal simulation. The images are scaled so that the total height of the grass is the same. Comparing the first and second images, we see the graphical expression of the production rule.

While it is fairly straightforward to see what is meant by substituting $F$, what could it mean to substitute $B$ instead, the “no operation” label? Let us look at the first three iterations of the simple tree construction process.
With this picture in mind, it is probably easier to grasp the meaning of a “no-operation” label $B$ in the production rules given by the equation 6.5. Actually, the stick tree is made only by bifurcations, with progressively smaller branches. Each “$B$-point” is nothing but a bifurcation with two branches coming from it. The axiom is $B$, since the process is started in statement N. 46 (Page 208) with a call to B(l, delta, iter, lp, lh). The first production rule, $B \rightarrow F[-B][+B]$, tells what to substitute for each $B$ occurrence. The second production rule, $F \rightarrow FF$, tells what to substitute for $F$ in the previous rule - this second rule tells how to reduce the branch length at each recursion level.
Here is another example of a grass simulation, very different from the one shown on page 214, which was achieved using only bifurcations:

\[
\begin{align*}
\text{Axiom: } & B \\
\text{Production rule: } & B \rightarrow F[+B]F[-B] + B \\
\text{Production rule: } & F \rightarrow FF
\end{align*}
\]

(6.8)

```python
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) # recursive call
```

```python
```
CHAPTER 6. SIMULATION: FRACTALS GROWTH

Listing 6.10: Another grass plant

Finally the simulation of a bush:

\[
\begin{aligned}
\text{Axiom: } & B \\
\text{Production rule: } & B \rightarrow F[B]F[-B] + B \\
\text{Production rule: } & F \rightarrow FF
\end{aligned}
\]  

(6.9)
6.1. FRACTALS

```python
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 forward
        F(l, delta, iter-1, lp, lh) # F forward
        left(delta) # + left rotation
        tpush(lp, lh) # [ beginning right F
        left(delta) # + left rotation
        F(l, delta, iter-1, lp, lh) # F forward
        right(delta) # - right rotation
        F(l, delta, iter-1, lp, lh) # F forward
        right(delta) # - right rotation
        F(l, delta, iter-1, lp, lh) # F forward
        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 forward
        left(delta) # + left rotation
        F(l, delta, iter-1, lp, lh) # F forward
        left(delta) # + left rotation
        F(l, delta, iter-1, lp, lh) # F forward
        tpop(lp, lh) # ] closing left F

l = 8
delta = 25
iter = 5
setPos(0,-200)
lp = []
lh = []
```
Thus, we have seen different examples from which the reader can invent new ones, playing with the elements of the production rules. At first glance, this seems to be a rather complex topic to propose in 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 shapes. 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.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>forward(l)</td>
</tr>
<tr>
<td>$f$</td>
<td>penUp()</td>
</tr>
<tr>
<td></td>
<td>forward(l)</td>
</tr>
<tr>
<td></td>
<td>penUp()</td>
</tr>
<tr>
<td>$+$</td>
<td>left(delta)</td>
</tr>
<tr>
<td>$-$</td>
<td>right(delta)</td>
</tr>
<tr>
<td>$[\ ]$</td>
<td>tpush(lp,lh)</td>
</tr>
<tr>
<td>$[\ ]$</td>
<td>tpop(lp,lh)</td>
</tr>
<tr>
<td>$B$</td>
<td>recursive call B(...)</td>
</tr>
</tbody>
</table>

Table 6.2: Where l is the step length and delta is the deviation angle, determined at the beginning once for all.

The Cantor dust

In the table above, there is one element, $f$, that we have not used in the examples so far. It is the element that performs a step 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, long before Benoit Mandelbrot introduced the concept of a fractal in the 1960s. Actually, there were several fractal objects in circulation before that, but they were considered mathematical monstrosities that were the exception rather than the rule. The construction of the Cantor set is very simple. You start by drawing a horizontal line. Then it is divided into three equal parts, of which only the first and the last are drawn, leaving a gap in between. And so each line drawn is subjected to the same process in turn. So you can think of it as an increasing number of spaces interspersed with a series of scattered dots: the “Cantor dust”. The process of creating the Cantor set is very easy to express with an L-system:

\[
\begin{align*}
\text{Axiom: } F \\
\text{Production rule: } F \rightarrow FfF
\end{align*}
\] (6.10)

```python
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)
```
Chapter 6. Simulation: Fractals Growth

Listing 6.12: Cantor dust

This code produces the following picture:

```
setPos(-500,100-(iter-1)*50)
```

In these examples, we have added a number of clues that you can try to explore on your own. It is amazing what can be done with the L-system language. The paper by Prusinkiewicz et al [3] shows some examples in three dimensions. It is well worth a look.

6.2 Fractals and randomness

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

```
TO TREE LL
A = RANDOM 50
IF LL > 2 [ FORWARD LL LEFT A TREE LL*3/5 RIGHT A*2 TREE LL*3/5 LEFT A BACK LL ] END
```
6.2. FRACTALS AND RANDOMNESS

Listing 6.13: A livelier tree in Logo

```
14 TREE 50
```

Figure 6.8: Three different executions of the previous code

The main difference from the original code is in instruction N. 2, where the rotation angle A is chosen as a random number between 0° and 50°.

We can make the example even more realistic by, for example, adding a random number between -2 and 2 to the step size, which in this example is set to 50 in the first iteration.

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

Listing 6.14: An even livelier tree
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.

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

An English meteorologist, Lewis Fry Richardson (1881-1953) [6], faced the problem of measuring coastlines. He used a simple procedure to find the perimeter of a natural object by measuring the coast with a ruler (yardstick) of a certain length. In the process, he discovered a startling 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, if the length of the ruler is reduced, the resulting perimeter will increase rather than converge to a precise value. This is counterintuitive because if we do the same with a geometric shape, we know that the result will converge to the perimeter of the shape. For example, when applying the Richardson method to a circle, we can use a series of inscribed regular polygons. Each polygon is what we get by using a ruler equal to its side.

The circumference \( c \) of a circle with 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 \( \alpha = \frac{2\pi}{N} \), and the side is \( l = 2r \sin \frac{\alpha}{2} = 2r \sin \frac{\pi}{N} \). So the perimeter is \( c_N = N2r \sin \frac{\pi}{N} \).
6.3. MATHEMATICAL AND NATURAL FRACTALS: THE IMPOSSIBLE TASK OF MEASURING THE LENGTH OF COASTLINES

Figure 6.10: Hexagonal approximation of a circle. This figure was drawn using LibreLogo. The arrow pointing up shows the initial position and direction of the turtle.

Figure 6.11: Regular polygons approximating the circle. The first 30 regular polygons are shown, starting with the equilateral triangle. Logo code in listing A.13 at pag. 271.

The fact that polygons tend to approximate a circle as the number of sides increases is intuitive. In mathematical terms, we have the following:

\[
\lim_{N \to \infty} N2r \sin \frac{\pi}{N} = 2\pi r
\]  

(6.11)

But what about measuring a coastline? Why shouldn’t it be possible? Let’s apply Richardson’s method to the coast of Great Britain⁴:

⁴The plots and maps shown in this chapter were generated using a program written in R, a free software language for statistical computing and processing of large data sets. Knowledge of this language is not a goal of this book. We put the code (listing A.15) in appendix III (pag. 254), for those who are interested.
Figure 6.12: Measurement of the perimeter of the United Kingdom using a variety of rulers: 400, 200, 100, 50, 25 and 10 km wide. The length of the ruler and the estimated perimeter are shown in brackets.
While the approximation improves considerably, at first glance the perimeter increases steadily. Let’s plot these data points.
Figure 6.13: The red dots represent the estimated perimeter of Great Britain using rulers of 10, 25, 50, 100, 200, 400 km, from left to right, respectively. The green dots represent the same estimates of the perimeter of a circle with a circumference of 3000 km, using ruler values (sides of the inscribed regular polygon) of 3, 4, 6, 8, 10, 12, 15, 20, 30, 60, 120, 300 km, from left to right. The green horizontal line represents the exact value of the perimeter of the circle, 3000 km.
Surprisingly, the same procedure, applied to Great Britain or a circular island of about the same size, gives very different results! Not only that, but in the case of the circular island, the perimeter estimate tends asymptotically to the true value, while that of Great Britain seems to diverge, giving the impression that it follows more of a power law, such as $1/x^s$. Power laws can be linearized by plotting them on a log-log graph, where the slope of the resulting linear relationship is equal to the power coefficient $s$. 
CHAPTER 6. SIMULATION: FRACTALS GROWTH

Figure 6.14: Again, the red dots represent the estimated circumference of Great Britain using rulers of 10, 25, 50, 100, 200, 400 km, from left to right. The green dots represent the same estimates of the perimeter of a circle with a circumference of 3000 km, using ruler values (sides of the inscribed regular polygon) of 3, 4, 6, 8, 10, 12, 15, 20, 30, 60, 120, 300 km, from left to right. The green horizontal line represents the exact value of the perimeter of the circle, 3000 km.
The British data are fitted quite well by a straight line (red in the plot) whose slope turns out to be $s = -0.22$. We’ll return to this later.

So which is the right ruler? This was the finding of Richardson, who was investigating how countries with common borders often reported different border lengths. So far the empirical results.

Not many years later, Benoit Mandelbrot (1924-2010) laid the foundations of fractal geometry by reasoning about Richardson’s empirical findings in a famous paper published in Science: “How long is the coast of Britain? Statistical self-similarity and fractional dimension” [4]. Let’s go through his argument in a simplified way.

Let’s take a segment of length 1 and divide it into three smaller parts of length $1/3$ (Fig. 6.15). Each of the smaller parts is derivable from the whole by a similarity of the 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 into equal subsquares, each of side $1/3$, we get 9 subsquares. Again, if we decompose a cube of side 1 into equal sub-cubes, each of side $1/3$, we get 27 sub-cubes. What we find here is a relationship between the number of elements $N$ into which we divide a geometric 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-, two-, or three-dimensional for the line, the square, and the cube, respectively:

---

Richardson reported discrepancies in border measurements while he was investigating the causes of conflicts that led to war. He did a lot of work on peace and conflict studies [6], probably motivated by the fact that he lived through the terrible events of World War II.
Figure 6.15: The concept of self-similarity in relation to the dimension of simple geometric structures.
6.3. MATHEMATICAL AND NATURAL FRACTALS: THE IMPOSSIBLE TASK OF MEASURING THE LENGTH OF COASTLINES

<table>
<thead>
<tr>
<th>Object</th>
<th>Number of parts</th>
<th>Self-similarity ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>3</td>
<td>1/3</td>
</tr>
<tr>
<td>square</td>
<td>9</td>
<td>1/9(^{1/2})</td>
</tr>
<tr>
<td>cube</td>
<td>27</td>
<td>1/27(^{1/3})</td>
</tr>
</tbody>
</table>

Pretty familiar, so far. But now let’s do what mathematicians like to do: generalize. First, focus on the self-similarity relation:

\[ r(N) = \frac{1}{N^{1/D}} \]  

and then derive the dimension D:

\[ D = - \frac{\log N}{\log r(N)} \]  

Here we can make a leap of imagination. This last property of the set D means that it could also be evaluated for other geometric figures that can be exactly decomposed into \( N \) parts, so that each of the parts can be deduced from the whole by a similarity given by \( r(n) \). If such numbers exist, we can say that they have dimension \( D = - \frac{\log N}{\log r(N)} \), regardless of the fact that this may result in a non-integer value.

**Math: thinking outside the box**

Do such geometric figures exist? The ability to think outside the box is the essence of mathematical creativity, and this is a good example. The self-similarity relation 6.12 has provided a solid picture that depends on the convenient notion of dimension, which can be 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 in terms 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 solving this kind of problem. Our choices 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 created by performing automatic exercises whose meaning is not understood, in a competitive context and under time pressure: these are poisons that prevent the development of mathematical thinking and thinking in general. And this is also what neuroscience tells us, with the emphasis on the emotional nature of cognitive processes that we have already quoted [7]:

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 variable $D$ float, free to take non-integer values. When exploring outside the box, you have to get rid of the need to understand everything at once, all the time. Instead, you have to accept the idea that it will take some time to become familiar with new scenarios. In our example, the first step is to accept the idea that the value of $D$ may not be an integer, but this does not mean that you understand the meaning of this step right away. However, taking time and playing with the iterations of Koch’s process, which we will see right next, can eventually lead to the idea that a very curly line occupies a significant portion of the plane and that at the limit it is something 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”. [4] does not provide a complete
To show how there are such strange curves that are associated with a fractional dimensionality, Mandelbrot suggests studying 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:
Here we have decomposed a figure, i.e. a segment of length 1, into four elements \( N = 4 \), each reduced by a factor \( R(N) = 1/3 \). The process is repeated for each successive step:
As the process continues, the curve becomes more detailed. Beyond step 7, however, there is no significant improvement because the details are too dense for the resolution and the line is so twisted that it just appears as a thicker, less detailed line.
Thus, even if we cannot see it, we can imagine that the line “occupies” an increasing amount of space as the iterative process progresses, within the thick line we see at step 7.

That said, we are ready to give a meaning to the equation 6.13. In fact, if we substitute \( N = 4 \) and \( r(N) = \frac{1}{3} \), we get \( D = -\frac{\log 4}{\log 1/3} \), i.e.

\[
D = \frac{\log 4}{\log 3} = 1.26 \quad (6.14)
\]

Since we started with the familiar one-, two-, or three-dimensional concept, the idea of a fractional dimension still seems strange, but our reasoning about a line that is so twisted that it takes up “a lot of space” might fit the idea of a fractional dimension: not as thin as a line, not as thick as a full two-dimensional figure. Well, this is a typical feature of a fractal object: fractals have fractional dimensions. The Koch curve is a fractal whose dimension is \( D = 1.26 \).

Koch curve looks like a coast and that’s why we introduced it here. We started talking about the coast of Great Britain, now we will look at the
6.3. MATHEMATICAL AND NATURAL FRACTALS: THE IMPOSSIBLE TASK OF MEASURING THE LENGTH OF COASTLINES

island of Koch.

In order to build the island we have to browse the code we used to draw the previous examples.

```
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 6.15: Logo code for drawing the 7-step approximation of the Koch curve

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.

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 1, the recursive process stops and a segment is actually drawn. Thus, at step \(k\), the curve is composed only of 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 will be useful 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.
Now the fun begins: How long is the coastline of Koch Island? The mathematical derivation is easy. Let’s focus on the Koch curve, which is just 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, or $4/3$, since each of the four subsegments is $1/3$ long. At step 2, the process is repeated for each subsegment, so $4(1/3)(4/3) = (4/3)^2$. So on to step n,
6.3. MATHEMATICAL AND NATURAL FRACTALS: THE IMPOSSIBLE TASK OF MEASURING THE LENGTH OF COASTLINES

where the length will be \((4/3)^k\). So as \(n\) increases, the length will increase more and more. Actually, we have this

\[
\lim_{k \to +\infty} a^k = \infty, \quad (6.15)
\]

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, since the sum of infinite sets is also infinite:

\[
\lim_{k \to +\infty} 3(4/3)^k = \infty. \quad (6.16)
\]

So what? The perimeter of Koch island is infinite? Is something wrong? Let’s try to investigate further by calculating the area enclosed by this perimeter.
Figure 6.20: Area calculation of the Koch island at iteration 3. At the beginning the island is just an equilateral triangle (green). In the first step three smaller triangles (orange), scaled by $1/3$, are added. At the second step $3 \times 4$ 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 \times 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:

\[
A_{k+1} = A_k + n_k l_k^2 \frac{\sqrt{3}}{4} = A_k + l^2 \frac{\sqrt{3}}{12} \left( \frac{4^{k-1}}{9^{k-1}} \right).
\] (6.17)

We can expand this expression in this way:

\[
A_{k+1} = A_0 + l^2 \frac{\sqrt{3}}{12} \left( 1 + \frac{4}{9} + \frac{4^2}{9^2} + \cdots + \frac{4^{k-1}}{9^{k-1}} \right).
\] (6.18)

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

\[
\lim_{n \to \infty} \sum_n \left( \frac{4}{9} \right)^n = \frac{1}{1 - 4/9} = 9/5,
\] (6.19)

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

\[
A = A_0 + l^2 \frac{\sqrt{3}}{12} \frac{9}{5} = l^2 \frac{\sqrt{3}}{4} + l^2 \frac{\sqrt{3}}{12} \frac{9}{5} = l^2 \frac{2}{5} \sqrt{3}.
\] (6.20)

This confirms the strange result: a finite area is surrounded by an infinite perimeter. Obviously, we are faced with a new concept of perimeter. In fact, our idea of perimeter is deeply connected to the classical figures of geometry: squares, circles, and so on. But here we are dealing with a coastline, or something very similar, like the coastline of Koch’s imaginary island. And what is the essential difference? The details. The coasts have much more detail than the outlines of regular geometric figures. This result
encourages a reflection on the concepts of finite and infinite, an intuitive one, but useful for facing basic concepts of mathematical analysis in the future.

**Finite and infinite** Koch island is a mathematical abstraction: we can draw any intermediate step we want, but not the final result, which is a mathematical limit. The figure obtained at each step is composed of segments, albeit progressively smaller ones, and the total number of them is always finite, though increasing. Thus, the perimeter of each intermediate approximation is finite. The infinity comes from the mathematical limit process. This means that we cannot draw the true Koch curve, nor can we represent it in digital form. The same is true for irrational numbers like \( \pi \), which can never be represented as a digital number, but only as an approximation. For example, there’s no advantage to going beyond seven iterations when drawing the Koch curve, because the graphical resolution of whatever display device you have won’t allow you to see the finest details of the curve. Playing with the number of iterations, thinking about the amount of visible details, trying to obtain 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, for STEM or other disciplines.

Looking at the plots 6.13 and 6.14 on pages 228 and 230 respectively, it seems that the coastline of Britain has more in common with that of Koch island than with any classical geometric shape. Since Koch curve is composed of segments at each step, we can assimilate it to the rulers used by Richardson to calculate the lengths of coastlines. In this way, we can add the Koch island data to the previous graphical comparison between Britain and the circular island:
Figure 6.21: This is the same plot as Fig. 6.14 (page 230) but with Koch Island data added. Red points are relative to Britain, green points are relative to the circular island, and blue points are relative to Koch Island. The total size of Koch Island has been adjusted to roughly match that of Britain.
The slopes for the Britain and Koch curves are similar. We saw earlier that the estimate derived from the statistical fit of the Britain curve was \( s = -0.22 \). What is the slope of the Koch curve? This can be estimated by direct calculation. For each step \( k \), i.e., for each point plotted for the Koch curve, we have the length of the ruler \( l_k = (1/3)^k \) on the abscissa and the length of the perimeter \( L_k = (4/3)^k \) on the ordinate. Now let’s derive \( L_k \) as a function of \( l_k \). From these definitions we can write

\[
L_k = \left( \frac{4}{3} \right)^k = 4^k \left( \frac{1}{3} \right)^k = 4^k l_k. \tag{6.21}
\]

Using the identity

\[
4 = 3^{\log 4 / \log 3} \tag{6.22}
\]

we have

\[
L_k = \left( 3^{\log 4 / \log 3} \right)^k l_k. \tag{6.23}
\]

Since \( \frac{\log 4}{\log 3} \) is equal to the fractal dimension \( D \) of the Koch curve (equation 6.14), we obtain

\[
L_k = 3^D k l_k, \tag{6.24}
\]

which can be written as

\[
L_k = l_{k}^{1-D}. \tag{6.25}
\]

This result is interesting because it establishes the relationship between the fractal dimension \( D \) of the perimeters and the slopes of the corresponding 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 a curve is with respect to regular geometric shapes: as long as the details increase, the slope \( s \) also increases (in absolute value) 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, the previous plots suggest that Britain and Koch island have something in common. However, we must keep in mind that the two lines are very different in nature: Koch’s approximation is obtained by a well-defined mathematical process of iteratively reproducing
a basic shape and applying a reductive scale factor at each step, which is the
definition of a fractal. Instead, the coast of Britain (or any real island) is
shaped over time by a complex combination of many factors. Therefore, in
this case, the details are statistical in nature. Thus, natural coastlines (and
many other natural forms) have a fractal nature, perhaps a wide range of
scales, but they are not fractal in the mathematical sense.

Uncertainty: A crucial “discovery” in the science of the 20th cen-
tury Since the advent of mass education, disciplines have been taught
differently. And even today, little or no effort is made to look at the big
picture and the connections between different fields. This means that
the intersections, which are often where knowledge grows most rapidly,
are largely excluded from the average education of young people. As
a result, a crucial achievement of the 20th century is almost com-
pletely missing from the scientific picture: the intimate interweaving
of certainty and uncertainty that characterizes all areas of science.

In chapter 1, section 1.2, we have highlighted this problem with the
words of Morin when he talks about the “School of Mourning”. We
have mentioned several areas of science where unpredictability plays
a critical role. But none or almost none of this comes into the school.
It can be argued that these topics require too complicated treatment.
Apart from the fact that appropriate and instructive narrative solutions
can almost always be found, there are actually simple examples. Mea-
suring the coastline of an island is an example of how trying to solve
a seemingly trivial problem can suddenly take you into the realm of
uncertainty. It is an interesting example because it can be tackled with
mathematical skills within the reach of a high school student. It is
also interesting because it involves various areas and investigations,
some of which are only mentioned in this description. An interesting
diversion could be made at the beginning: why did Richardson at some
point come to measure the circumference 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 emo-
Richardson was not only an accomplished scientist, but also a well-known pacifist, and the tragic events of World War II led him to explore the causes of many disputes between nations, such as coastlines and border lengths. These kinds of considerations might lead to an exploration of historical and social context. Then, having demonstrated Richardson’s method of measurement, we posed the problem of doing the same with a circular region. This is a geometry problem that students could be asked to solve. For example, using Logo (hints could be given based on the program listing 271 on page 271 that we used to reproduce Fig. 6.10). If the students are already familiar with the concept of mathematical limit, then the limit of the perimeter of a regular polygon as the number of sides tends to infinity can be investigated (equation 6.11). When comparing the dependence of the perimeter on the ruler in the case of Britain and in the case of the circular island, there is an opportunity to discuss in detail the different nature of the plotted data: estimates calculated from experimental data versus theoretical values. Data in a graph can be of different types: students must learn to recognize this, otherwise we will end up with citizens who cannot understand a graph in a newspaper. Then, when we encounter 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 chance to revise the logarithmic function. Guided by Benoit Mandelbrot, the father of fractals, we get a vivid example of what it means to think outside the box, typical of mathematical thinking. And by tinkering with the logo approximations of the Koch curve, we can explore the recursive and autosimilar nature of fractals. Determining the length of the Koch curve involves the use of geometric series approximations, but most importantly, it allows us to think deeply about the relationship between infinity and finity — important insights for all future STEM students.

In conclusion, measuring coastlines is an example of a seemingly trivial exploration that, when presented in a meaningful way in school, reveals unexpected aspects about the relationship between certainty and uncertainty.
Bibliography


Appendix A

Appendices
A.1 Index of powerful ideas and relevant reflections

This is an index of the grey text boxes we have added to point out powerful mathematical ideas, concepts or approaches that are evoked by the exercises. Concepts that do not need to be explicitly described to students, but that teachers should be aware of in order to understand the potential of the exercises and their key points.

Pag. 64 — Isomorphism

Pag. 67 — Break the problem into smaller parts: Divide et impera

Pag. 69 — State of a system

Pag. 72 — The process of constructing scientific knowledge: the problem of epistemology

Pag. 76 — Encapsulating functionality in new commands: modular thinking

Pag. 79 — Thinking modular again when drawing the house

Pag. 82 — The door to algebra

Pag. 87 — Syntonic learning described through in a dialogue

Pag. 88 — Differential equations

Pag. 95 — Linear and exponential growth

Pag. 96 — Self-similarity → fractals

Pag. 104 — Concept of “law”

Pag. 106 — The concept of integration

Pag. 107 — The limits of the machine (and of theory): how can the execution of a program unintentionally become a never-ending story?

Pag. 108 — Trying to find a stop condition for a (simple) program
A.1. INDEX OF POWERFUL IDEAS AND RELEVANT REFLECTIONS

Pag. 115 — Successive approximations
Pag. 124 — Physics — Initial conditions
Pag. 125 — Physics — Computational vs algebraic approach
Pag. 154 — Randomness in science
Pag. 164 — Feedback in nature
Pag. 172 — Scope of a theory
Pag. 179 — Scalar and vector fields
Pag. 183 — Multitasking: the computer juggling trick
Pag. 244 — Fractals: infinity in the finite → towards calculus
Pag. 247 — Uncertainty: an essential ingredient in 20th century science
A.2 Program listings

The full commented listings of many of the programs used in the book are collected in this appendix. Only the relevant chunks of code are included in the text to make it easier to read. The programs can also be downloaded from the MOOC and are ready to run in the LibreLogo or TigerYjthon environments.

Modeling smell

This list refers to the simple olfactory model described in section 5.3. The turtle is confined to a circular garden and its movement is disturbed by a degree of randomness.

```python
# Turtle smelling model
# confined to a 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 turn it green
def circle(r):
    c = r * 2 * pi
    s = c / 360

Listing A.1: The turtle reaches the salad by smell, despite random distractions.
A.2. PROGRAM LISTINGS

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

# Take one step and turn if the arrival
# point is inside the green area.
# Otherwise keep turning right until
# 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 that given position and direction
# is within the 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 A.2: The turtle reaches the salad by smell, despite distractions.
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 dist(p1,p2):
    return sqrt((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)

# Setting up the Python Turtle environment
setPlaygroundSize(800, 800)
makeTurtle()
hideTurtle()

r = 300
sPos = (-150, 180)
start = (150, -100)

circle(r)
heading(0)
salad(sPos)

l1 = 1
l2 = 5
a1 = -90
a2 = 90
data = (l1, l2, a1, a2)

# Place the Turtle and start the steps loop
n = 1000
showTurtle()
setColor("darkgreen")
setPenColor("darkgreen")
setPos(start)
p1 = getPos()
p2 = getPos()
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
A.2.  PROGRAM LISTINGS

Facing light

This is the simplest of our sighting models, where the turtle simply tries to keep its bearing constant in relation to a light source. (section 5.4).
# Sighted turtle keeping bearing
towards(px, py) and bearing(px, py) tested in towards-test.
pv
# 14.3.2020

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

def r2d(ar):
    return ar / pi * 180

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))
    elif dy < 0:
        return 180 + r2d(atan(dx/dy))

def bearing(px, py):
    h = heading()
    if h > 0:
        return towards(px, py) - h
    else:
        return 360 - (towards(px, py) - h)

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

# Moves the turtle from a given position
# and heading, while maintaining a
# fixed bearing
def keepBearing(px, py):
    i = 0

Listing A.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 target (px, py)
px = 50
py = 50
setPenColor("red")
setPos(px, py)
dot(10)
setPenColor("darkgreen")

# Set initial turtle 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 A.5: The turtle moves towards a light source by maintaining a constant bearing.
Two-eye vision

In this example, we will try to model two eye vision by simply working on the fields of view. (section 5.4).
# Turtle two-eye model
# towards(px, py) and bearing(px, py) tested in towards-test.py
# 26.2.2020

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

# turns radians 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 the bearing angle of the direction
# to point (px, py) with respect to the
# turtle's heading
def bearing(px, py):
    h = heading() % 360
    t = towards(px, py)
    b = abs(t - h)
    return b

# Check that the object is visible to the left eye
def leftEye(b):
    if b > 350:
        return True
    if b < 60:
        return True
    return False

Listing A.6: The turtle moves toward a light source by keeping it within the eyes field of view.
# check if seen by right eye

def rightEye(b):
    if b > 300:
        return True
    if b < 10:
        return True
    return False

# heading for light source

def headFor(px, py):
    while True:
        b = bearing(px, py)
        if leftEye(b) or rightEye(b):
            forward(10)
        else:
            r = randint(1,359)
            left(r)

setPlaygroundSize(1000, 1000)
maketurtle()
hideturtle()

# Drawing reference system

setPos(0, -200)
moveto(0, 200)
setPos(-200, 0)
moveto(200, 0)

# (px, py) light source
# (xcor, ycor) turtle initial location
# h turtle initial heading

px = 100
py = -100
setPos(px, py)
dot(10)

xcor = -100
ycor = -200
a = 315
setPos(xcor, ycor)
heading(a)

showturtle()

headFor(px, py)

Listing A.7: The turtle moves toward a light source by keeping it within the eyes field of view.
Two-eyes vision with intensity perception

Here we model two eyes seeing and try to keep the intensity detected by each eye in balance. (section 5.4).
# Turtle two-eye model with intensity

# towards(px, py) and bearing(px, py) tested in towards-test.

# At #83 a drop of randomness in this version...

# 27.2.2020

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

# turns radians 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 A.8: The turtle moves toward a light source by keeping left and right intensities in balance.
# check if seen by right eye
def rightEye(b):
    if b > 350:
        return True
    if b < 60:
        return True
    return False

# gives distance from point (px, py)
def dist(px, py):
    p = getPos()
    return sqrt((px - p[0])**2 + (py - p[1])**2)

# 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)

# heading for light source
def headBySight(px, py):
    i = 0
    while True:
        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:
                left(randint(-180,180))

setPlaygroundSize(1000, 1000)
maketurtle()
# (px, py) light source
# (xcor, ycor) turtle initial location
# h turtle initial heading
px = 100
py = -200
setPos(px, py)
dot(10)
xcor = 100
ycor = 200
a = -90
setPos(xcor, ycor)
heading(a)
hideTurtle()

print("Start!")
headBySight(px, py)

Listing A.10: The turtle moves toward a light source by keeping left and right intensities in balance.
Managing two turtles at the same time in LibreLogo

LibreLogo is a monotasking environment, i.e. you can only control one turtle at a time. It is designed to generate a graphic, not to model dynamic behaviour. Here we enforce these limitations by implementing the basic mechanism used by computers to achieve a multitasking effect. (section 5.5)
TO SAVEDRED
   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 A.11: Managing two turtle simultaneously in LibreLogo
TO EXECBOTH
    REPEAT 50 [
        RESTGREEN
        EXECGREEN
        SAVEGREEN
        RESTRED
        EXECRED
        SAVERED
    ]
END

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

PENCOLOR 'black'
FILLCOLOR 'RED'
HEADING ANY
SAVEGREEN
HEADING ANY
SAVERED
CIRCLE 4
EXECBOTH
PRINT 'Ok!'
Approximating the circle perimeter

This is the LibreLogo code used to create the first figures in the 6.3 section to show how a circle can be approximated by regular polygons.
R = 150 ; radius of circle
C = 2*PI*R ; circumference

CLEARSCREEN
HIDETURTLE
PENSIZE 0.5

REPEAT 30 [  
  HOME  
PENUP  
  LEFT 90 FORWARD 150 RIGHT 90 ; transpose the home 150 pt left  
PENDOWN  
  N = REPCOUNT+2 ; REPCOUNT iteration counter  
  A = 2*PI/N  
  A2 = A/2  
  L = 2*R*SIN(A2) ; polygon side  
  D = PI/2-(PI-A)/2  
  DD = D/PI*180 ; turtle deviation angle in degrees  
  RIGHT DD ; first half deviation  
  REPEAT N [ ; drawing N sides  
    FORWARD L  
    RIGHT DD*2  
  ]  
]  
HOME ; now draw the `'true'` circumference
PENUP  
LEFT 90 FORWARD 150 RIGHT 90  
PENDOWN  
N = 360  
A = 2*PI/N  
A2 = A/2  
L = 2*R*SIN(A2)  
D = PI/2-(PI-A)/2  
DD = D/PI*180  
RIGHT DD  
REPEAT N [ ; drawing N sides  
  FORWARD L  
  RIGHT DD*2  
]  

Listing A.13: Circle radius and number of polygons sides can be adjusted as desired
Fractal L-system stick tree

Script for drawing a fractal stick tree with the L-system formal language (section 6.1).
# 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 A.14: Stick tree "plant" with L-system.
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 calculate the perimeters of Great Britain and other imaginary geometric islands (section 6.3). R is a free software language for statistical computing and large data processing. Knowledge of this language is not one of the aims of this book. We show the code here for reference only.
# Original code: Spatial Data Science: The length of a coastline
# https://r-spatial.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.

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 A.15: Remiking of “Spatial Data Science: The length of a coastline”
# Britain has an area of about 220,000 km**2.

par(mai=rep(0,4))
plot(b)

# Now the function to go around the coast with a ruler (yardstick)
# of a certain length

measure_with_ruler <- function(pols, length, lonlat=FALSE) {
  # some sanity checking
  stopifnot(inherits(pols, 'SpatialPolygons'))
  stopifnot(length(pols) == 1)
  # get the coordinates of the polygon
  g <- geom(pols)[, c('x', 'y')]
  nr <- nrow(g)
  # we start at the first point
  pts <- 1
  newpt <- 1
  while(TRUE) {
    # start here
    p <- newpt
    # order the points
    j <- p:(p+nr-1)
    j[j > nr] <- j[j > nr] - nr
    gg <- g[j,]
    # compute distances
    pd <- pointDistance(gg[1,], gg, lonlat)
    # get the first point that is past the end of the ruler
    # this is precise enough for our high resolution coastline
    i <- which(pd > length)[1]
    if (is.na(i)) {
      stop('Ruler is longer than the maximum distance found')
    }
    # get the record number for new point in the original order
    newpt <- i + p
    # stop if past the last point
    if (newpt >= nr) break
    pts <- c(pts, newpt)
  }
  # add the last (incomplete) stick.
  pts <- c(pts, 1)
  # return the locations
  g[pts, ]
}

# Let's call the function with rulers of different lengths.

y <- list()
rulers <- c(10, 25, 50, 100, 200, 400) # km
# rulers <- c(50, 100, 250) # km
for (i in 1:length(rulers)) {
  y[[i]] <- measure_with_ruler(b, rulers[i]*1000)
}
# Object y is a list of matrices containing the locations where the ruler touched the coast. We can plot these on top of a map of Britain.

```r
plot -maps.png
```n

```r
par(mfrow=c(2,3), mai=rep(0,4))
```

```r
# for (i in 1:length(y)) {
for (i in length(y):1) {
  p <- y[[i]]
  lines(p, col='lightgray', lwd=3)
  lines(p, col='black', lwd=1)
  points(p, pch=20, col='blue', cex=2)
  bar <- rbind(cbind(525000, 900000), cbind(525000, 900000-rulers[i]*1000))
  lines(bar, lwd=2)
  points(bar, pch=20, cex=1.5)
  text(525000, mean(bar[,2]), paste(rulers[i], ' km '), cex=1.5)
  # text(525000, bar[2,2]-50000, paste0('(', nrow(p), '*' rulers[i], ' km)'), cex=1.25)
}
```

```r
dev.off()
```

# Here is the fractal (log-log) plot. Note how the axes are on the log scale, but that we used the non-transformed values for the labels.

```r
# number of times a ruler was used
n <- sapply(y, nrow)
```

# Plot of perimeter lengths vs ruler lengths

```r
plot(rulers, n, type='n', xlim=c(5,500), ylim=c(2000,6000), axes=FALSE, xaxs="i", yaxs="i", xlab='Ruler length (km)', ylab='Approximated perimeter (km)')
```

```r
xtics <- c(0,100,200,300,400,500,600)
ytics <- c(1000,2000,3000,4000,5000,6000)
axis(1, at=xtics, labels=xtics)
axis(2, at=ytics, labels=ytics, las=2)
```

# linear regression line

```r
m <- lm(log(n) ~ log(rulers))
```

```r
abline(m, lwd=1, col='red')
```
# Log-log plot of perimeter lengths vs ruler lengths
# Britain and circular isle
# plot-log.png

gn(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/POWERFUL-IDEAS/Turtle-book/coastlines/plot-log.png")

# set up empty plot
plot(log(rulers), log(n*rulers), type='n', xlim=c(1,7), ylim=c(7.5,9), axes=FALSE, xaxs="i",yaxs="i", xlab='Ruler length (km)', ylab='Approximated perimeter (km)')

xtics <- c(5,10,25,50,100,200,300,400,500,600,700,800,900,1000)
ytics <- c(2000,2800,3600,4400,5200,6000)
axis(1, at=log(xtics), labels=xtics)
axis(2, at=log(ytics), labels=ytics, las=2)

# linear regression line
m <- lm(log(n*rulers)˜log(rulers))
abline(m, lwd=1, col='red')

# add observations
points(log(rulers), log(n*rulers), pch=20, cex=2, col='red')
poly_n <- c(300,120,60,30,20,15,12,10,8,6,4,3)
c_rulers <- 3000/pi*sin(pi/poly_n)
points(log(c_rulers), log(poly_n*c_rulers), pch=20, cex=2, col='green')
abline(h=log(3000), lwd=1, col="green")

dev.off()

# Log-log plot of perimeter lengths vs ruler lengths
# Britain, circular isle and Koch data
# plot-log-koch.png

gn(filename="/Users/Andreas Formiconi/arf/Didattica/PROJECTS/POWERFUL-IDEAS/Turtle-book/coastlines/plot-log-koch.png")

# set up empty plot
plot(log(rulers), log(n*rulers), type='n', xlim=c(1,7), ylim=c(7.5,9), axes=FALSE, xaxs="i",yaxs="i", xlab='Ruler length (km)', ylab='Approximated perimeter (km)')

xtics <- c(5,10,25,50,100,200,300,400,500,600,700,800,900,1000)
ytics <- c(2000,2800,3600,4400,5200,6000)
axis(1, at=log(xtics), labels=xtics)
axis(2, at=log(ytics), labels=ytics, las=2)

# linear regression line
m_b <- lm(log(n*rulers)˜log(rulers))
abline(m_b, lwd=1, col='red')

# add observations
points(log(rulers), log(n*rulers), pch=20, cex=2, col='red')
poly_n <- c(300,120,60,30,20,15,12,10,8,6,4,3)
c_rulers <- 3000/pi*sin(pi/poly_n)
points(log(c_rulers), log(poly_n*c_rulers), pch=20, cex=2, col='green')
abline(h=log(3000), lwd=1, col="green")

dev.off()