# Chapter 2

# Iterated growth {chap:ca}

The bewildering array of complex shapes and forms encountered in the natural world, from tiny crystals to living organisms, often results from a growth process driven by the repeated action of simple "rules". In this chapter we examine this general idea in the specific context of *cellular automata*, (hereafter abbreviated CA), which are arguably the simplest type of computer programs conceivable. Yet they can sometimes exhibit behaviors that, by any standard, can only be described as extremely complex.

Cellular automata also exemplify, in a straightforward computational context, a recurring theme that runs through all instances of natural complexity to be encountered in this book: simple rules can produce complex global "patterns" that cannot be inferred or predicted even when a complete, *a priori* knowledge of these rules is at hand.

## 2.1 Cellular automata in one spatial dimension {sec:CA1D}

Imagine a one-dimensional array of contiguous cells, sequentially numbered by an index $j$ starting at $j = 0$ for the leftmost cell. Each cell can be "painted" either white or black, with the rule for updating the $j^{\text{th}}$ cell depending only on its current color and those of the two neighbouring cells at positions $j-1$ and $j+1$. Consider now the following graphical procedure: at each iteration, the CA looks like a linear array of cells that are either black or white. If we now stack successive snapshots of this row of cells below one another, we obtain a two dimensional spatiotemporal picture of the CA's evolution, in the form of a (black & white) pixellized image such as formed on the CCD of a digital camera, except that each successive row of pixels captures an iteration of the growth process, rather than the vertical dimension of a truly two-dimensional image.

The simple question is then: starting from some given initial pattern of white and black cells, how will the array evolve in response to the repeated application of the update rule to every cell of the array ? As a first example, consider the following very simple CA rule:

- **First Rule:** Cell $j$ becomes (or stays) black if one or more of the neighbour triad $[j-1, j, j+1]$ is black; otherwise it becomes (or stays) white.

The top panel of Figure 2.1 shows the first 20 iterations of a CA abiding to this rule, starting from a single black cell in the middle of an otherwise all-white array. On this spatiotemporal diagram, the sideways growth of the CA translates into a black triangular shape expanding by one cell per iteration from the single initial black cell. Starting again from a single black cell but adopting instead the following, equally simple second rule:

- **Second Rule:** Cell $j$ becomes (or stays) black if either or both of its neighbours $j-1$ and $j+1$ are black; otherwise it stays (or becomes) white.

yields the pattern plotted on the bottom panel Figure 2.1. The global shape is triangular again, but now the interior is a checkerboard pattern of white and black cell alternating regularly in

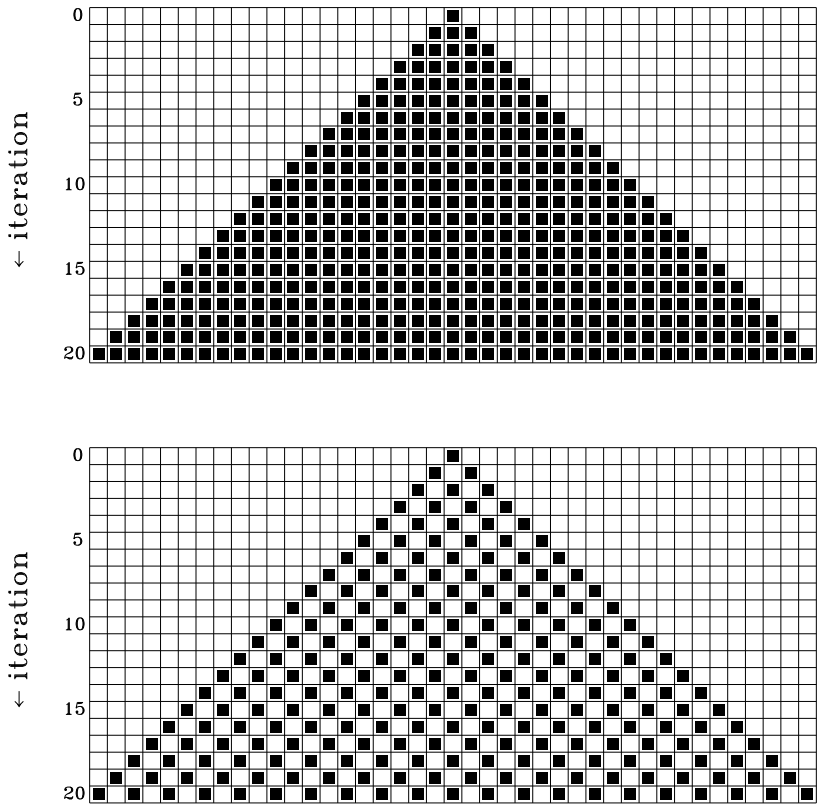


Figure 2.1: The first 20 iterations of a 1D cellular automaton abiding to the first (top) and second (bottom) update rules introduced in the text, in both cases starting from a single black cell at the center of the array (iteration 0, on top). The horizontal direction is the "spatial" dimension of the 1D CA, and time/iteration runs downwards. {`fig:ca1D`}

both the spatial and temporal dimensions. With just a bit of thinking, these two patterns could certainly have been expected on the basis of the above two rules.

But is it always the case that simple CA rules lead to such simple, predictable spatiotemporal patterns? Consider now the following update rule:

- **Third Rule:** Cell $j$ becomes (or stays) black if one and only one of its two neighbours $j-1$ or $j+1$ is black; otherwise it stays (or becomes) white.

This is again a pretty simple update rule, certainly as simple as our first and second rules. The top panel of Figure 2.2 shows the pattern resulting from the application of this rule to the same initial condition as before, namely a single black cell at array center. The globally triangular shape of the structure is again there, but the pattern materializing within the structure is no longer so simple. Many white cells remain, clustered in inverted triangles of varing sizes but organized in an ordered fashion. This occurs because our third rule implies that once the cells have reached an alternating pattern of white/black across the full width of the growing structure, as on iterations 3, 7, and 15 here, the rule forces the CA to reverts to all-white at the next iteration, leaving only two black cells at its right and left extremities. Right/left

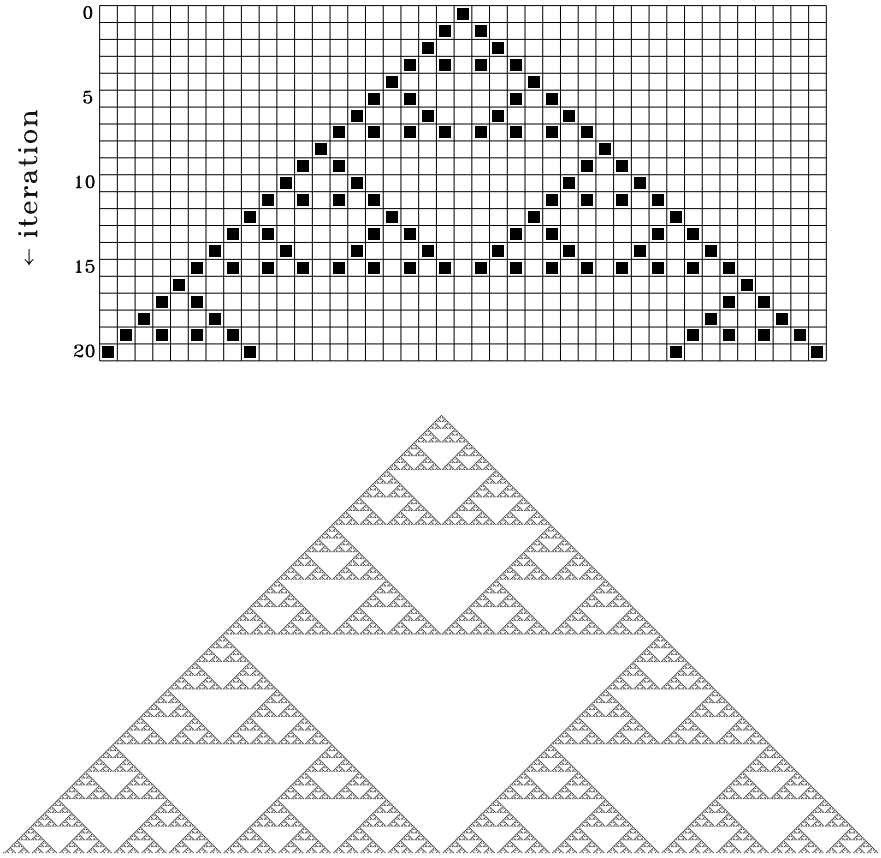


Figure 2.2: The top panel is identical in format to Figure 2.1, but shows now the structure produced after 20 iterations by the third CA update rules introduced in the text. The bottom panel shows the same CA, now pushed to 512 iterations, with cell boundaries removed for clarity. {fig:ca1D2a}

symmetrical growth of new black cells then resumes from these points, replicating at each end the triangular fanning pattern produced initially from the single original black cell. Growth thus proceeds as a sequence of successive branching, fanning out, and extinction in the interior.

The bottom panel of Figure 2.2 displays 512 iterations of the same CA as on the top panel, with the cell boundaries now omitted. Comparing the top and bottom panels highlights the fundamental difference between the "microscopic" and "macroscopic" views of the structure. On the scale at which the CA is operating, namely triad of neighbouring cells, a cell is either white or black. On this microscopic scale the more conspicuous pattern to be noticed on the top panel of Fig. 2.2 is that blacks cells always have white cells for neighbours at right, left, up and down, but no so such "checkerboard" constraint applies to white cells (unlike on the bottom panel of Fig. 2.1). At the macroscopic level, on the other hand, the immediate perception is one of recursively nested white triangles. In fact, the macroscopic triangular structure can be considered as being made from three scaled-down copies of itself, touching at their vertices; each of these three copies, in turn, is made up of three scaled-down copies of itself, and so on

down to the "microscopic" scale of the individual cells[1].

This type of recursive nesting is a hallmark of *self-similarity*, and flags the structure as a *fractal*. For now you may just think of this concept as capturing the fact that successive zooms on a small part of a structure always reveal the same geometrical pattern, like with the bifurcation diagram for the logistic map encountered in chapter 1 (cf. Fig. 1.2). More generally, self-similarity is a characteristic feature of many complex systems, and will be encountered again and again throughout this book.

Consider finally a last, fourth CA rule, hardly more complicated than our third:

- **Fourth Rule:** Cell $j$ becomes (or stays) black if one and only one of the triad $[j-1, j, j+1]$ is black, or if only $j$ and $j+1$ are black; otherwise it stays (or becomes) white.

This rule differs from the previous three in that it now embodies a directional bias, being asymmetrical with respect to the central cell $j$: a white cell at $j-1$ and blacks cells at $j$ and $j+1$ will leave $j$ black at the next iteration, but the mirror configuration, $j-1$ and $j$ black and $j+1$ white, will turn $j$ white. The top panel of Figure 2.3 shows the first 20 iterations of this CA, as usual (by now) starting from a single black cell. The expected symmetrical triangular shape is there again, but now the interior pattern lacks mirror symmetry, not surprisingly so perhaps, considering that our fourth rule itself lacks right/left symmetry. But there is more to it than that. Upon closer examination one also realizes that the left side of the structure shows some regularities, whereas the right half appears not to. This impression is spectacularly confirmed upon pushing the CA to a much larger number of iterations (bottom panel). On the right the pattern appears globally random. As with our third rule, inverted white triangles of varying sizes are generated in the course of the evolution, but their spatial distribution is quite irregular and does not abide to any obvious recursive nesting pattern. On the left, in contrast, the pattern is far more regular, with well-defined structures of varying periodicities recurring along diagonal lines running parallel to the left boundary of the structure.

With eight possible three-cell permutations of two possible states (white/black, or 0/1, or whatever) and evolution rules based on three contiguous cells (the cell itself plus its right and left neighbours), there exist 256 possible distinct evolutionary rules[2]. Even if always starting from a single active cell, as on Fig. 2.1, these various rules lead to a staggering array of patterns, going from triangular wedges, repeating checkerboard or stripe patterns, simple or not-so-simple patterns propagating at various angles, nested patterns (as on Fig. 2.2), mixtures of order and disorder (as on Fig. 2.3), all the way to complete randomness. You get to explore some of these in one of the suggested computational exercises proposed at the end of this chapter.

These 256 1D CA rules can be divided fairly unambiguously into four classes, according to general properties of the end state they lead to that are independent of the initial condition[3].

- Class I CAs evolve to a stationary state; our First rule (Fig. 2.1) offers an example, although keep in mind that the stationary state need not be all-black or all-white.

[1] This structure belongs to a class of geomerical objects known as *Sierpinski triangles*. It can be constructed by a number of alternate geometrical procedures. The simplest consists in drawing a first triangle, then partitioning it into 4 smaller triangles by tracing three straight line segments joining the edge centers, then repeating this process for the three outer triangles so produced, and so on. The CA, in contrast, generates the same macroscopic structure via a directional iterative spatiotemporal growth process.

[2] Describing CA rules in words, as done so far, can rapidly becomes quite awkward. A much superior and compact description can be made using a 8-bit binary string, with each bit giving the update (black= 0 and white= 1, say) associated with one of the eight possible permutations of black/white on three cells. As a bonus, interpreting each such string as a binary coding of an integer yields a number ranging from 0 to 255, which then uniquely labels each possible rule. Chapter 3 of the book by Wolfram cited in the bibliography describes this procedure in detail. Under this numbering scheme the four rules introduced above are numbered 254, 250, 90, and 30, respectively.

[3] The classification is best established through the use of a random initial condition where every cell in the initial state is randomly assigned white or black with equal probability. In such a situation it is also necessary to introduce periodic boundary conditions, as if the 1D CA were in fact defined on a closed ring: the last, rightmost cell acts as the left neighbour of the first, leftmost cell; and vice-versa.

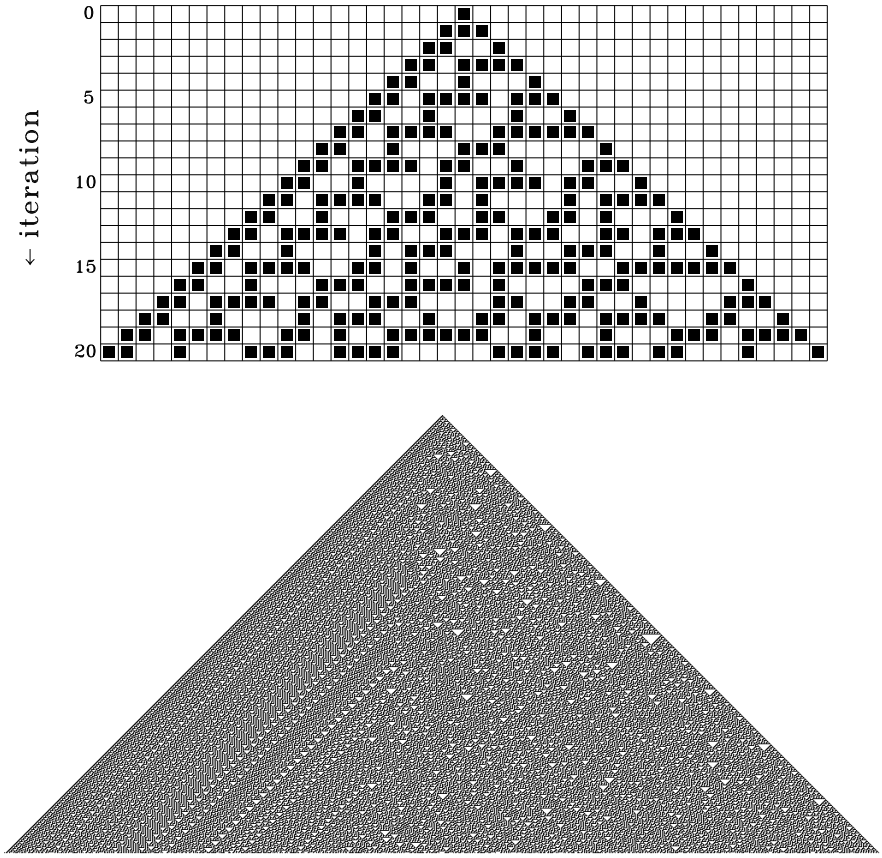


Figure 2.3: Identical in format to Figure 2.2, but now for the fourth CA update rule introduced in the text. {fig:ca1D2b}

- Class II CAs evolve into a periodic configuration, where each cell repeatedly cycles through the same set of states (which may differ from one cell to the next). Our second rule is a particularly simple examplar of this class.

- Class III CAs evolve into a non-periodic configuration. Even though Figure 2.2 looks quite regular, our third rule belongs in fact to this class, as you get to verify in one of the computational exercises suggested at the end of this chapter.

- Class IV CAs collect everything else that does not fit into the first three classes; they are also, in some sense, the more interesting rules, in that they yield configurations that are neither stationary, periodic, nor completely aperiodic.

Figure 2.4 gives a minimal source code in the Python programming language for running the 1D two-state CA of this section, with a value zero for white cells and one for black cells. The CA evolution is stored in the 2D array `image`, the first dimension corresponding to time/iteration, and the second to the spatial extent of the CA (line 8). This code uses a single black cell at lattice center for initial condition (line 9), and is set up to run the third rule introduced above. The condition "one and only one of the nearest neighbours being black" is evaluated by summing the corresponding numerical values of the cells (line 14); if the sum is one, then node $j$ turns black

```
1  # 1D 2-STATES CELLULAR AUTOMATON
2  import numpy as np
3  import matplotlib.pyplot as plt
4  #---------------------------------------------------------------------
5  N=129                                        # Size of 1D CA
6  n_iter=64                                    # Number of iterations
7  #---------------------------------------------------------------------
8  image=np.zeros([n_iter,N],dtype='int')       # Initialize lattice to white
9  image[0,N/2]=1                               # But set central node to black
10
11 for iterate in range(1,n_iter):              # Iteration loop
12
13     for j in range(1,N-1):                   # Lattice loop
14         if image[iterate-1,j+1]+image[iterate-1,j-1] == 1:  # Third rule
15             image[iterate,j]=1               # Turn node black
16     # End of lattice loop
17
18     image[iterate,0]=image[iterate,N-2]      # Enforce periodicity
19     image[iterate,N-1]=image[iterate,1]
20 # End of iteration loop
21
22 plt.imshow(image,interpolation="nearest")    # Display structure
23 plt.show()
24 # END
25
```

Figure 2.4: A minimal source code in the Python programming language for the one-dimensional CA introduced in this section. As with all Python codes given in this book, the Python `numpy` library is used. The `numpy` function `np.zeros()` generates an array of whatever size given to it as argument and fills it with values zero; here a 2D array of size `n_iter`×`N`, through the argument `[n_iter,N]`. The code given here operates according to the third CA rule introduced in the text. {`code:ca1d`}

at the next iteration (value "1" in array `image`, line 15), otherwise remaines white (initialized value "0"). Periodicity is enforced in the spatial direction (lines 18–19; see §D.2 for further detail on this). Upon completion of the CA's evolution over the preset number of iterations, the structure produced is displayed using the `imshow()` function from the `matplotlib.pyplot` graphics library (lines 22–23).

{`sec:CA2D`}

## 2.2 Cellular automata in two spatial dimensions

Cellular automata are readily generalized to two (or more) spatial dimensions, but the various possible lattice geometries open yet another dimension (figuratively speaking!) to the specification of the CA and its update rules. It will prove useful to adopt an alternate but entirely equivalent formulation of CA based on a *lattice* of interconnected *nodes*, conceptually equivalent to the center of cells on Fig. 2.1–2.3. Figure 2.5 illustrates the idea, for different types of two-dimensional lattices using different *connectivities* between neighbouring nodes.

On Cartesian lattices in two spatial dimensions (panels A and B), connectivity typically involves either only the 4 nearest neighbours (in red) at right/left/top/down of a given node (in black), or also the four neighbours along the two diagonals (panel B)[4]. Anisotropic connec-

[4]The top/bottom/right/left 4-neighbour connectivity on a Cartesian lattice is sometimes referred to the *von*

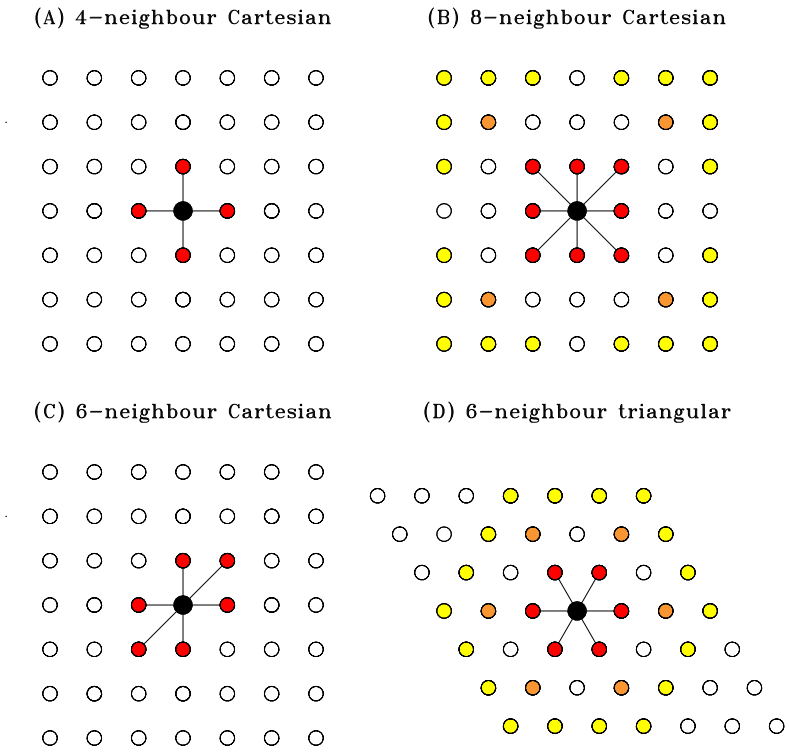


Figure 2.5: Example of some lattices and associated nodal connectivities, as indicated by the line segments connecting the central black node to its nearest-neighours in red. The meaning of orange- and yellow-colored nodes in (B) and (D) will be elucidated further below. {`fig:cahexdemo`}

tivities, as on panel C, can be reinterpreted as changes in lattice geometry; upon introducing a horizontal displacement of half an internodal distance per row and compressing vertically by a factor $\sin(\pi/3) \simeq 0.866$, as shown on panel D, one obtains a regular triangular lattice with 6-neighbour connectivity. From the point of view of CA evolutionary rules, the two lattices in C and D are topologically and operationally equivalent. What is interesting in practice is that whether triangular or cartesian, these lattices can all be conveniently stored as two-dimensional arrays in the computer's memory, and the "true" geometry becomes set by the assumed connectivity.

We first restrict ourselves to the following very simple 2D CA rule:

- A node becomes active if one and only one of its neighbours nodes is also active.

Note that such a rule has no directional bias other than that imposed by the lattice geometry and connectivity: any one active node will do. However, a noteworthy difference with the 1D CA rules considered previously is that here, once activated a node remains activated throughout the remainder of the iterative process. Nonetheless, as far as rules go, this is probably about as simple as it could get in this context. Figure 2.6 lists a minimalistic Python source code for this automaton, defined on a triangular lattice with 6-neighbour connectivity. Note the following:

*Neumann neighbourhood*, and the 8-neighbour connectivity as the *Moore neighbourhood*. See §D.1 for more on these matters.

```
1  # 2D 2-STATES CELULAR AUTOMATON ON TRIANGULAR LATTICE
2  import numpy as np
3  import matplotlib.pyplot as plt
4  #-----------------------------------------------------------------------
5  N=24                                        # Size of 2D CA
6  n_iter=10                                   # Number of iterations
7  n_neighbour=6                               # Number of connected neighbours
8  #-----------------------------------------------------------------------
9  dx=np.array([ 1, 0,1,-1,0,-1])              # nearest neighbour template
10 dy=np.array([-1,-1,0, 1,1, 0])
11 image=np.zeros([N,N],dtype='int')           # Initialize lattice to white...
12 image[N//2,N//2]=1                          # ...except central node to black
13 plt.scatter(N//2,N//2)                      # Set up plot, with central node
14 plt.axis([0,N,0,N])
15 plt.axes().set_aspect('equal')
16 for iterate in range(1,n_iter):             # Iteration loop
17
18     update=np.zeros([N,N],dtype='int')      # Set/reset evolution array
19
20     for i in range(1,N-1):                  # Lattice loops
21         for j in range(1,N-1):
22             cumul=0
23             for k in range(0,n_neighbour):  # Loop over nearest-neighbour
24                 cumul+=image[i+dx[k],j+dy[k]]
25             if image[i,j]==0 and cumul==1:  # Only one active neighbour
26                 update[i,j]=1               # Activate node
27                 plt.scatter(j+(i-N//2)/2.,N//2+0.866*(i-N//2)) # Plot node
28     # End of lattice loops
29
30     image+=update                           # Synchronous update of CA
31
32 # End of iteration loop
33 plt.show()                                  # Display structure
34 # END
```

Figure 2.6: A source code in the Python programming language for a two-dimensional CA defined over a triangular lattice with 6-neighbour connectivity. This is a minimal implementation, emphasizing conceptual clarity over programming elegance, code length, or run-time speed. The various `matplotlib` instructions (lines 13–15, 27 and 33) display the final structure, with activated nodes shifted horizontally and compressed vertically (line 27) so as to correctly display the structure in physical space, as on Fig. 2.7. {`code:ca2d`}

1. The code is structured as an outer temporal loop running a preset number of temporal iterations `n_iter` (lines 16–28), inside of which two nested loops over each lattice dimension (lines 20–21) carry out the activation test.

2. The connectivity is enforced through the use of the 1D template arrays `dx` and `dy` in which are hardwired the relative location, measured in lattice increments, of the connected neighbours (lines 9–10); these 1D arrays are then used to access the 2D array `image` which stores the state of the CA proper (lines 23–24).

3. If a cell is to become active at the next iteration, its new state is temporarily stored in the 2D work array `update` (line 26), which is reset to zero at the beginning of each temporal iteration (line 18); only once all nodes have been tested is the lattice array `image` updated (line 30). This *synchronous update* is necessary, otherwise the lattice update would depend on the order in which nodes are tested in the first set of lattice loops, thus introducing an undesirable spatial bias that would distort growth.

Consider now growth beginning from a single occupied node (the "seed") at the center of a triangular lattice. The first three steps of the iterated growth process are illustrated via the color-coding of lattice nodes on Fig. 2.5D. Starting from a single active node, the next iteration is a hexagonal ring of 6 active nodes (in red) surrounding the original active node (in black). At the next iteration only the six nodes colored in orange abide to the 1-neighbour-only activation rule, but on the following iteration each of these 6 "branches" will generate an arc-shaped clump of 3 active nodes (in yellow) at its extremity. Figure 2.7 picks up the growth at iteration 5 (top left), with subsequent frames plotted at a cadence of 3 iterations. As the six "spines" grow radially outwards, the faces of the growing structure eventually fill inwards from the corners, until a hexagonal shape is produced; from that point on growth can only pick up again at the six corners, and later towards the centers of the faces, eventually adding another "layer" to the growing structure. The broken concentric white hexagons within the structure plotted in the bottom right corner of Fig. 2.7 are the imprint of this layered growth process. Evidently, here the 6-fold symmetry of the connectivity remains reflected in the global, "macroscopic" shape of the growing structure; this could perhaps have been expected, but certainly not the intricacies of details produced within the structure itself. In fact there is much more to these details than meets the eye; step back a bit to view the bottom right structure from a distance, and it will be hardly distinguishable from the middle left structure viewed at normal reading distance. This is again an indication of self-similarity.

All this being said, looking at Figure 2.7 the first thing that comes to mind is of course: snowflakes! It might appear ludicrous to suggest that our very artificial computational setup —triangular lattice, neighbour-based growth rule, etc— has anything to do with the "natural" growth of snowflakes, but we will have occasions to revisit this issue in due time.

A similar 2D CA simulation can be run on a Cartesian lattice with 8-neighbour connectivity, starting again with a single active node at lattice center. All that is needed is to append two elements to the stencil arrays `dx` and `dy` in the code listed on Fig. 2.6. The first four steps of the growth process are again indicated by the nodal color coding on Fig. 2.5B. The first iteration produces a $3 \times 3$ block of active nodes but at the next iteration our one-neighbour rule makes growth possible only along diagonals quartering this $3 \times 3$ block (orange nodes). The next iteration (yellow nodes) generate a 5-node 90-degree wedge about each of the four extrusions generated at the preceding iteration; except for the 4-fold symmetry, this is essentially the same growth pattern observed in 6-fold symmetry on the triangular lattice (Fig. 2.5D). Figure 2.8 picks up the growth process at iteration 5, and subsequent frames are plotted on a 3-iteration cadence, as on Fig. 2.7. Growth now proceeds from the corners of the squares, which spawn more squares at their corners, and so on as the structure keeps growing, once again in a self-similar fashion.

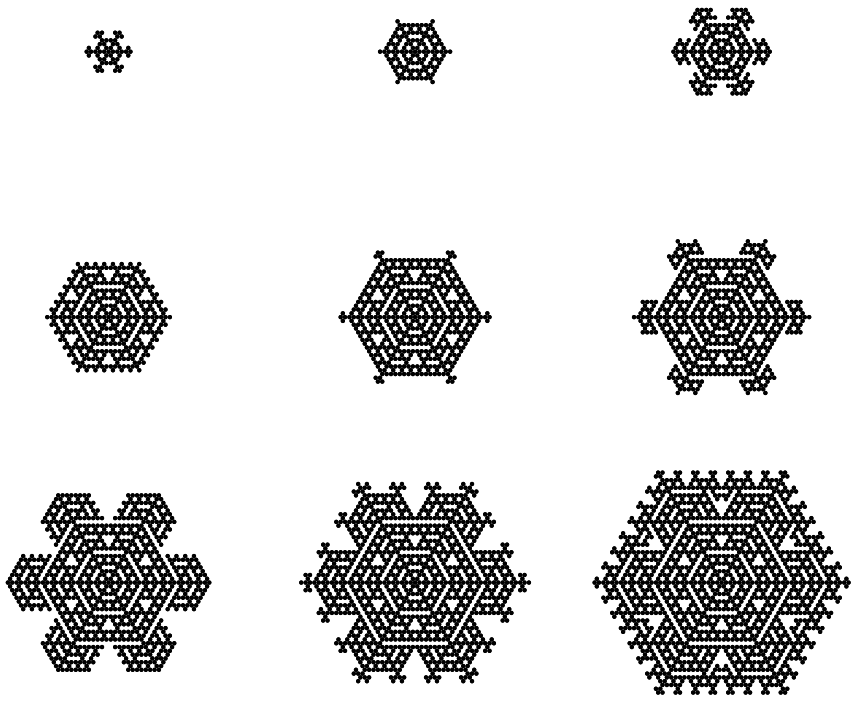

Figure 2.7: Structure growth generated by the 2D cellular automaton with the simple one-neighbour rule on an hexagonal, 6-neighbour lattice. The top left image shows the structure after 5 iterations, and the other images display the subsequent evolution on a 3-iteration cadence, the growth sequence being obvious. In the notation of §2.3 this rule is written as $(1)+1$. {fig:cahexgrowth}

## 2.3 A zoo of 2D structures from simple rules {sec:cazoo}

We henceforth restrict ourselves to a Cartesian lattice with 8-neighbours connectivity, and introduce a generalized 8-neighbour activation rule as follows: a node becomes active if either $n_1$ or $n_2$ neighbouring nodes are active; We also include in the rule the number $s$ of "seed" active nodes used to initialize the growth process. We write all this as:

{eq:2dgenrule}
$$(n_1, n_2) + s \qquad 1 \leq n_1, n_2 \leq 8 \ , \qquad n_1 < n_2 \ . \tag{2.1}$$

A specific example will likely help more than further explanations: the rule $(1, 5) + 1$ means that we start from one active node ($s = 1$); a node becomes active if either 1 or 5 neighbouring nodes are active; and remains inactive otherwise, namely if it has either 0, 2, 3, 4, 6, 7 or 8 active neighbours. Once again, no directional bias is introduced, as it does not matter where the 5 active nodes (say) are located in the 8-node group of neighbouring nodes. Under this notation, the rules used to grow the structure on Fig. 2.7 would be written as $(1) + 1$.

Figure 2.9 shows a sample of structures grown using various such rules, as labeled. The variety of structures produced even by this narrow subset of rules is quite staggering, including again self-similarity, mixture of order and disorder, compact structures porous or solid, etc. Some rules, such as $(3, 6) + 5$, do not even generate regular outward growth, as extrusions fold back inward to fill deep crevices left open in earlier phases of the iterated growth process.

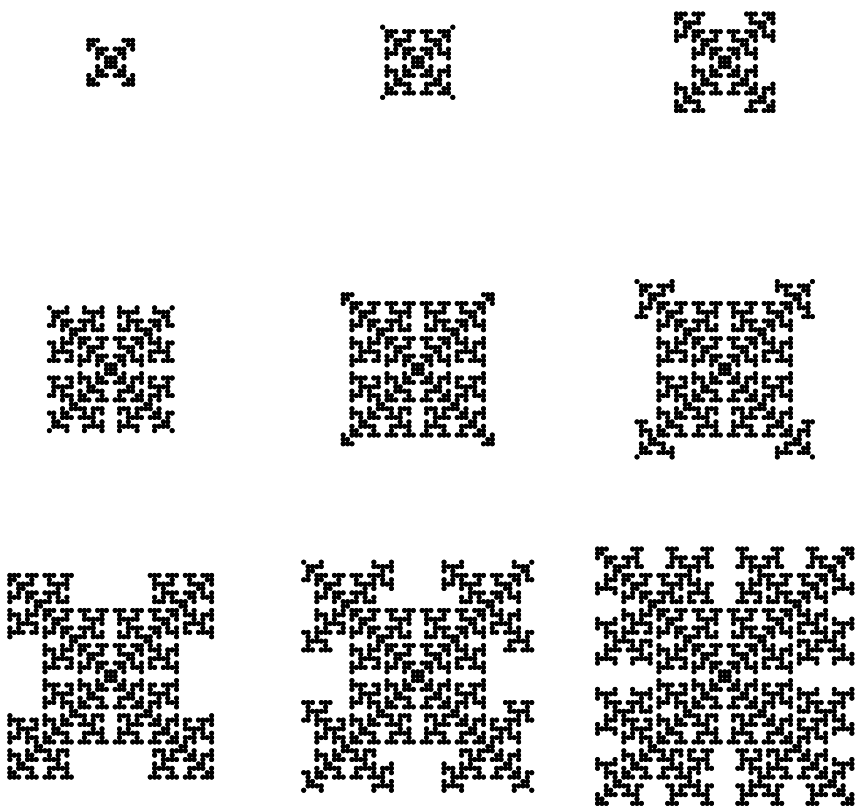

Figure 2.8: Structure generated by the 2D cellular automaton with the simple rule $(1)+1$, now on a cartesian, 8-neighbour lattice. The top left image shows the structure after 5 iterations, and the other images display the subsequent evolution on a 3-iteration cadence, the growth sequence being again obvious. {fig:cacartgrowth}

Staring as these and other structures generated by other specific incarnation of the 2-member rule (2.1), one is naturally tempted to extract some general trends; for example, Rules beginning with "1" produce squares that grow by spawning more squares at their corners, in a manner qualitatively similar to the basic $(1)+1$ rule; the numerical choice for $n_2$ affects primarily the internal pattern. Rules beginning with a "2', on the other hand, produce diamonds-shaped structures, with ordered and disordered internal regions, growing along their 4 approximately linear faces; the numerical choice for $n_2$ affects mostly the relative importance of ordered and disordered regions in the interior. Rules with a "3" produce compact structures, sometimes solid sometimes porous, with various patterns of symmetry about vertical, horizontal or diagonal axes present at the global scale. Now, if you find this convincing on the basis of Fig. 2.9, try running a simulation for rule $(3,7)+5$ and reconsider your position!

The overall conclusion of our relatively limited explorations of two-dimensional CA remains the same as with the one-dimensional CA considered previously: simple, microscopic growth rules can produce macroscopic structures ranging from highly regular to highly "complex", and very, very few of these structures could have been anticipated knowing only the lattice geometry and the growth rules.

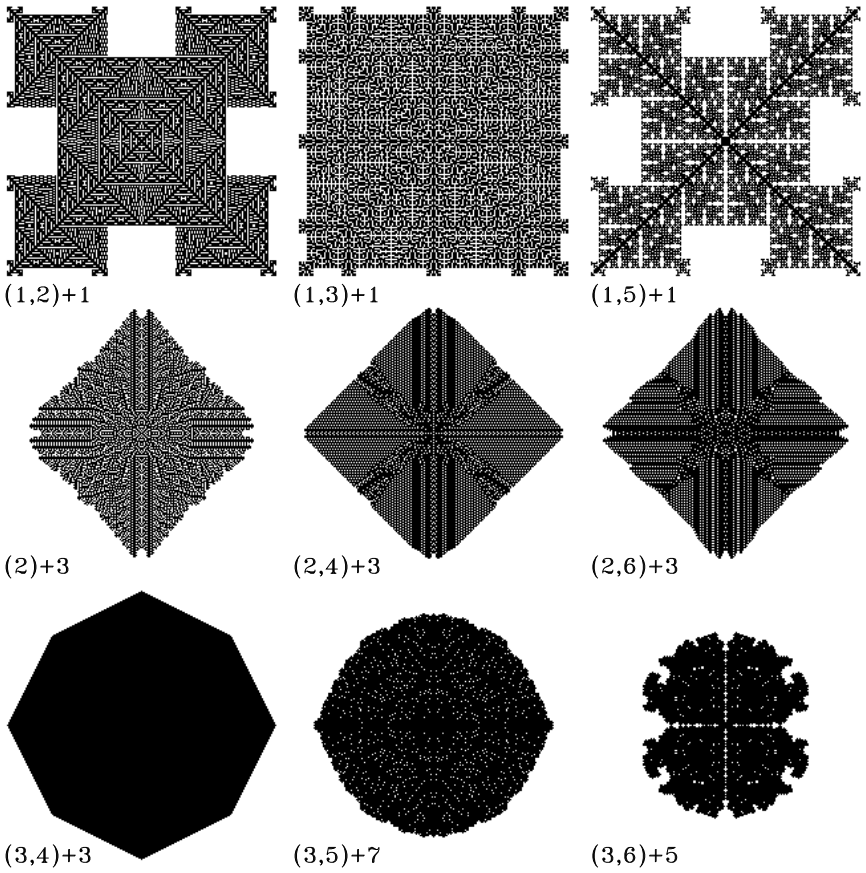


Figure 2.9: A zoo of structures grown by the 2D cellular automaton on a cartesian 8-neighbour lattice operating under a variety of rules, as labeled. All automata executed over 100 iterations, except for the bottom three, for which 200 iterations were executed. {fig:cagallery}

## 2.4 Agents, ants and highways {sec:ant}

In the "classical" CAs considered thus far, the active elements are the lattice nodes themselves, and so are fixed in space by definition. Another mechanism for iterated growth involves active elements moving on and interacting with the lattice (and/or with each other), according once again to set rules. Henceforth, such active elements will be defined as "Agents". For example, an agent known as an "ant" moves and operates on a lattice as follows, from one iteration to the next:

- Move forward;
- If standing on a white node, paint it black and turn right by 90 degrees;
- If standing on a black node, paint it a whiter shade of pale (meaning white) and turn left by 90 degrees;

These are pretty simple behavioral rules, yet they hold surprises in stock for us.

Figure 2.10 gives a simple numerical implementation of these behavioral rules. As with most codes listed throughout this book, logical clarity and readability have been given precedence

over programming elegance, code length, or run-time speed, and computing language-specific capabilities are systematically avoided. Note the following:

1. The code is again structured as an outer temporal loop running a preset number of temporal iterations `n_iter` (starting on line 15).

2. The two arrays `x_step` and `y_step` store the $x$- and $y$-increments associated with the four possible displacements, in the order down, left, up, right (lines 8–9). These are used to update the ant's position on the lattice (`ix, iy`) as per the ant's direction, stored in the variable `direction`. Under this ordering convention, turning right requires incrementing `direction` by $+1$ (line 24), and left by $-1$ (line 28).

3. The modulus operator "`%`" is used to enforce periodicity for the ant's position (lines 19–20) and stepping direction (lines 25 and 29). The instruction `a % b` returns the remainer of the division of `a` by `b`, e.g., $7\%3 = 1$, $2\%3 = 2$, $3\%3 = 0$. See §A.3 for more on the use of the modulus operator in Python.

4. The *change* in the lattice state at the ant's position is first calculated (variable `update`) and the lattice updated (line 31) only once the **if**...**else** construct is exited. This is needed because the lattice state at the ant's position sets the operating condition of this logical structure, so changing its value *within* its blocks of instructions is definitely not a good idea in most programming languages.

The top panel on Figure 2.11 shows the structure built by a single ant moving on a $300 \times 300$ lattice, starting at the location marked by the red dot, and initially pointing North (top of the page). The initial state of the lattice is all-white nodes. These are the parameter setting and initial conditions implemented in the code listed in Fig. 2.10. The first few thousands of iterations, shown in the inset framed in red, produce if not a strictly random, at least highly disordered clump of white and black nodes. But after a bit more than 10000 time steps, a switch to a different behavior takes place. The ant now executes a periodic series of steps, involving a lot of backtracking but also a net drift velocity along a diagonal with respect to the lines of the Cartesian lattice, leaving behind in its trail a highly ordered, spatially periodic pattern of white and black nodes (see green inset). This behavior has been dubbed "highway building", and it could hardly have been expected on the basis of the ant's simple behavioral rules. Highway building always proceeds along 45 degree diagonals, and once started would go on forever on an infinite size lattice. The fact that the highway points here to the South-East on Fig. 2.11 is determined by the initial condition: all-white nodes and the ant pointing North.

In practice simulations such as on Fig. 2.11 are carried out on a finite size lattice, on which horizontal and vertical periodicity is enforced. So here, pushing the simulation farther would eventually lead to the ant (and its highway) leaving the lattice near the SE corner, to reappear near the NW corner, still heading SE, eventually hitting the structure it just built. This throws the ant into a fit; disordered (re)painting prevails for a while, forming a structure statistically similar to that characterizing the first $10^4$ iterations, but after many thousands of iterations highway building resumes, in a direction orthogonal to that of the original highway, to the SW here. As the lattice fills with blotches of disorder and stretches of highways crossing each other, highway building becomes increasingly difficult, and if the evolution is pushed sufficiently far, on any finite-sized lattice the end result is randomness.

Highway building is a pretty delicate process that is easily disturbed. The bottom panel of Figure 2.11 shows what happens when a small number of randomly selected lattice nodes are painted black before the ant starts moving. At first the evolution proceeds as before, and highway building towards the SE begins, but soon the ant hits one of the randomly distributed black node, triggering disordered painting. Highway building eventually resumes, still towards the SE, until another black node is encountered, triggering another, shorter disordered episode that ends with highway building resuming now towards the NE; and so on over the 51000 iterations over which this specific simulation was pursued.

```
# HIGHWAY BUILDING BY LANGTON'S ANT
import numpy as np
import matplotlib.pyplot as plt
#-----------------------------------------------------------------------------
N     =300                              # Lattice size
n_iter=20000                            # Number of temporal iterations
#-----------------------------------------------------------------------------
x_step=np.array([0,-1,0,1])             # Template arrays for steps
y_step=np.array([1,0,-1,0])
image=np.zeros([N,N],dtype='int')       # Initialize lattice array, all white
ix=N//4                                 # Ant's starting position in x
iy=N//4                                 # Ant's starting position in y
direction=1                             # Ant's starting direction, North

for iteration in range(0,n_iter):       # Temporal loop

    ix+=x_step[direction]               # Ant moves
    iy+=y_step[direction]
    ix=(N+ix) % N                       # Enforce periodicity in x
    iy=(N+iy) % N                       # Enforce periodicity in y

    if image[iy,ix] == 0:               # On a white node
        update=1                        # Paint it black...
        direction+=1                    # ...and turn right...
        direction=direction % 4         # ...but stay within step array
    else:                               # On a black node
        update=-1                       # Paint it white...
        direction-=1                    # ...and turn left...
        direction=(4+direction) % 4     # ...but stay within step array

    image[iy,ix]+=update
# End of temporal loop
plt.imshow(image,interpolation="nearest")
plt.show()                              # Display final structure
# END
```

Figure 2.10: A source code in the Python programming language for an "ant" agent abiding to the rules introduced in §2.4. This code generates the simulation plotted on the top panel of Fig. 2.11 below (minus the colored dots and insets). {code:ant}

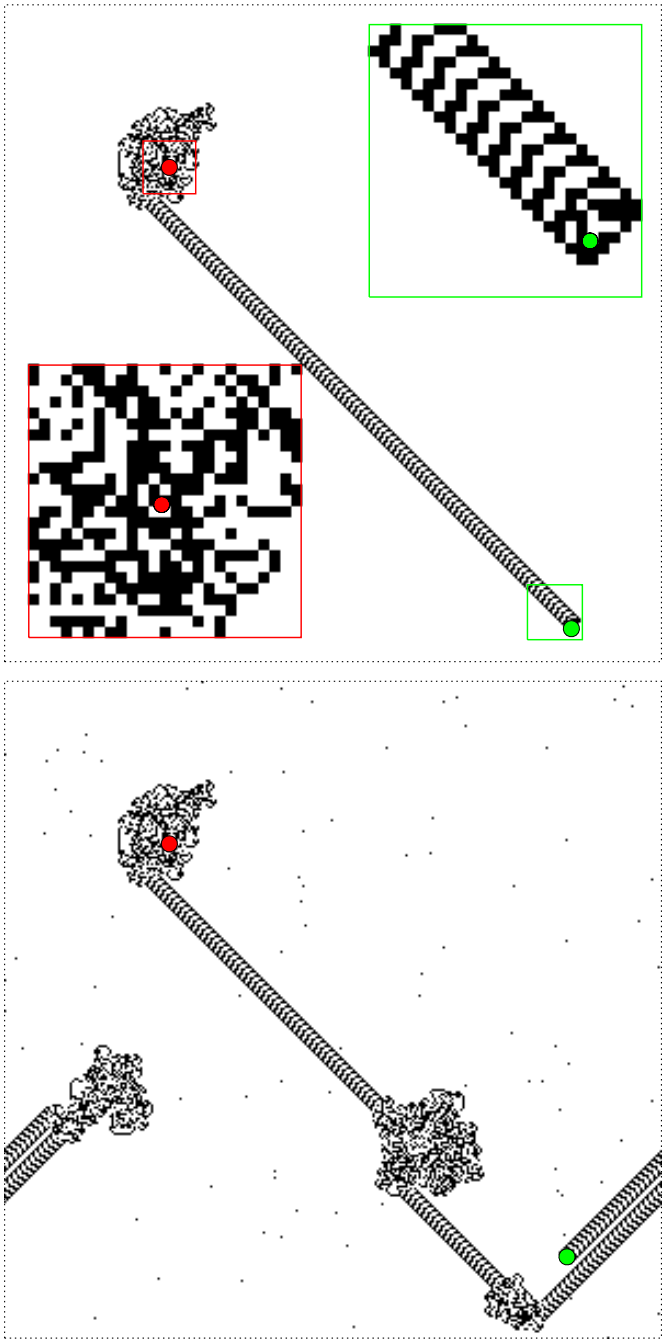

Figure 2.11: Highway building by an "ant agent" in a clean (top, 20000 iterations) and noisy (bottom, 51000 iterations) background environment. The solid dots show the starting (red) and ending (green) position of the ant, with the inset on the top panel providing closeups of the lattice about these two points. The lattice is assumed periodic in both the horizontal and vertical. See §D.2 for more on periodic boundaries conditions on lattices. {fig:ant}

## 2.5 Emergent structures and behaviors

{sec:meascomp

We have barely scratched into the realm of structures and behaviors that can be produced by CA and CA-like systems. Nonetheless, the take-home message of this chapter should be already clear: very simple rules can produce very complex-looking structures. But should we really be calling these structures "complex" if their generating rules are simple? Or do we remain tied to an intuitive definition of "complex" relying on our visual perception of structures and behaviors? Students of complexity have been rattling their brains over that one for quite a while now.

If defining complexity is hard, coming up with an unambiguous and universal quantitative *measure* of complexity is perhaps the next hardest thing. Many such measures have been defined, and can be categorized in terms of the question they attempt to answer; given a complex structure or system, the following three queries are all pertinent:

1. How hard is it to describe ?
2. How hard is it to create ?
3. How is it dynamically organized ?

Consider for example the measure known as *algorithmic complexity*, namely the length of the smallest computer program that can generate a given output —a spatial pattern, a time series, a network, whatever. It may appear eminently reasonable to suppose that more complex patterns require longer programs; simulating the evolving climate certainly requires a much longer code (and a lot more computer time!) than simulating the harmonic oscillation of a frictionless pendulum. It seems to make sense, but we need to look no further than the simple 1D CAs investigated in §2.1 to realize the limitations of this measure of complexity. The CAs of Figs. 2.1, 2.2 and 2.3 can be produced by programs of exactly the same length, yet they could hardly be considered "equally complex".

Our brief foray into cellular automata also highlights a theme that will recur throughout this book and that is almost universally considered a defining feature of natural complexity: *emergence*; this term is used to refer to the fact that global structures or behaviors on macroscopic scales cannot be reduced to (or inferred from) the rules operating at the microscopic level of individual components making up the system; instead, they emerge from the *interactions* between these components. Synthetic snowflakes and ant highways are such examples of emergence, and are by no means the last to be encountered in this book.

## 2.6 Exercises and further computational explorations

1. Use the 1D CA code of Figure 2.4 to explore the behavior of the four CA rules introduced in §2.1 when starting from a random initial condition, i.e., each cell is randomly assigned black or white color with equal probability. If needed see §C.2 for a quick start on generating uniformly distributed random deviates in Python. Make sure also to enforce periodic boundary conditions (see §D.2).

2. The aim of this exercise is to explore further the patterns produced by 1D CA rules, all of which relatively easy to implement in the source code of Fig. 2.4. The following 5 individual rules produce patterns qualitatively distinct from those already examined in §2.1. Unless the pattern looks really trivial, make sure to run the CA for enough iterations to ascertain its long-term behavior.

   - Cell $j$ becomes (or stays) black only if both $j-1$ and $j$ are white; otherwise the cell stays (or becomes) white.
   - Cell $j$ becomes (or stays) black if any two cells of the triad $[j-1, j, j+1]$ are black, or if both $j-1$ and $j$ are white; otherwise the cell stays (or becomes) white.

- Cell $j$ becomes (or stays) white if $j-1$ and $j$ are both black, or if the triad $[j-1, j, j+1]$ are all white; otherwise the cell stays (or becomes) black.
- Cell $j$ becomes (or stays) black if cell $j-1$ and at least one of the pair $[j, j+1]$ are black, or if the triad $[j-1, j, j+1]$ are all white; otherwise the cell stays (or becomes) white.
- Cell $j$ becomes (or stays) white if $j+1$ and $j$ are both white, or if the triad $[j-1, j, j+1]$ are all black; otherwise the cell stays (or becomes) black.

The last two rules, in particular, should be iterated over many hundreds of iteration over a large lattice; the patterns produced are particularly intriguing. You should also run these five rules starting from a random initial condition. To which CA class does each belong ?

3. Modify the code of Fig. 2.6 to introduce 2-member rules such as those used to produce Fig. 2.9 on the 6-neighbour triangular lattice. Explore the growth produced by the set of rules $(1,2)+1$ through $(1,6)+1$. Is lattice structure always imprinted on global shape ?

4. Modify the code of Fig. 2.6 to operate on a Cartesian 8-neighbour lattice, and explore the sensitivity to initial conditions for rules $(3,4)+n$ and $(3,5)+n$. More specifically, consider the growth produced by using either $n = 3$, 4 or 5 active nodes, organized either linearly, as a $2 \times 2$ block, as a diamond-shaped 5-node block, etc. Is the geometry of the initial condition imprinted on global shape ?

5. The Game of Life is one of the most intensely studied 2D cellular automaton. It is defined on a two-dimension Cartesian lattice periodic horizontally and vertically, with eight-neighbour connectivity. Each lattice node can be in one of two possible states, say "inactive" and "active" (or 0 and 1; or white and black; or dead and alive, whatever), and evolves from one iteration to the next according to the following rules:

   - if an active node has less than two active neighbours, it becomes inactive;
   - if an active node has more than three active neighbours, it becomes inactive;
   - if an inactive node has three active neighbour, it become active;
   - if a node has two active neighbours, it remains in its current state.

   This automaton can generate "organisms", i.e. shape-preserving structures moving on the lattice, in some cases interacting with one one another or with their environment to produce even more intricate behaviors. Modify the code of Fig. 2.6 to incorporate the above rules, and run simulations starting from a random initialization of the lattice in which each node is assigned active or inactive status with equal probability.

6. The Grand Challenge for this chapter is to explore the behavior of another interesting ant-like agent, known as the "termite". Termites move randomly on a lattice on which "wood chips" (i.e. black) have been randomly dispersed. The termite's behavioral rules are the following:

   - Random walk until coming up against a wood chip;
   - **if** currently carrying a wood chip, drop it at current position (i.e., next to the one just bumped into), and resume random walk;
   - **else**, pick up the chip bumped into, and resume random walk.

   Code this up, perhaps starting from the "ant" code of Fig. 2.10. Section D.3 may prove useful if you need a kickstart on how to code up random walk on a lattice. How is the distribution of wood chips evolving with time ? Does this change if you let loose more than one termite on the lattice ?

## 2.7 Further readings

Pretty much anything and everything that could be written on cellular automata can be found in

Wolfram, S., *A new kind of science*, Wolfram Media Inc. (2002).

The material covered in §2.1 and 2.2 follows rather closely parts of chapters 2 and 8 of this massive tome. The Wikipedia page on Cellular Automata includes a good discussion of the history of CA research, with copious references to the early literature (viewed March 2015).

For a succinct and engaging introduction to virtual ants and similar computational insects, see

Resnick, M., *Turtles, Termites and Traffic Jams*, MIT Press (1994).

as well as chapter 16 in

Flake, G.W., *The computational beauty of Nature*, MIT Press (1998).

Chapter 15 of this volume also offers a nice introduction to cellular automata, including the Game of Life. As far as I know, the general notion of an "Agent" has been borrowed from economics and introduced in complexity science by John Holland; for more on this concept see:

Holland, J.H., *Hidden Order*, Reading: Addison-Wesley (1995).

Some years ago the term *Artificial Life* was coined to define a category for computational ants, termites, turmites, boids, and other similarly designed computational critters, as well as those appearing in John Conway's Game of Life. The following collection of papers remains a great overview of this computational zoology and its underlying motivations:

Langton, C. (ed.), *Artificial Life*, MIT Press (1995).

Langton is actually the designer of the ant agent starring in §2.4. The Wikipedia page on Langton's ant is worth viewing; it also provides examples of extensions to multiple states ants, as well as a good sample of references into the technical literature:

`http://en.wikipedia.org/wiki/Langton%27s_ant` (viewed March 2015)

and, not to be missed, a detailed study of the real thing:

Gordon, D.M., *Ant encounters: interaction networks and colony behavior*, Princeton University Press (2010).

On measuring complexity, see

Lloyd, S., *IEEE Control Syst. Mag,*, **21** (2001),
Machta, J., *Chaos*, **21**, 037111 (2011) .

The three-questions categorization of complexity measures introduced in §2.5 is taken from the first of these papers, which lists no less than 42 such measures, in a "nonexhaustive list" ! The second paper is part of a focus issue of the research Journal *Chaos*, devoted to measures of complexity.