EXCERPTED FROM

STEPHEN WOLFRAM
A NEW KIND OF SCIENCE

NOTES FOR CHAPTER 5:

Two Dimensions and Beyond

Two Dimensions and Beyond

Introduction

■ **Other lattices.** See page 929.

■ **Page 170 · 1D phenomena.** Among the phenomena that cannot occur in one dimension are those associated with shape, winding and knotting, as well as traditional phase transitions with reversible evolution rules (see page 981).

Cellular Automata

■ **Implementation.** An *n×n* array of white squares with a single black square in the middle can be generated by

```
PadLeft[{{1}}, {n, n}, 0, Floor[{n, n}/2]]
```

For the 5-neighbor rules introduced on page 170 each step can be implemented by

```
CAStep[rule_, a_] := Map[rule[[10 - #]] &,
    ListConvolve[{{0, 2, 0}, {2, 1, 2}, {0, 2, 0}}, a, 2], {2}]
```

where *rule* is obtained from the *code* number by *IntegerDigits[code, 2, 10]*.

For the 9-neighbor rules introduced on page 177

```
CAStep[rule_, a_] := Map[rule[[18 - #]] &,
    ListConvolve[{{2, 2, 2}, {2, 1, 2}, {2, 2, 2}}, a, 2], {2}]
```

where *rule* is given by *IntegerDigits[code, 2, 18]*.

In *d* dimensions with *k* colors, 5-neighbor rules generalize to *(2d+1)*-neighbor rules, with

```
CAStep[{rule_, d_}, a_] :=
  Map[rule[[-1 - #]] &, a + k AxesTotal[a, d], {d}]
AxesTotal[a_, d_] := Apply[Plus, Map[RotateLeft[a, #] +
    RotateRight[a, #] &, IdentityMatrix[d]]]
```

with *rule* given by *IntegerDigits[code, k, k (2 d (k - 1) + 1)]*.

9-neighbor rules generalize to 3^d-neighbor rules, with

```
CAStep[{rule_, d_}, a_] :=
  Map[rule[[-1 - #]] &, a + k FullTotal[a, d], {d}]
FullTotal[a_, d_] :=
  Array[RotateLeft[a, {##}] &, Table[3, {d}], -1, Plus] - a
```

with *rule* given by *IntegerDigits[code, k, k ((3^d - 1) (k - 1) + 1)]*.

In 3 dimensions, the positions of black cells can conveniently be displayed using

```
Graphics3D[Map[Cuboid[-Reverse[#]] &, Position[a, 1]]]
```

■ **General rules.** One can specify the neighborhood for any rule in any dimension by giving a list of the offsets for the cells used to update a given cell. For 1D elementary rules the list is *{{-1}, {0}, {1}}*, while for 2D 5-neighbor rules it is *{{-1, 0}, {0, -1}, {0, 0}, {0, 1}, {1, 0}}*. In this book such offset lists are always taken to be in the order given by *Sort*, so that for range *r* rules in *d* dimensions the order is the same as *Flatten[Array[List, Table[2 r + 1, {d}], -r], d - 1]*. One can specify a neighborhood configuration by giving in the same order as the offset list the color of each cell in the neighborhood. With offset list *os* and *k* colors the possible neighborhood configurations are

```
Reverse[Table[IntegerDigits[i - 1,
    k, Length[os]], {i, k^Length[os]}]]
```

(These are shown on page 53 for elementary rules and page 941 for 5-neighbor rules.) If a cellular automaton rule takes the new color of a cell with neighborhood configuration *IntegerDigits[i, k, Length[os]]* to be *u[[i + 1]]*, then one can define its rule number to be *FromDigits[Reverse[u], k]*. A single step in evolution of a general cellular automaton with state *a* and rule number *num* is then given by

```
Map[IntegerDigits[num, k, k^Length[os]][[-1 - #]] &,
  Apply[Plus, MapIndexed[k^(Length[os] - First[#2])
      RotateLeft[a, #1] &, os]], {-1}]
```

or equivalently by

```
Map[IntegerDigits[num, k, k^Length[os]][[-# - 1]] &,
  ListCorrelate[Fold[ReplacePart[k #1, 1, #2 + r + 1] &,
    Array[0 &, Table[2 r + 1, {d}]], os], a, r + 1], {d}]
```

■ **Numbers of possible rules.** The table below gives the total number of 2D rules of various types with two possible colors for each cell. Given an initial pattern with a certain symmetry, a rule will maintain that symmetry if the rule is such that every neighborhood equivalent under the symmetry yields the same color of cell. Rules are considered rotationally symmetric in the table below if they preserve any possible rotational symmetry consistent with the underlying arrangement of cells. Totalistic rules depend only on the total number of black cells in a neighborhood; outer totalistic rules (as in the previous note) also depend on the color of the center cell. Growth totalistic rules make any cell that becomes black remain black forever.

In such a rule, given a list of how many neighbors around a given cell (out of *s* possible) make the cell turn black the outer totalistic code for the rule can be obtained from

```
Apply[Plus, 2^Join[2 list, 2 Range[s + 1] - 1]]
```

	5 - neighbor square	9 - neighbor square	hexagonal
general	$2^{32} \simeq 4\times10^9$	$2^{512} \simeq 10^{154}$	$2^{128} \simeq 3\times10^{38}$
rotationally symmetric	$2^{12} = 4096$	$2^{140} \simeq 10^{42}$	$2^{28} \simeq 3\times10^8$
completely symmetric	$2^{12} = 4096$	$2^{102} \simeq 5\times10^{30}$	$2^{26} \simeq 7\times10^7$
outer totalistic	$2^{10} = 1024$	$2^{18} \simeq 3\times10^5$	$2^{14} = 16384$
totalistic	$2^6 = 64$	$2^{10} = 1024$	$2^8 = 256$
growth totalistic	$2^5 = 32$	$2^9 = 512$	$2^7 = 128$

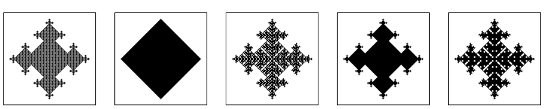
■ **Symmetric 5-neighbor rules.** Among the 32 possible 5-cell neighborhoods shown for example on page 941 there are 12 classes related by symmetries, given by

```
s = {{1}, {2, 3, 9, 17}, {4, 10, 19, 25},
   {5}, {6, 7, 13, 21}, {8, 14, 23, 29}, {11, 18},
   {12, 20, 26, 27}, {15, 22}, {16, 24, 30, 31}, {28}, {32}}
```

Completely symmetric 5-neighbor rules can be numbered from 0 to 4095, with each digit specifying the new color of the cell for each of these symmetry classes of neighborhoods. Such rule numbers can be converted to general form using

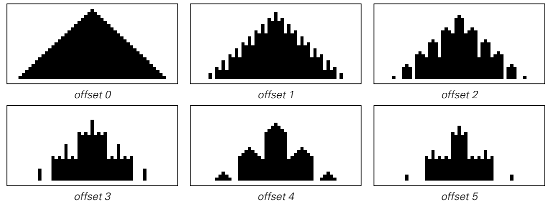
```
FromDigits[Map[Last, Sort[Flatten[Map[Thread,
      Thread[{s, IntegerDigits[n, 2, 12]}]], 1]]], 2]
```

■ **Growth rules.** The pictures below show examples of rules in which a cell becomes black if it has exactly the specified numbers of black neighbors (the initial conditions used have the minimal number of black cells for growth). The code numbers in these cases are given by $2/3\,(4^n - 1) + \mathit{Apply}[\mathit{Plus}, 4^{\mathit{list}}]$ where *n* is the number of neighbors, here 5. (See also the 9-neighbor examples on page 373.)



{1} {1, 2} {1, 3} {1, 4} {1, 3, 4}

■ **Page 171 · Code 942 slices.** The following is the result of taking vertical slices through the pattern with a sequence of offsets from the center:

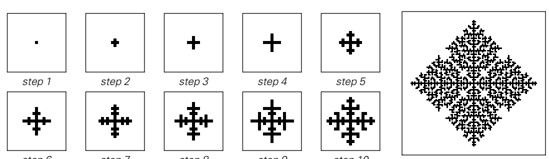


offset 0 offset 1 offset 2 offset 3 offset 4 offset 5

■ **History.** As indicated on pages 876–878, 2D cellular automata were historically studied more extensively than 1D ones—though rarely with simple initial conditions. The 5-cell neighborhood on page 170 was considered by John von Neumann in 1952; the 9-cell one on page 177 by Edward Moore in 1962. (Both are also common in finite difference approximations in numerical analysis.) (The 7-cell hexagonal neighborhood of page 369 was considered for image processing purposes by Marcel Golay in 1959.) Ever since the invention of the Game of Life around 1970 a remarkable number of hardware and software simulators have been built to watch its evolution. But until after my work in the 1980s simulators for more general 2D cellular automata were rare. A sequence of hardware simulators were nevertheless built starting in the mid-1970s by Tommaso Toffoli and later Norman Margolus. And as mentioned on page 1077, going back to the 1950s some image processing systems have been based on particular families of 2D cellular automaton rules.

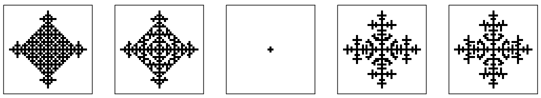
■ **Ulam systems.** Having formulated the system around 1960, Stanislaw Ulam and collaborators (see page 877) in 1967 simulated 120 steps of the process shown below, with black cells after *t* steps occurring at positions

```
Map[First,
  First[Nest[UStep[p[q[r[#1], #2]] &, {{1, 0}, {0, 1}, {-1, 0},
        {0, -1}}, #] &, ({#, #} &)[{{{0, 0}, {0, 0}}}], t]]]
UStep[f_, os_, {a_, b_}] := ({Join[a, #], #} &)[f[Flatten[
      Outer[{#1 + #2, #1} &, Map[First, b], os, 1], 1], a]]
r[c_] := Map[First, Select[Split[Sort[c],
      First[#1] == First[#2] &], Length[#] == 1 &]]
q[c_, a_] := Select[c,
    Apply[And, Map[Function[u, qq[#1, u, a]], a]] &]
p[c_] := Select[c,
    Apply[And, Map[Function[u, pp[#1, u]], c]] &]
pp[{x_, u_}, {y_, v_}] := Max[Abs[x - y]] > 1 || u == v
qq[{x_, u_}, {y_, v_}, a_] := x == y || Max[Abs[x - y]] > 1 ||
    u == y || First[Cases[a, {u, z_} → z]] == y
```



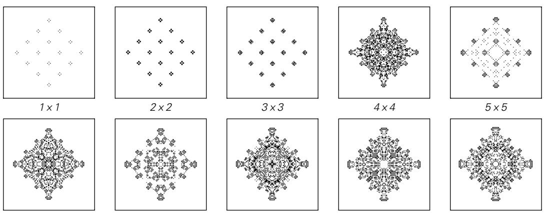
step 1 step 2 step 3 step 4 step 5 step 6 step 7 step 8 step 9 step 10 step 50

These rules are fairly complicated, and involve more history than ordinary cellular automata. But from the discoveries in this book we now know that much simpler rules can also yield very complicated behavior. And as the pictures below show, this is true even just for parts of the rules above (*s* alone yields outer totalistic code 686 in 2D, and rule 90 in 1D).



r[] q[] p[] p[q[]] p[q[r[]]]

Ulam also in 1967 considered the pure 2D cellular automaton with outer totalistic code 12 (though he stated its rule in a complicated way). As shown in the pictures below, when started from blocks of certain sizes this rule yields complex patterns—although nothing like this was noted in 1967.



1×1 2×2 3×3 4×4 5×5 6×6 7×7 8×8 9×9 10×10

■ **Limiting shapes.** When growth occurs at the maximum rate the outer boundaries of a cellular automaton pattern reflect the neighborhood involved in its underlying rule (in rough analogy to the Wulff construction for shapes of crystals). When growth occurs at a slower rate, a wide range of polygonal and other shapes can be obtained, as illustrated in the main text.

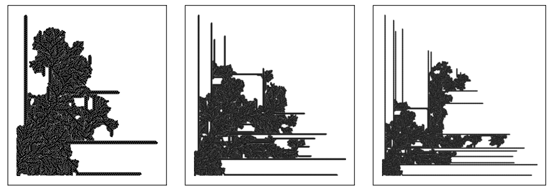
■ **Additive rules.** See page 1092.

■ **Page 174 · Cellular automaton art.** 2D cellular automata can be used to make a wide range of designs for rugs, wallpaper, and similar objects. Repeating squares of pattern can be produced by using periodic boundary conditions. Rules with more than two colors will sometimes be appropriate. For rugs, it is typically desirable to have each cell correspond to more than one tuft, since otherwise with most rules the rug looks too busy. (Compare page 872.)

■ **Page 177 · Code 175850.** See also page 980.

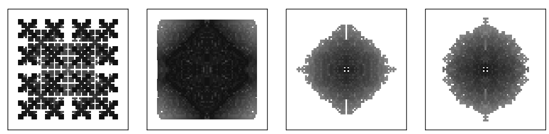
■ **Page 178 · Code 746.** The pattern generated is not perfectly circular, as discussed on page 979. Its interior is mostly fixed, but there are scattered small regions that cycle with a variety of periods.

■ **Page 181 · Code 174826.** The pictures below show the upper-right quadrant for more steps. Most of the lines visible are 8 cells across, and grow by 4 cells every 12 steps. They typically survive being hit by more complicated growth from the side. But occasionally runners 3 cells wide will start on the side of a line. And since these go 2 cells every 3 steps they always catch up with lines, producing complicated growth, often terminating the lines.



step 1000 step 2000 step 3000

■ **Page 183 · Projections from 3D.** Looking from above, with closer cells shown darker, the following show patterns generated after 30 steps, by (a) the rule at the top of page 183, (b) the rule at the bottom of page 183, (c) the rule where a cell becomes black if exactly 3 out of 26 neighbors were black and (d) the same as (c), but with a 3×3×1 rather than a 3×1×1 initial block of black cells:

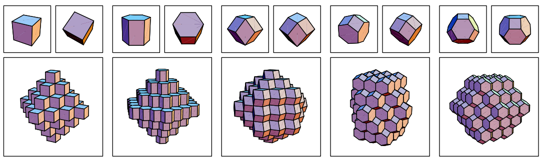


(a) (b) (c) (d)

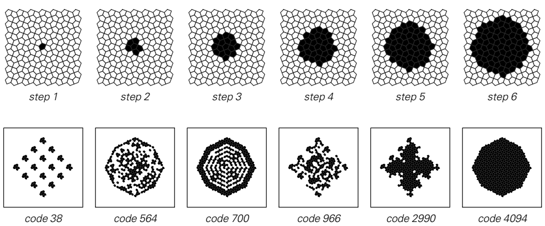
■ **Other geometries.** Systems like cellular automata can readily be set up on any geometrical structure in which a limited number of types of cells can be identified, with every cell of a given type having a similar neighborhood.

In the simplest case, the cells are all identical, and are laid out in the same orientation in a repetitive array. The centers of the cells form a lattice, with coordinates that are integer multiples of some set of basis vectors. The possible complete symmetries of such lattices are much studied in crystallography. But for the purpose of nearest-neighbor cellular automaton rules, what matters is not detailed geometry, but merely what cells are adjacent to a given cell. This can be determined by looking at the Voronoi region (see page 987) for each point in the lattice. In any given dimension, this region (variously known as a Dirichlet domain or Wigner-Seitz cell, and dual to the primitive cell, first Brillouin zone or Wulff shape) has a limited number of possible overall shapes. The most symmetrical versions of these shapes in 2D are the square (4 neighbors) and hexagon (6) and in 3D (as found by Evgraf Fedorov in 1885) the cube (6), hexagonal prism (8), rhombic dodecahedron (12) (e.g.

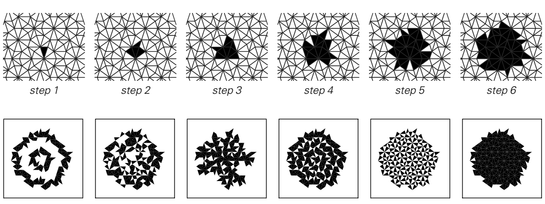
face-centered cubic crystals), rhombo-hexagonal or elongated dodecahedron (12) and truncated octahedron or tetradecahedron (14) (e.g. body-centered cubic crystals), as shown below. (In 4D, 8, 16 and 24 nearest neighbors are possible; in higher dimensions possibilities have been investigated in connection with sphere packing.) (Compare pages 1029 and 986.)



In general, there is no need for individual cells in a cellular automaton to have the same orientation. A triangular lattice is one example where they do not. And indeed, any tiling of congruent figures can readily be used to make a cellular automaton, as illustrated by the pentagonal example below. (Outer totalistic codes specify rules; the first rule makes a particular cell black when any of its five neighbors are black and has code 4094. Note that even though individual cells are pentagonal, large-scale cellular automaton patterns usually have 2-, 4- or 8-fold symmetry.)



There is even no need for the tiling to be repetitive; the picture below shows a cellular automaton on a nested Penrose tiling (see page 932). This tiling has two different shapes of tile, but here both are treated the same by the cellular automaton rule, which is given by an outer totalistic code number. The first example is code 254, which makes a particular cell become black when any of its three neighbors are black. (Large-scale cellular automaton patterns here can have 5-fold symmetry.) (See also page 1027.)



■ **Networks.** Cellular automata can be set up so that each cell corresponds to a node in a network. (See page 936.) The only requirement is that around each node the network must have the same structure (or at least a limited number of possible structures). For nearest-neighbor rules, it suffices that each node has the same number of connections. For longer-range rules, the network must satisfy constraints of the kind discussed on page 483. (Cayley graphs of groups always have the necessary homogeneity.) If the connections at each node are not labelled, then only totalistic cellular automaton rules can be implemented. Many topological and geometrical properties of the underlying network can affect the overall behavior of a cellular automaton on it.

Turing Machines

■ **Implementation.** With rules represented as a list of elements of the form *{s, a} → {sp, ap, {dx, dy}}* (*s* is the state of the head and *a* the color of the cell under the head) each step in the evolution of a 2D Turing machine is given by

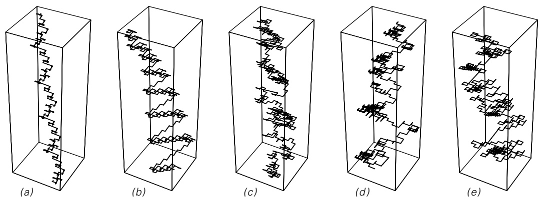
```
TM2DStep[rule_, {s_, tape_, r : {x_, y_}}] :=
  Apply[{#1, ReplacePart[tape, #2, {r}], r + #3} &,
    {s, tape[[x, y]]} /. rule]
```

■ **History.** At a formal level 2D Turing machines have been studied since at least the 1950s. And on several occasions systems equivalent to specific simple 2D Turing machines have also been constructed. In fact, much as for cellular automata, more explicit experiments have been done on 2D Turing machines than 1D ones. A tradition of early robotics going back to the 1940s—and leading for example to the Logo computer language—involved studying idealizations of mobile turtles. And in 1971 Michael Paterson and John Conway constructed what they described as an idealization of a prehistoric worm, which was essentially a 2D Turing machine in which the state of the head records the direction of the motion taken at each step. Michael Beeler in 1973 used a computer at MIT to investigate all 1296 possible worms with rules of the simplest type on a hexagonal grid, and he found several with fairly complex behavior. But this discovery does not appear to have been followed up, and systems equivalent to simple 2D Turing machines were reinvented again, largely independently, several times in the mid-1980s: by Christopher Langton in 1985 under the name "vants"; by Rudy Rucker in 1987 under the name "turmites"; and by Allen Brady in 1987 under the name "turning machines". The specific 4-state rule

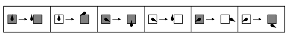
```
{s_, c_} :→ With[{sp = s (2 c - 1) ⅈ},
    {sp, 1 - c, {Re[sp], Im[sp]}}]
```

has been called Langton's ant, and various studies of it were done in the 1990s.

■ **Visualization.** The pictures below show the 2D position of the head at 500 successive steps for the rules on page 185.



Some 2D Turing machines exhibit elements of randomness at some steps, but then fill in every so often to form simple repetitive patterns. An example is the 3-state rule



■ **Rules based on turning.** The rules used in the main text specify the displacement of the head at each step in terms of fixed directions in the underlying grid. An alternative is to specify the turns to make at each step in the motion of the head. This is how turtles in the Logo computer language are set up. (Compare the discussion of paths in substitution systems on page 892.)

■ **2D mobile automata.** Mobile automata can be generalized just like Turing machines. Even in the simplest case, however, with only four neighbors involved there are already $(4k)^{k^5}$ possible rules, or nearly 10^{29} even for $k = 2$.

Substitution Systems and Fractals

■ **Implementation.** With the rule on page 187 given for example by `{1 → {{1, 0}, {1, 1}}, 0 → {{0, 0}, {0, 0}}}` the result of t steps in the evolution of a 2D substitution system from a initial condition such as `{{1}}` is given by

```
SS2DEvolve[rule_, init_, t_] :=
  Nest[Flatten2D[# /. rule] &, init, t]
Flatten2D[list_] :=
  Apply[Join, Map[MapThread[Join, #] &, list]]
```

■ **Connection with digit sequences.** Just as in the 1D case discussed on page 891, the color of a cell at position `{i, j}` in a 2D substitution system can be determined using a finite automaton from the digit sequences of the numbers i and j. At step n, the complete array of cells is

```
Table[If[FreeQ[Transpose[IntegerDigits[{i, j}, k, n]], form],
    1, 0], {i, 0, k^n - 1}, {j, 0, k^n - 1}]
```

where for the pattern on page 187, $k = 2$ and `form = {0, 1}`. For patterns (a) through (f) on page 188, $k = 3$ and `form` is given respectively by (a) `{1, 1}`, (b) `{0|2, 0|2}`, (c) `{0|2, 0|2}|{1, 1}`, (d) `{i_, j_} /; j > i`, (e) `{0, 2}|{1, 1}|{2, 0}`, (f) `{0, 2}|{1, 1}`. Note that the excluded pairs of digits are in exact correspondence with the positions of which squares are `0` in the underlying rules for the substitution systems. (See pages 608 and 1091.)

■ **Page 187 · Sierpiński pattern.** Other ways to generate step n of the pattern shown here in various orientations include:

- `Mod[Array[Binomial, {2, 2}^n, 0], 2]` (see pages 611 and 870)
- `1 - Sign[Array[BitAnd, {2, 2}^n, 0]]` (see pages 608 and 871)
- `NestList[Mod[RotateLeft[#] + #, 2] &, PadLeft[{1}, 2^n], 2^n - 1]` (see page 870)
- `NestList[Mod[ListConvolve[{1, 1}, #, -1], 2] &, PadLeft[{1}, 2^n], 2^n - 1]` (see page 870)
- `IntegerDigits[NestList[BitXor[2#, #] &, 1, 2^n - 1], 2, 2^n]` (see page 906)
- `NestList[Mod[Rest[FoldList[Plus, 0, #]], 2] &, Table[1, {2^n}], 2^n - 1]` (see page 1034)
- `Table[PadRight[Mod[CoefficientList[(1 + x)^(t-1), x], 2], 2^n - 1], {t, 2^n}]` (see pages 870 and 951)
- `Reverse[Mod[CoefficientList[Series[1/(1 - (1 + x) y), {x, 0, 2^n - 1}, {y, 0, 2^n - 1}], {x, y}], 2]]` (see page 1091)
- `Nest[Apply[Join, MapThread[Join, {{#, #}, {0#, #}}, 2]] &, {{1}}, n]` (compare page 1073)

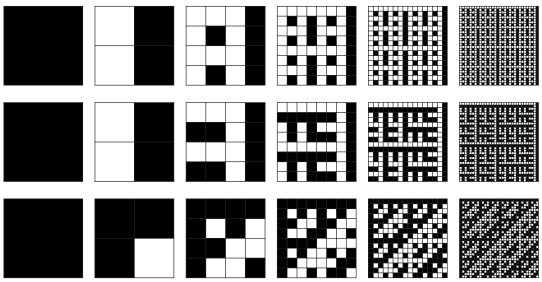
The positions of black squares can be found from:

- `Nest[Flatten[2# /. {x_, y_} → {{x, y}, {x + 1, y}, {x, y + 1}}, 1] &, {{0, 0}}, n]`
- `(Transpose[{Re[#], Im[#]}] &)[Flatten[Nest[{2#, 2# + 1, 2# + ⅈ} &, {0}, n]]]` (compare page 1005)
- `Position[Map[Split, NestList[Sort[Flatten[{#, # + 1}]] &, {0}, 2^n - 1]], _?(OddQ[Length[#]] &), {2}]` (see page 358)
- `Flatten[Table[Map[{t, #} &, Fold[Flatten[{#1, #1 + #2}] &, 0, Flatten[2^(Position[Reverse[IntegerDigits[t, 2]], 1] - 1)]]], {t, 2^n - 1}], 1]` (see page 870)
- `Map[Map[FromDigits[#, 2] &, Transpose[Partition[#, 2]]] &, Position[Nest[{{#, #}, {#}} &, 1, n], 1] - 1]` (see page 509)

A formatting hack giving the same visual pattern is

```
DisplayForm[Nest[SubsuperscriptBox[#, #, #] &, "1", n]]
```

■ **Non-white backgrounds.** The pictures below show substitution systems in which white squares are replaced by blocks which contain black squares. There is still a nested structure but it is usually not visually as obvious as before. (See page 583.)



■ **Higher-dimensional generalizations.** The state of a d-dimensional substitution system can be represented by a nested list of depth d. The evolution of the system for t steps can be obtained from

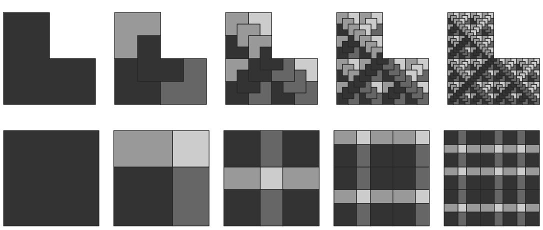
```
SSEvolve[rule_, init_, t_, d_Integer] :=
  Nest[FlattenArray[# /. rule, d] &, init, t]
FlattenArray[list_, d_] :=
  Fold[Function[{a, n}, Map[MapThread[Join, #, n] &,
       a, -{d + 2}]], list, Reverse[Range[d] - 1]]
```

The analog in 3D of the 2D rule on page 187 is

```
{1 → Array[If[LessEqual[##], 0, 1] &, {2, 2, 2}],
  0 → Array[0 &, {2, 2, 2}]}
```

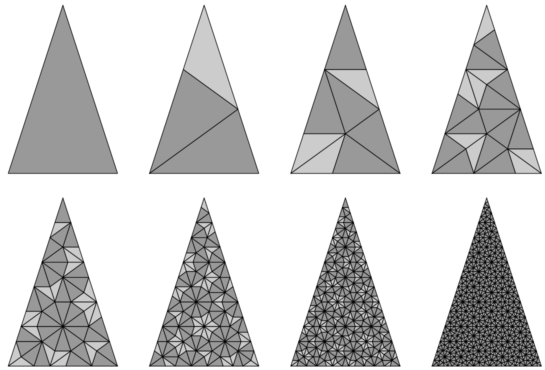
Note that in d dimensions, each black cell must be replaced by at least $d+1$ black cells at each step in order to obtain an object that is not restricted to a dimension $d-1$ hyperplane.

■ **Other shapes.** The systems on pages 187 and 188 are based on subdividing squares into smaller squares. But one can also set up substitution systems that are based on subdividing other geometrical figures, as shown below.



The second example involves two distinct shapes: a square and a *GoldenRatio* aspect ratio rectangle. Labelling each shape and orientation with a different color, the behavior of this system can be reproduced with equal-sized squares using the rule *{3 → {{1, 0}, {3, 2}}, 2 → {{1}, {3}}, 1 → {{3, 2}}, 0 → {{3}}}* starting from initial condition *{{3}}*.

■ **Penrose tilings.** The nested pattern shown below was studied by Roger Penrose in 1974 (see page 943).



The arrangement of triangles at step t can be obtained from a substitution system according to

```
With[{ϕ = GoldenRatio}, Nest[# /. a[p_, q_, r_] ⧴
      With[{s = (p + ϕ q) (2 - ϕ)}, {a[r, s, q], b[r, s, p]}] /.
    b[p_, q_, r_] ⧴ With[{s = (p + ϕ r) (2 - ϕ)}, {a[p, q, s], b[
       r, s, q]}] &, a[{1/2, Sin[2 π/5] ϕ}, {1, 0}, {0, 0}], t]]
```

This pattern can be viewed as generalizations of the pattern generated by the 1D Fibonacci substitution system (c) on page 83. As discussed on page 903, this 1D sequence can be obtained by looking at how a line with *GoldenRatio* slope cuts through a 2D lattice of squares. Penrose tilings can be obtained by looking at how a 2D plane with slopes based on *GoldenRatio* cuts through a lattice of hypercubes in 5D. The tilings turn out to have approximate 5-fold symmetry. (See also page 943.)

In general, projections onto any regular lattice in any number of dimensions from hyperplanes with any quadratic irrational slopes will yield nested patterns that can be generated by subdividing some shape or another according to a substitution system. Despite some confusion in the literature, however, this procedure can reproduce only a tiny fraction of all possible nested patterns.

■ **Page 189 · Dragon curve.** The pattern shown here can be obtained in several related ways, including from numbers in base $i-1$ (see below) and from a doubled version of the paths generated by 1D paperfolding substitution systems (see page 892). Its boundary has fractal dimension *2 Log[2, Root[2 + #1^2 - #1^3, 1]] ≃ 1.52*.

■ **Implementation.** The most convenient approach is to represent each pattern by a list of complex numbers, with the center of each square being given in terms of each complex number z by `{Re[z], Im[z]}`. The pattern after n steps is then given by `Nest[Flatten[f[#]] &, {0}, n]`, where for the rule on page 189 `f[z_] = 1/2 (1 - ⅈ) {z + 1/2, z - 1/2}` (`f[z_] = (1 - ⅈ) {z + 1, z}` gives a transformed version). For the rule on page 190, `f[z_] = 1/2 (1 - ⅈ) {ⅈ z + 1/2, z - 1/2}`. For rules (a), (b) and (c) (Koch curve) on page 191 the forms of `f[z_]` are respectively:

`(0.296 - 0.57 ⅈ) z - 0.067 ⅈ - {1.04, 0.237}`

`N[1/40 {17 (√3 - ⅈ) z, -24 + 14 z}]`

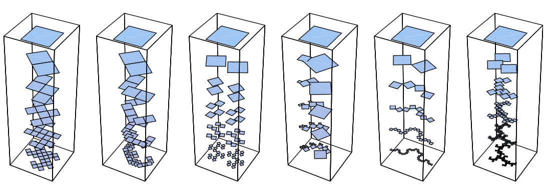
`N[(1/2 (1/√3 - 1) (ⅈ + {1, -1}) - ⅈ - (1 + {ⅈ, -ⅈ}/√3) z)/2]`

■ **Connection with digit sequences.** Patterns after t steps can be viewed as containing all t-digit integers in an appropriate complex base. Thus the patterns on page 189 can be formed from t-digit integers in base $i - 1$ containing only digits 0 and 1, as given by

`Table[FromDigits[IntegerDigits[s, 2, t], ⅈ - 1], {s, 0, 2^t - 1}]`

In the particular case of base $i - q$ with digits 0 through q^2, it turns out that for sufficiently large t any complex integer can be represented, and will therefore be part of the pattern. (Compare page 1094.)

■ **Visualization.** The 3D pictures below show successive steps in the evolution of each of the geometric substitution systems from the main text.

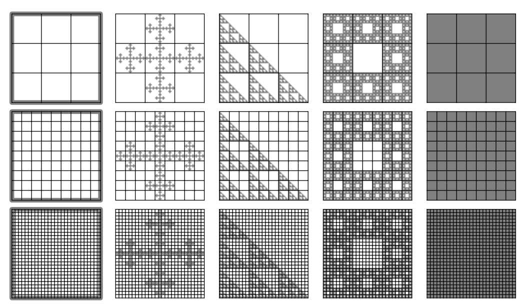


■ **Parameter space sets.** See pages 407 and 1006 for a discussion of varying parameters in geometrical substitution systems.

■ **Affine transformations.** Any set of so-called affine transformations that take the vector for each point, multiply it by a fixed matrix and then add a fixed vector, will yield nested patterns similar to those shown in the main text. Linear operations on complex numbers of the kind discussed above correspond geometrically to rotations, translations and rescalings. General affine transformations also allow reflection and skewing. In addition, affine transformations can readily be generalized to any number of dimensions, while complex numbers represent only two dimensions.

■ **Complex maps.** Many kinds of nonlinear transformations on complex numbers yield nested patterns. Sets of so-called Möbius transformations of the form $z \to (a z + b)/(c z + d)$ always yield such patterns (and correspond to so-called modular groups when $a d - b c == 1$). Transformations of the form `z → {Sqrt[z - c], -Sqrt[z - c]}` yield so-called Julia sets which form nested patterns for many values of c (see note below). In fact, a fair fraction of all possible transformations based on algebraic functions will yield nested patterns. For typically the continuity of such functions implies that only a limited number of shapes not related by limited variations in local magnification can occur at any scale.

■ **Fractal dimensions.** Certain features of nested patterns can be characterized by so-called fractal dimensions. The pictures below show five patterns with three successively finer grids superimposed. The dimension of a pattern can be computed by looking at how the number of grid squares that have any gray in them varies with the length a of the edge of each grid square. In the first case shown, this number varies like $(1/a)^1$ for small a, while in the last case, it varies like $(1/a)^2$. In general, if the number varies like $(1/a)^d$, one can take d to be the dimension of the pattern. And in the intermediate cases shown, it turns out that d has non-integer values.

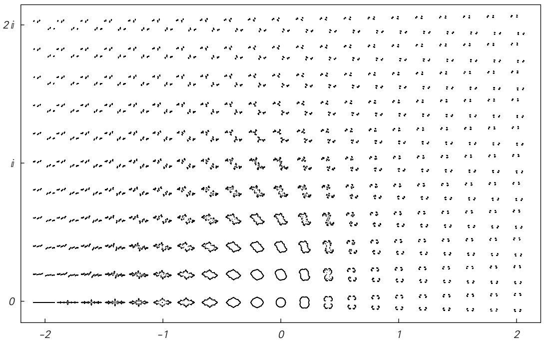


The grid in the pictures above fits over the pattern in a very regular way. But even when this does not happen, the limiting behavior for small a is still $(1/a)^d$ for any nested pattern. This form is inevitable if the underlying pattern effectively has the same structure on all scales. For some of the more complex patterns encountered in this book, however, there continues to be different structure on different scales, so that the effective value of d fluctuates as the scale changes, and may not converge to any definite value. (Precise definitions of dimension based for example on the maximum ever achieved by d will often in general imply formally non-computable values, as in the discussion of page 1138.)

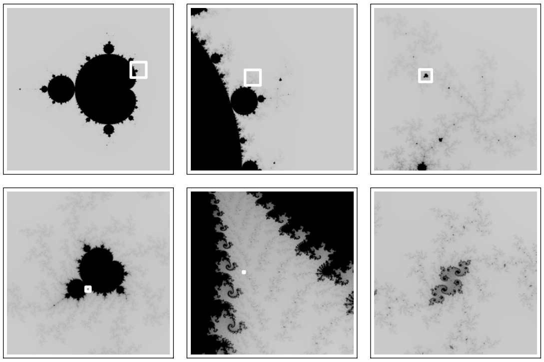
Fractal dimensions characterize some aspects of nested patterns, but patterns with the same dimension can often look very different. One approach to getting better characterizations is to look at each grid square, and to ask not just whether there is any gray in it, but how much. Quantities derived from the mean, variance and other moments of the probability distribution can serve as generalizations of fractal dimension. (Compare page 959.)

▪ **History of fractals.** The idea of using nested 2D shapes in art probably goes back to antiquity; some examples were shown on page 43. In mathematics, nested shapes began to be used at the end of the 1800s, mainly as counterexamples to ideas about continuity that had grown out of work on calculus. The first examples were graphs of functions: the curve on page 918 was discussed by Bernhard Riemann in 1861 and by Karl Weierstrass in 1872. Later came geometrical figures: example (c) on page 191 was introduced by Helge von Koch in 1906, the example on page 187 by Wacław Sierpiński in 1916, examples (a) and (c) on page 188 by Karl Menger in 1926 and the example on page 190 by Paul Lévy in 1937. Similar figures were also produced independently in the 1960s in the course of early experiments with computer graphics, primarily at MIT. From the point of view of mathematics, however, nested shapes tended to be viewed as rare and pathological examples, of no general significance. But the crucial idea that was developed by Benoit Mandelbrot in the late 1960s and early 1970s was that in fact nested shapes can be identified in a great many natural systems and in several branches of mathematics. Using early raster-based computer display technology, Mandelbrot was able to produce striking pictures of what he called fractals. And following the publication of Mandelbrot's 1975 book, interest in fractals increased rapidly. Quantitative comparisons of pure power laws implied by the simplest fractals with observations of natural systems have had somewhat mixed success, leading to the introduction of multifractals with more parameters, but Mandelbrot's general idea of the importance of fractals is now well established in both science and mathematics.

▪ **The Mandelbrot set.** The pictures below show Julia sets produced by the procedure of taking the transformation $z \to \{Sqrt[z-c], -Sqrt[z-c]\}$ discussed above and iterating it starting at $z = 0$ for an array of values of c in the complex plane.

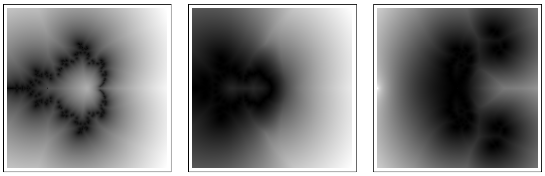


The Mandelbrot set introduced by Benoit Mandelbrot in 1979 is defined as the set of values of c for which such Julia sets are connected. This turns out to be equivalent to the set of values of c for which starting at $z = 0$ the inverse mapping $z \to z^2 + c$ leads only to bounded values of z. The Mandelbrot set turns out to have many intricate features which have been widely reproduced for their aesthetic value, as well as studied by mathematicians. The first picture below shows the overall form of the set; subsequent pictures show successive magnifications of the regions indicated. All parts of the Mandelbrot set are known to be connected. The whole set is not self-similar. However, as seen in the third and fourth pictures, within the set are isolated small copies of the whole set. In addition, as seen in the last picture, near most values of c the boundary of the Mandelbrot set looks very much like the Julia set for that value of c.



On pages 407 and 1006 I discuss parameter space sets that are somewhat analogous to the Mandelbrot set, but whose properties are in many respects much clearer. And from this discussion there emerges the following interpretation of the Mandelbrot set that appears not to be well known but which I find most illuminating. Look at the array of Julia sets and ask for each c whether the Julia set includes the point $z = 0$.

The set of values of c for which it does corresponds exactly to the boundary of the Mandelbrot set. The pictures below show a generalization of this idea, in which gray level indicates the minimum distance *Abs[z - z_0]* of any point z in the Julia set from a fixed point z_0. The first picture shows the case $z_0 = 0$, corresponding to the usual Mandelbrot set.



$z_0 = 0$ $z_0 = 1$ $z_0 = i$

■ **Page 192 · Neighbor-dependent substitution systems.** Given a list of individual replacement rules such as *{{_, 1}, {0, 1}} → {{1, 0}, {1, 1}}*, each step in the evolution shown corresponds to

```
Flatten2D[Partition[list, {2, 2}, 1, -1] /. rule]
```

One can consider rules in which some replacements lead to subdivision of elements but others do not. However, unlike for the 1D case, there will in general in 2D be an arbitrarily large set of different possible neighborhood configurations around any given cell.

■ **Page 192 · Space-filling curves.** One can conveniently scan a finite 2D grid just by going along each successive row in turn. One can scan a quadrant of an infinite grid using the σ function on page 1127, or one can scan a whole grid by for example going in a square spiral that at step t reaches position

```
(1/2 (-1)^# ({1, -1} (Abs[#^2 - t] - #) + #^2 - t - Mod[#, 2]) &)[
  Round[√t ]]
```

Network Systems

■ **Implementation.** The nodes in a network system can conveniently be labelled by numbers *1, 2, … n*, and the network obtained at a particular step can be represented by a list of pairs, where the pair at position i gives the numbers corresponding to the nodes reached by following the above and below connections from node i. With this setup, a network consisting of just one node is *{{1, 1}}* and a 1D array of n nodes can be obtained with

```
CyclicNet[n_] := RotateRight[
   Table[Mod[{i - 1, i + 1}, n] + 1, {i, n}]]
```

With above connections represented as *1* and the below connections as *2*, the node reached by following a succession s of connections from node i is given by

```
Follow[list_, i_, s_List] := Fold[list[[#1]][[#2]] &, i, s]
```

The total number of distinct nodes reached by following all possible succession of connections up to length d is given by

```
NeighborNumbers[list_, i_Integer, d_Integer] :=
   Map[Length, NestList[Union[Flatten[list[[#]]]] &,
        Union[list[[i]]], d - 1]]
```

For each such list the rules for the network system then specify how the connections from node i should be rerouted. The rule *{2, 3} → {{2, 1}, {1}}* specifies that when *NeighborNumbers* gives *{2, 3}* for a node i, the connections from that node should become *{Follow[list, i, {2, 1}], Follow[list, i, {1}]}*. The rule *{2, 3} → {{{2, 1}, {1, 1}}, {1}}* specifies that a new node should be inserted in the above connection, and this new node should have connections *{Follow[list, i, {2, 1}], Follow[list, i, {1, 1}]}*. With rules set up in this way, each step in the evolution of a network system is given by

```
NetEvolveStep[{depth_Integer, rule_List}, list_List] := Block[
  {new = {}}, Join[Table[Map[NetEvolveStep1[#, list, i] &,
     Replace[NeighborNumbers[list, i, depth],
      rule]], {i, Length[list]}], new]]

NetEvolveStep1[s : {___Integer}, list_, i_] := Follow[list, i, s]

NetEvolveStep1[{s1 : {___Integer}, s2 : {___Integer}},
   list_, i_] := Length[list] + Length[
     AppendTo[new, {Follow[list, i, s1], Follow[list, i, s2]}]]
```

The set of nodes that can be reached from node i is given by

```
ConnectedNodes[list_, i_] :=
  FixedPoint[Union[Flatten[{#, list[[#]]}]] &, {i}]
```

and disconnected nodes can be removed using

```
RenumberNodes[list_, seq_] :=
  Map[Position[seq, #][[1, 1]] &, list[[seq]], {2}]
```

The sequence of networks obtained on successive steps by applying the rules and then removing all nodes not connected to node number *1* is given by

```
NetEvolveList[rule_, init_, t_Integer] :=
  NestList[(RenumberNodes[#, ConnectedNodes[#, 1]] &)[
      NetEvolveStep[rule, #]] &, init, t]
```

Note that the nodes in each network are not necessarily numbered in the order that they appear on successive lines in the pictures in the main text. Additional information on the origin of each new node must be maintained if this order is to be found.

■ **Rule structure.** For depth 1, the possible results from *NeighborNumbers* are *{1}* and *{2}*. For depth 2, they are *{1, 1}*, *{1, 2}*, *{2, 1}*, *{2, 2}*, *{2, 3}* and *{2, 4}*. In general, each successive element in a list from *NeighborNumbers* cannot be more than twice the previous element.

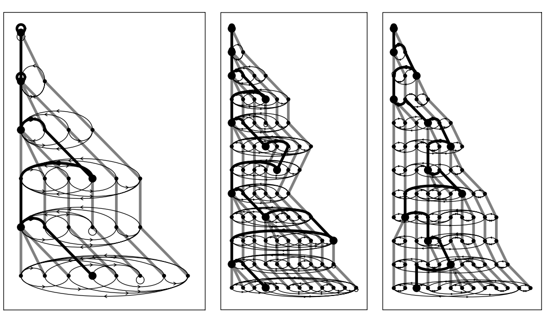
■ **Undirected networks.** Networks with connections that do not have definite directions are discussed at length in Chapter 9, mainly as potential models for space in the universe. The rules for updating such networks turn out to be somewhat more difficult to apply than those for the network systems discussed here.

■ **Page 199 · Computer science.** The networks discussed here can be thought of as very simple analogs of data structures in practical computer programs. The connections correspond to pointers between elements of these data structures. The fact that there are two connections coming from each node is reminiscent of the LISP language, but in the networks considered here there are no leaves corresponding to atoms in LISP. Note that the process of dropping nodes that become disconnected is analogous to so-called "garbage collection" for data structures. The networks considered here are also related to the combinator systems discussed on page 1121.

■ **Page 202 · Properties.** Random behavior seems to occur in a few out of every thousand randomly selected rules of the kind shown here. In case (c), the following gives a list of the numbers of nodes generated up to step *t*:

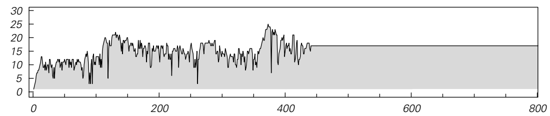
```
FoldList[Plus, 1, Join[{1, 4, 12, 10, -20, 6, 4},
    Map[d, IntegerDigits[Range[4, t - 5], 2]]]]
d[{___, 1}] = 1
d[{1, p : ((0) ..), 0}] :=
  -Apply[Plus, 4 Range[Length[{p}]] - 1] + 6
d[{__, 1, p : ((0) ..), 0}] := d[{1, p, 0}] - 7
d[{___, p : ((1) ..), q : ((0) ...), 1, 0}] :=
  4 Length[{p}] + 3 Length[{q}] + 2
d[{___, p : ((1) ..), 1, 0}] := 4 Length[{p}] + 2
```

■ **Sequential network systems.** In the network systems discussed in the main text, every node is updated in parallel at each step. It is however also possible to consider systems in which there is only a single active node, and operations are performed only on that node at any particular step. The active node can move by following its above or below connections, in a way that is determined by a rule which depends on the local structure of the network. The pictures below show examples of sequential network systems; the path of the active node is indicated by a thick black line.

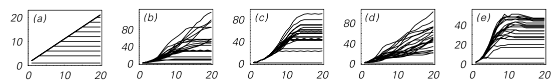


It is rather common for the active node eventually to get stuck at a particular position in the network; the picture below shows the effect of this on the total number of nodes in the last case illustrated above. The rule for this system is

```
{{1, 1}→{{{{}, {1, 1}}, {2}}, 2}, {1, 2}→{{{2, 2}, {{}, {2, 2}}}, 2},
  {2, 1}→{{{}, {2, 2}}, 2}, {2, 2}→{{{1, 2}, {{1}, {2}}}, 1},
  {2, 3}→{{{{1, 2}, {1}}, {{2}, {2, 1}}}, 2},
  {2, 4}→{{{2, 2}, {{2, 1}, {}}}, 1}}
```



■ **Dimensionality of networks.** As discussed on page 479, if a sufficiently large network has a d-dimensional form, then by following r connections in succession from a given node, one should reach about r^d distinct nodes. The plots below show the actual numbers of nodes reached as a function of r for the systems on pages 202 and 203 at steps 1, 10, 20, ..., 200.



■ **Cellular automata on networks.** The cellular automata that we have considered so far all have cells arranged in regular arrays. But one can also set up generalizations in which the cells correspond to nodes in arbitrary networks. Given a network of the kind discussed in the main text of this section, one can assign a color to each node, and then update this color at each step according to a rule that depends on the colors of the nodes to which the connections from that node go. The behavior obtained depends greatly on the form of the network, but with networks of finite size the results are typically like those obtained for other finite size cellular automata of the kind discussed on page 259.

■ **Implementation.** Given a network represented as a list in which element i is $\{a, i, b\}$, where a is the node reached by the above connection from node i, and b is the node reached by the below connection, each step corresponds to

```
NetCAStep[{rule_, net_}, list_] :=
  Map[Replace[#, rule] &, list[[net]]]
```

■ **Boolean networks.** Several lines of development from the cybernetics movement (notably in immunology, genetics and management science) led in the 1960s to a study of random Boolean networks—notably by Stuart Kauffman and Crayton Walker. Such systems are like cellular automata on networks, except for the fact that when they are set up each node has a rule that is randomly chosen from all 2^{2^s} possible ones with s inputs. With $s = 2$ class 2 behavior (see Chapter 6) tends to dominate. But for $s > 2$, the behavior one sees quickly approaches what is typical for a random mapping in which the network representing the evolution of the 2^m states of the m underlying nodes is itself connected essentially randomly (see page 963). (Attempts were made in the 1980s to study phase transitions as a function of s in analogy to ones in percolation and spin glasses.) Note that in almost all work on random Boolean networks averages are in effect taken over possible configurations, making it impossible to see anything like the kind of complex behavior that I discuss in cellular automata and many other systems in this book.

Multiway Systems

▪ **Implementation.** It is convenient to represent the state of a multiway system at each step by a list of strings, where an individual string is for example *"ABBAAB"*. The rules for the multiway system can then be given for example as

```
{"AAB" → " BB", " BA" → " ABB"}
```

The evolution of the system is given by the functions

```
MWStep[rule_List, slist_List] := Union[Flatten[
      Map[Function[s, Map[MWStep1[#, s] &, rule]], slist]]]
MWStep1[p_String → q_String, s_String] :=
   Map[StringReplacePart[s, q, #] &, StringPosition[s, p]]
MWEvolveList[rule_, init_List, t_Integer] :=
   NestList[MWStep[rule, #] &, init, t]
```

An alternative approach uses lists instead of strings, and in effect works by tracing the internal steps that *Mathematica* goes through in trying out possible matchings. With the rule from above written as

```
{{x___, 0, 0, 1, y___} → {x, 1, 1, y},
   {x___, 1, 0, y___} → {x, 0, 1, 1, y}}
```

MWStep can be rewritten as

```
MWStep[rule_List, slist_List] :=
   Union[Flatten[Map[ReplaceList[#, rule] &, slist], 1]]
```

The case shown on page 206 is

```
{"AB" → "", "ABA" → "ABBAB", "ABABBB" → "AAAAABA"}
```

starting with *{"ABABAB"}*. Note that the rules are set up so that a string for which there are no applicable replacements at a given step is simply dropped.

▪ **General properties.** The merging of states (as done above by *Union*) is crucial to the behavior seen. Note that the pictures shown indicate only which states yield which states—not for example in how many ways the rules can be applied to a given state to yield a given new state.

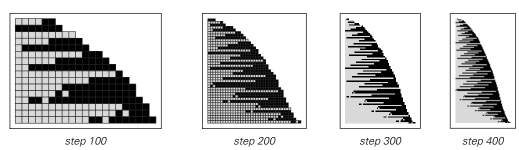
If there was no merging, then if a typical state yielded more than one new state, then inevitably the total number of states would increase exponentially. But when there is merging, this need not occur—making it difficult to give probabilistic estimates of growth rates. Note that a given rule can yield very different growth rates with different initial conditions. Thus, for example, the growth rate for *{"A" → "AA", "AB" → "BA", "BA" → "AB"}* is t^{n+1}, where n is the number of initial *B*'s. With most rules, states that appear at one step can disappear at later steps. But if *"A" → "A"* and its analogs are part of the rule, then every state will always be kept, almost inevitably leading to overall nesting in pictures like those on page 208.

In cases where all strings that appear both in rules and initial conditions are sorted—so that for example *A*'s appear before *B*'s—any string generated will also be sorted, so it can be specified just by giving a list of how many *A*'s and how many *B*'s appear in it. The rule for the system can then be stated in terms of a difference vector—which for *{"BA" → "AAA", "BAA" → "BBBA"}* is *{{2, -1}, {-1, 2}}*. Given a list of string specifications, a step in the evolution of the multiway system corresponds to

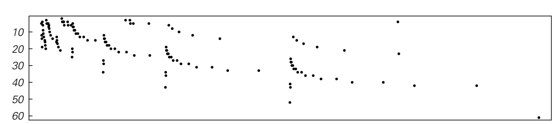
```
Select[Union[Flatten[Outer[Plus, diff, list, 1], 1]],
   Abs[#] == # &]
```

▪ **Page 206 · Properties.** The total number of strings grows approximately quadratically; its differences repeat (offset by 1) with period 1071. The number of new strings generated at successive steps grows approximately linearly; its differences repeat with period 21. The third element of the rule is at first used only on some steps—but after step 50 it appears to be used somewhere in every step.

The pictures below show in stacked form (as on page 208) all sequences generated at various steps of evolution. Note that after just a few steps, the sequences produced always seem to consist of white elements followed by black, with possibly one block of black in the white region. Without this additional block of black, only the first case in the rule can ever apply.



In analogy with page 796 the picture below shows when different strings with lengths up to 10 are reached in the evolution of the system.



Different initial conditions for this multiway system lead to behavior that either dies out (as for *"ABA"*), or grows exponentially forever (as for *"ABAABABA"*).

▪ **Frequency of behavior.** Among multiway systems with randomly chosen rules, one finds about equal numbers that grow rapidly and die out completely. A few percent exhibit repetitive behavior, while only one in several million exhibit more complex behavior. One common form of more complex behavior is quadratic growth, with essentially periodic fluctuations superimposed—as on page 206.

▪ **History.** Versions of multiway systems have been invented many times in a variety of contexts. In mathematics specific examples of them arose in formal group theory (see below) around the end of the 1800s. Axel Thue considered versions with two-way rules (analogous to semigroups, as discussed below) in 1912, leading to the name semi-Thue systems sometimes being used for general multiway systems. Other names for multiway systems have included string and term rewrite systems, production systems and associative calculi. From the early 1900s various generalizations of multiway systems were used as idealizations of mathematical proofs (see page 1150); multiway systems with explicit pattern variables (such as *s_*) were studied under the name canonical systems by Emil Post starting in the 1920s. Since the 1950s, multiway systems have been widely used as generators of formal languages (see below). Simple analogs of multiway systems have also been used in genetic analysis in biology and in models for particle showers and other branching processes in physics and elsewhere.

▪ **Semigroups and groups.** The multiway systems that I discuss can be viewed as representations for generalized versions of familiar mathematical structures. Semigroups are obtained by requiring that rules come in pairs: with each rule such as *"ABB"* → *"BA"* there must also be the reversed rule *"BA"* → *"ABB"*. Such pairs of rules correspond to relations in the semigroup, specifying for example that *"ABB"* is equivalent to *"BA"*. (The operation in the semigroup is concatenation of strings; *""* acts as an identity element, so in fact a monoid is always obtained.) Groups require that not only rules but also symbols come in pairs. Thus, for example, in addition to a symbol *A*, there must be an inverse symbol *a*, with the rules *"Aa"* → *""*, *"aA"* → *""* and their reversals.

In the usual mathematical approach, the objects of greatest interest for many purposes are those collections of sequences that cannot be transformed into each other by any of the rules given. Such collections correspond to distinct elements of the group or semigroup, and in general many different choices of underlying rules may yield the same elements with the same properties. In terms of multiway systems, each of the elements corresponds to a disconnected part of the network formed from all possible sequences.

Given a particular representation of a group or semigroup in terms of rules for a multiway system, an object that is often useful is the so-called Cayley graph—a network where each node is an element of the group, and the connections show what elements are reached by appending each possible symbol to the sequences that represent a given element. The so-called free semigroup has no relations and thus no rules, so that all strings of generators correspond to distinct elements, and the Cayley graph is a tree like the ones shown on page 196. The simplest non-trivial commutative semigroup has rules *"AB"* → *"BA"* and *"BA"* → *"AB"*, so that strings of generators with *A*'s and *B*'s in different orders are equivalent and the Cayley graph is a 2D grid.

For some sets of underlying rules, the total number of distinct elements in a group or semigroup is finite. (Compare page 945.) A major mathematical achievement in the 1980s was the complete classification of all possible so-called simple finite groups that in effect have no factors. (For semigroups no such classification has yet been made.) In each case, there are many different choices of rules that yield the same group (and similar Cayley graphs). And it is known that even fairly simple sets of rules can yield large and complicated groups. The icosahedral group A_5 defined by the rules $x^2 == y^3 == (x\,y)^5 == 1$ has 60 elements. But in the most complicated case a dozen rules yield the Monster Group, where the number of elements is

808017424794512875886459904961710757005754368000000000

(See also pages 945 and 1032.)

Following work in the 1980s and 1990s by Mikhael Gromov and others, it is also known that for groups with randomly chosen underlying rules, the Cayley graph is usually either finite, or has a rapidly branching tree-like structure. But there are presumably also marginal cases that exhibit complex behavior analogous to what we saw in the main text. And indeed for example, despite conjectures to the contrary, it was found in the 1980s by Rostislav Grigorchuk that complicated groups could be constructed in which growth intermediate between polynomial and exponential can occur. (Note that different choices of generators can yield Cayley graphs with different local subgraphs; but the overall structure of a sufficiently large graph for a particular group is always the same.)

▪ **Formal languages.** The multiway systems that I discuss are similar to so-called generative grammars in the theory of formal languages. The idea of a generative grammar is that all possible expressions in a particular formal language can be produced by applying in all possible ways the set of replacement rules given by the grammar. Thus, for example, the rules {"x" → "xx", "x" → "(x)", "x" → "()"} starting with "x" will generate all expressions that consist of balanced sequences of parentheses. (Final expressions correspond to those without the "non-terminal" symbol *x*.) The hierarchy described by Noam Chomsky in 1956 distinguishes four kinds of generative grammars (see page 1104):

Regular grammars. The left-hand side of each rule must consist of one non-terminal symbol, and the right-hand side can contain only one non-terminal symbol. An example is {"x" → "xA", "x" → "yB", "y" → "xA"} starting with "x" which generates sequences in which no pair of *B*'s ever appear together. Expressions in regular languages can be recognized by finite automata of the kind discussed on page 957.

Context-free grammars. The left-hand side of each rule must consist of one non-terminal symbol, but the right-hand side can contain several non-terminal symbols. Examples include the parenthesis language mentioned above, {"x" → "AxA", "x" → "B"} starting with "x", and the syntactic definitions of *Mathematica* and most other modern computer languages. Context-free languages can be recognized by a computer using only memory on a single last-in first-out stack. (See pages 1091 and 1103.)

Context-sensitive grammars. The left-hand side of each rule is no longer than the right, but is otherwise unrestricted. An example is {"Ax" → "AAxx", "xA" → "BAA", "xB" → "Bx"} starting with "AAxBA", which generates expressions of the form Table["A", {n}] <> Table["B", {n}] <> Table["A", {n}].

Unrestricted grammars. Any rules are allowed.

(See also page 944.)

▪ **Multidimensional multiway systems.** As a generalization of multiway systems based on 1D strings one can consider systems in which rules operate on arbitrary blocks of elements in an array in any number of dimensions. Still more general network substitution systems are discussed on page 508.

▪ **Limited size versions.** One can set up multiway systems of limited size by applying transformations cyclically to strings.

▪ **Multiway tag systems.** See page 1141.

▪ **Multiway systems based on numbers.** One can consider for example the rule n → {n + 1, 2 n} implemented by

```
NestList[Union[Flatten[{# + 1, 2 #}]] &, {0}, t]
```

In this case there are Fibonacci[t + 2] distinct numbers obtained at step *t*. In general, rules based on simple arithmetic operations yield only simple nested structures. If the numbers *n* are allowed to have both real and imaginary parts then results analogous to those discussed for substitution systems on page 933 are obtained. (Somewhat related systems based on recursive sequences are discussed on page 907. Compare also sorted multiway systems on page 937.)

▪ **Non-deterministic systems.** Multiway systems are examples of what are often in computer science called non-deterministic systems. The general idea of a non-deterministic system is to have rules with several possible outcomes, and then to allow each of these outcomes to be followed. Non-deterministic Turing machines are a common example. For most types of systems (such as Turing machines) such non-deterministic versions do not ultimately allow any greater range of computations to be performed than deterministic ones. (But see page 766.)

▪ **Fundamental physics.** See page 504.

▪ **Game systems.** One can think of positions or configurations in a game as corresponding to nodes in a large network, and the possible moves in the game as corresponding to connections between nodes. Most games have rules which imply that if certain states are reached one player can be forced in the end to lose, regardless of what specific moves they make. And even though the underlying rules in the game may be simple, the pattern of such winning positions is often quite complex. Most games have huge networks whose structure is difficult to visualize (even the network for tic-tac-toe, for example, has 5478 nodes). One example that allows easy visualization is a simplification of several common games known as nim. This has *k* piles of objects, and on alternate steps each of two players takes as many objects as they want from any one of the piles. The winner is the player who manages to take the very last object. With just two piles one player can force the other to lose by arranging that after each of their moves the two piles have equal heights. With more than two piles it was discovered in 1901 that one player can in general force the other to lose by arranging that after each of their moves Apply[BitXor, h] == 0, where *h* is the list of heights. For $k > 1$ this yields a nested pattern, analogous to those shown on page 871. If one allows only specific numbers of objects to be taken at each step a nested pattern is again obtained. With more general rules it seems almost inevitable that much more complicated patterns will occur.

Systems Based on Constraints

■ **The notion of equations.** In the mathematical framework traditionally used in the exact sciences, laws of nature are usually represented not by explicit rules for evolution, but rather by abstract equations. And in general what such equations do is to specify constraints that systems must satisfy. Sometimes these constraints just relate the state of a system at one time to its state at a previous time. And in such cases, the constraints can usually be converted into explicit evolution rules. But if the constraints relate different features of a system at one particular time, then they cannot be converted into evolution rules. In computer programs and other kinds of discrete systems, explicit evolution rules and implicit constraints usually work very differently. But in traditional continuous mathematics, it turns out that these differences are somewhat obscured. First of all, at a formal level, equations corresponding to these two cases can look very similar. And secondly, the equations are almost always so difficult to deal with at all that distinctions between the two cases are not readily noticed.

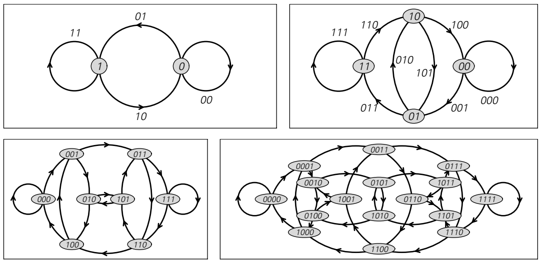
In the language of differential equations—the most widely used models in traditional science—the two cases we are discussing are essentially so-called initial value and boundary value problems, discussed on page 923. And at a formal level, the two cases are so similar that in studying partial differential equations one often starts with an equation, and only later tries to work out whether initial or boundary values are needed in order to get either any solution or a unique solution. For the specific case of second-order equations, it is known in general what is needed. Elliptic equations such as the Laplace equation need boundary values, while hyperbolic and parabolic equations such as the wave equation and diffusion equation need initial values. But for higher-order equations it can be extremely difficult to work out what initial or boundary values are needed, and indeed this has been the subject of much research for many decades.

Given a partial differential equation with initial or boundary values, there is then the question of solving it. To do this on a computer requires constructing a discrete approximation. But it turns out that the standard methods used (such as finite difference and finite element) involve extremely similar computations for initial and for boundary value problems, leaving no trace of the significant differences between these cases that are so obvious in the discrete systems that we discuss in most of this book.

■ **Linear and nonlinear systems.** A vast number of different applications of traditional mathematics are ultimately based on linear equations of the form $u == m . v$ where u and v are vectors (lists) and m is a matrix (list of lists), all containing ordinary continuous numbers. If v is known then such equations in essence provide explicit rules for computing u. But if only u is known, then the equations can instead be thought of as providing implicit constraints for v. However, it so happens that even in this case v can still be found fairly straightforwardly using *LinearSolve[m, u]*. With vectors of length n it generically takes about n^2 steps to compute u given v, and a little less than n^3 steps to compute v given u (the best known algorithms—which are based on matrix multiplication—currently involve about $n^{2.4}$ steps). But as soon as the original equation is nonlinear, say $u == m_1 . v + m_2 . v^2$, the situation changes dramatically. It still takes only about n^2 steps to compute u given v, but it becomes vastly more difficult to compute v given u, taking perhaps 2^{2^n} steps. (Generically there are 2^n solutions for v, and even for integer coefficients in the range $-r$ to $+r$ already in 95% of cases there are 4 solutions with $n = 2$ as soon as $r \geq 6$.)

■ **Explanations based on constraints.** In some areas of science it is common to give explanations in terms of constraints rather than mechanisms. Thus, for example, in physics there are so-called variational principles which state that physical systems will behave in ways that minimize or maximize certain quantities. One such principle implies that atoms in molecules will tend to arrange themselves so as to minimize their energy. For simple molecules, this is a useful principle. But for complicated molecules of the kind that are common in living systems, this principle becomes much less useful. In fact, in finding out what configuration such molecules actually adopt, it is usually much more relevant to know how the molecule evolves in time as it is created than which of its configurations formally has minimum energy. (See pages 342 and 1185.)

■ **Page 211 · 1D constraints.** The constraints in the main text can be thought of as specifying that only some of the k^n possible blocks of cells of length n (with k possible colors for each cell) are allowed. To see the consequences of such constraints consider breaking a sequence of colors into blocks of length n, with each block overlapping by $n - 1$ cells with its predecessor, as in *Partition[list, n, 1]*. If all possible sequences of colors were allowed, then there would be k possibilities for what block could follow a given block, given by *Map[Rest, Table[Append[list, i], {i, 0, k - 1}]]*. The possible sequences of length n blocks that can occur are conveniently represented by possible paths by so-called de Bruijn networks, of the kind shown for $k = 2$ and $n = 2$ through 5 below.



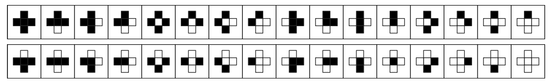
Given the network for a particular n, it is straightforward to see what happens when only certain length n blocks are allowed: one just keeps the arcs in the network that correspond to allowed blocks, and drops all other ones. Then if one can still form an infinite path by going along the arcs that remain, this path will correspond to a pattern that satisfies the constraints. Sometimes there will be a unique such path; in other cases there will be choices that can be made along the path. But the crucial point is that since there are only k^{n-1} nodes in the network, then if any infinite path is possible, there must be such a path that visits the same node and thus repeats itself after at most k^{n-1} cells. The constraint on page 210 has $k = 2$ and $n = 3$; the pattern that satisfies it repeats with period 4, thus saturating the bound. (See also page 266.)

■ **1D cellular automata.** In a cellular automaton with k colors and r neighbors, configurations that are left invariant after t steps of evolution according to the cellular automaton rule are exactly the ones which contain only those length $2r + 1$ blocks in which the center cell is the same before and after the evolution. Such configurations therefore obey constraints of the kind discussed in the main text. As we will see on page 225 some cellular automata evolve to invariant configurations from any initial conditions, but most do not. (See page 954.)

■ **Dynamical systems theory.** Sets of sequences in which a finite collection of blocks are excluded are sometimes known as finite complement languages, or subshifts of finite type. (See page 958.)

■ **Page 215 · 2D constraints.** The constraints shown here are minimal, in the sense that in each case removing any of the allowed templates prevents the constraint from ever being satisfied. Note that constraints which differ only by overall rotation, reflection or interchange of black and white are not explicitly shown. The number of allowed templates out of the total of 32 possible varies from 1 to 15 for the constraints shown, with 12 being the most common. Smaller sets of allowed templates typically seem to lead to constraints that can be satisfied by visually simpler patterns.

■ **Numbering scheme.** The constraint numbered n allows the templates at *Position[IntegerDigits[n, 2, 32], 1]* in the list below. (See also page 927.)



■ **Identifying the 171 patterns.** The number of constraints to consider can be reduced by symmetries, by discarding sets of templates that are supersets of ones already known to be satisfiable, and by requiring that each template in the set be compatible with itself or with at least one other in each of the eight immediately adjacent positions. The remaining constraints can then be analyzed by attempting to build up explicit patterns that satisfy them, as discussed below.

■ **Checking constraints.** A set of allowed templates can be specified by a *Mathematica* pattern of the form $t_1 \,|\, t_2 \,|\, t_3$ etc. where the t_i are for example *{{_, 1, _}, {0, 0, 1}, {_, 0, _}}*. To check whether an array *list* contains only arrangements of colors corresponding to allowed templates one can then use

```
SatisfiedQ[list_, allowed_] :=
  Apply[And, Map[MatchQ[#, allowed] &,
     Partition[list, {3, 3}, {1, 1}], {2}], {0, 1}]
```

■ **Representing repetitive patterns.** Repetitive patterns are often most conveniently represented as tessellations of rectangles whose corners overlap. Pattern (a) on page 213 can be specified as

```
{{2, -1, 2, 3}, {{0, 0, 0, 0}, {1, 1, 0, 0}, {1, 0, 0, 0}}}
```

Given this, a complete *nx* by *ny* array filled with this pattern can be constructed from

```
c[{d1_, d2_, d3_, d4_}, {x_, y_}] :=
  With[{d = d1 d2 + d1 d4 + d3 d4},
    Mod[{{d2 x + d4 x + d3 y, d4 x - d1 y}}/d, 1]]
Fill[{dlist_, data_}, {nx_, ny_}] :=
 Array[c[dlist, {##}] &, {nx, ny}] /. Flatten[MapIndexed[
    c[dlist, Reverse[#2]] → #1 &, Reverse[data], {2}], 1]
```

■ **Searching for patterns.** The basic approach to finding a pattern which satisfies a particular constraint on an infinite array of cells is to start with a pattern which satisfies the constraint in a small region, and then to try to extend the pattern. Often the constraint will immediately force a unique extension of the pattern, at least for some distance. But eventually there will normally be places where the pattern is not yet uniquely determined, and so a series of choices have to be made. The procedure used to find the results in this book attempts to extend patterns along a square spiral, making whatever choices are needed, and backtracking if these turn out to be inconsistent with the constraint. At every step in the procedure, regularities are tested for that would imply the possibility of an infinite repetitive pattern. In addition, whenever there is a choice, the first cases to be tried are set up to be ones that tend to extend whatever regularity has developed so far. And when backtracking is needed, the procedure always goes back to the most recent choice that actually affected whatever inconsistency was discovered. And in addition it remembers what has already been worked out, so as to avoid, for example, unnecessarily working out the pattern on the opposite side of the spiral again.

▪ **Undecidability.** The general problem of whether an infinite pattern exists that satisfies a particular constraint is formally undecidable (see page 1139). This means that in general there can be no upper bound on the size of region for which the constraints can be satisfied, even if they are not satisfiable for the complete infinite grid.

▪ **NP completeness.** The problem of whether a pattern can be found that satisfies a constraint even in a finite region is NP-complete. (See page 1145.) This suggests that to determine whether a repetitive pattern with repeating blocks of size *n* exists may in general take a number of steps which grows more rapidly than any polynomial in *n*.

▪ **Enumerating patterns.** Compare page 959.

▪ **Page 219 · Non-periodic pattern.** The color at position *x*,*y* in the pattern is given by

```
a[x_, y_] := Mod[y + 1, 2] /; x + y > 0
a[x_, y_] := 0 /; Mod[x + y, 2] == 1
a[x_, y_] :=
  Mod[Floor[(x - y) 2^((x+y-6)/4)], 2] /; Mod[x + y, 4] == 2
a[x_, y_] := 1 - Sign[Mod[x - y + 2, 2^((-x-y+8)/4)]]
```

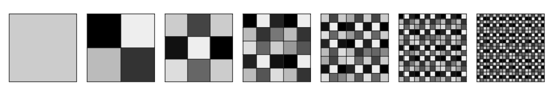
The origin of the *x*,*y* coordinates is the only freedom in this pattern. The nested structure is like the progression of base 2 digit sequences shown on page 117. Negative numbers are effectively represented by complements of digit sequences, much as in typical practical computers. With the procedure described above for finding patterns that satisfy a constraint, generating the pattern shown here is straightforward once the appropriate constraint is identified.

▪ **Other types of constraints.** Constraints based on smaller templates simply require smaller numbers of repetitive patterns: :4; :7; :17; :11; :12. To extend the class of systems considered in the main text, one can increase the size of the templates, or increase the number of possible colors for each cell. For 3×3 templates with two colors extensive randomized searches have failed to discover examples where non-repetitive patterns are forced to occur. Another extension of the constraints in the main text is to require that not just a single template, but every template in the set, must occur somewhere in the pattern. Searches of such systems have also failed to discover examples of forced non-repetitive patterns beyond the one shown in the text.

▪ **Forcing nested patterns.** It is straightforward to find constraints that allow nested patterns; the challenge is to find ones that force such patterns to occur. Many nested patterns (such as the one made by rule 90, for example) contain large areas of uniform white, and it is typically difficult to prevent pure repetition of that area. One approach to finding constraints that can be satisfied only by nested patterns is nevertheless to start from specific nested patterns, look at what templates occur, and then see whether these templates are such that they do not allow any purely repetitive patterns. A convenient way to generate a large class of nested patterns is to use 2D substitution systems of the kind discussed on page 188. But searching all 4 billion or so possible such systems with 2×2 blocks and up to four colors one finds not a single case in which a nested pattern is forced to occur. It can nevertheless be shown that with a sufficiently large number of extra colors any nested pattern can be forced to occur. And it turns out that a result from the mid-1970s by Robert Ammann for a related problem of tiling (see below) allows one to construct a specific system with 16 colors in which constraints of the kind discussed here force a nested pattern to occur. One starts from the substitution system with rules

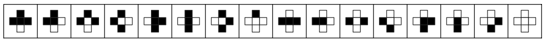
```
{1 → {{3}}, 2 → {{13, 1}, {4, 10}}, 3 → {{15, 1}, {4, 12}},
  4 → {{14, 1}, {2, 9}}, 5 → {{13, 1}, {4, 12}}, 6 → {{13, 1}, {8, 9}},
  7 → {{15, 1}, {4, 10}}, 8 → {{14, 1}, {6, 10}}, 9 → {{14}, {2}},
  10 → {{16}, {7}}, 11 → {{13}, {8}}, 12 → {{16}, {3}},
  13 → {{5, 11}}, 14 → {{2, 9}}, 15 → {{3, 11}}, 16 → {{6, 10}}}
```

This yields the nested pattern below which contains only 51 of the 65,536 possible 2×2 blocks of cells with 16 colors. It then turns out that with the constraint that the only 2×2 arrangements of colors that can occur are ones that match these 51 blocks, one is forced to get the nested pattern below.



▪ **Relation to 2D cellular automata.** The kind of constraints discussed are exactly those that must be satisfied by configurations that remain unchanged in the evolution of a 2D cellular automaton. The argument for this is similar to the one on pages 941 and 954 for 1D cellular automata. The point is that of the 32 5-cell neighborhoods involved in the 2D cellular automaton rule, only some subset will have the property that the center cell remains unchanged after applying the rule. And any configuration which does not change must involve only these subsets. Using the results of this section it then follows that in the evolution of all 2D cellular automata of the type discussed on page 170 there exist purely repetitive configurations that remain unchanged.

■ Relation to 1D cellular automata. A picture that shows the evolution of a 1D cellular automaton can be thought of as a 2D array of cells in which the color of each cell satisfies a constraint that relates it to the cells above according to the cellular automaton rule. This constraint can then be represented in terms of a set of allowed templates; the set for rule 30 is as follows:



To reproduce an ordinary picture of cellular automaton evolution, one would have to specify in advance a whole line of black and white cells. Below this line there would then be a unique pattern corresponding to the application of the cellular automaton rule. But above the line, except for reversible rules, there is no guarantee that any pattern satisfying the constraints can exist.

If one specifies no cells in advance, or at most a few cells, as in the systems discussed in the main text, then the issue is different, however. And now it is always possible to construct a repetitive pattern which satisfies the constraints simply by finding repetitive behavior in the evolution of the cellular automaton from a spatially repetitive initial condition.

■ Non-computable patterns. It is known to be possible to set up constraints that will force patterns in which finding the color of a particular cell can require doing something like solving a halting problem—which cannot in general be done by any finite computation. (See also page 1139.)

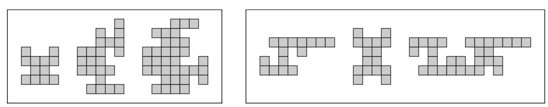
■ Tiling. The constraints discussed here are similar to those encountered in covering the plane with tiles of various shapes. Of regular polygons, only squares, triangles and hexagons can be used to do this, and in these cases the tilings are always repetitive. For some time it was believed that any set of tiles that could cover the plane could be arranged to do so repetitively. But in 1964 Robert Berger demonstrated that this was not the case, and constructed a set of about 20,000 tiles that could cover the plane only in a nested fashion. Later Berger reduced the number of tiles needed to 104. Then Raphael Robinson in 1971 reduced the number tiles to six, and in 1974 Roger Penrose showed that just two tiles were necessary. Penrose's tiles can cover the plane only in a nested pattern that can be constructed from a substitution system that successively subdivides each tile, as shown on page 932. (Note that various dissections of these tiles can also be used. The edges of the particular shapes shown should strictly be distinguished in order to prevent trivial periodic arrangements.) The triangles in the construction have angles which are multiples of $\pi/5$, so that the whole tiling has an approximate 5-fold symmetry (see page 994). Repetitive tilings of the plane can only have 3-, 4- or 6-fold symmetry.

No single shape is known which has the property that it can tile the plane only non-repetitively, although one strongly suspects that one must exist. In 3D, John Conway has found a single biprism that can fill space only in a sequence of layers with an irrational rotation angle between each layer.

In addition, in no case has a simple set of tiles been found which force a pattern more complicated than a nested one. The results on page 221 in this book can be used to constructed a complicated set of tiles with this property, but I suspect that a much simpler set could be found.

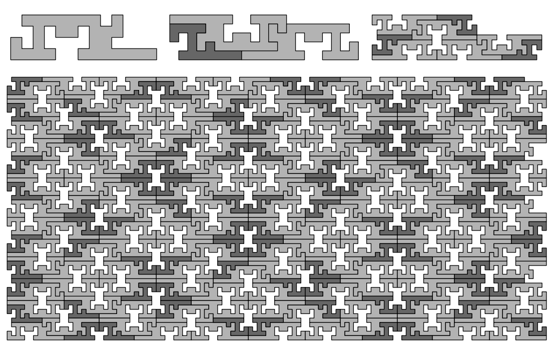
(See also page 1139.)

■ Polyominoes. An example of a tiling problem that is in some respects particularly close to the grid-based constraint systems discussed in the main text concerns covering the plane with polyominoes that are formed by gluing collections of squares together. Tiling by polyominoes has been investigated since at least the late 1950s, particularly by Solomon Golomb, but it is only very recently that sets of polyominoes which force non-periodic patterns have been found. The set (a) below was announced by Roger Penrose in 1994; the slightly smaller set (b) was found by Matthew Cook as part of the development of this book.



(a) *(b)*

Both of these sets yield nested patterns. Steps in the construction of the pattern for set (b) are shown below. At stage n the number of polyominoes of each type is *Fibonacci[2 n - {2, 0, 1}]/{1, 2, 1}*. Set (a) works in a roughly similar way, but with a considerably more complicated recursion.



■ **Ground states of spin systems.** The constraints discussed in the main text are similar to those that arise in the physics of 2D spin systems. An example of such a system is the so-called Ising model discussed on page 981. The idea in all such systems is to have an array of spins, each of which can be either up or down. The energy associated with each spin is then given by some function which depends on the configuration of neighboring spins. The ground state of the system corresponds to an arrangement of spins with the smallest total energy. In the ordinary Ising model, this ground state is simply all spins up or all spins down. But in generalizations of the Ising model with more complicated energy functions, the conditions to get a state of the lowest possible energy can correspond exactly to the constraints discussed in the main text. And from the results shown one sees that in some cases random-looking ground states should occur. Note that a rather different way to get a somewhat similar ground state is to consider a spin glass, in which the standard Ising model energy function is used, but multiplied by -1 or +1 at random for each spin.

■ **Correspondence systems.** For a discussion of a class of 1D systems based on constraints see page 757.

■ **Sequence equations.** Another way to set up 1D systems based on constraints is by having equations like *Flatten[{x, 1, x, 0, y}] === Flatten[{0, y, 0, y, x}]*, where each variable stands for a list. Fairly simple such equations can force fairly complicated results, although as discussed on page 1141 there are known to be limits to this complexity.

■ **Pattern-avoiding sequences.** As another form of constraint one can require, say, that no pair of identical blocks ever appear together in a sequence, so that the sequence does not match *{___, x__, x__, ___}*. With just two possible elements, no sequence above length 3 can satisfy this constraint. But with $k = 3$ possible elements, there are infinite nested sequences that can, such as the one produced by the substitution system *{0 → {0, 1, 2}, 1 → {0, 2}, 2 → {1}}*, starting with *{0}*. One can find the sequences of length n that work by using

```
Nest[DeleteCases[Flatten[Map[Table[Append[#, i - 1],
        {i, k}] &, #], 1], {___, x__, x__, ___}] &, {{}}, n]
```

and the number of these grows roughly like $3^{n/4}$.

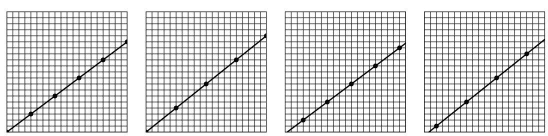
The constraint that no triple of identical blocks appear together turns out to be satisfied by the Thue-Morse nested sequence from page 83—as already noted by Axel Thue in 1906. (The number of sequences that work seems to grow roughly like $2^{n/2}$.)

For any given k, many combinations of blocks will inevitably occur in sufficiently long sequences (compare page 1068). (For example, with $k = 2$, *{___, x__, y__, x__, y__, ___}* always matches any sequence with length more than 18.) But some patterns of blocks can be avoided. And for example it is known that for $k \geq 2$ any pattern with length 6 or more (excluding the ___'s) and only two different variables (say *x__* and *y__*) can always be avoided. But it also known that among the infinite sequences which do this, there are always nested ones (sometimes one has to iterate one substitution rule, then at the end apply once a different substitution rule). With more variables, however, it seems possible that there will be patterns that can be avoided only by sequences with a more complicated structure. And a potential sign of this would be patterns for which the number of sequences that avoid them varies in a complicated way with length.

■ **Formal languages.** Formal languages of the kind discussed on page 938 can be used to define constraints on 1D sequences. The constraints shown on page 210 correspond to special cases of regular languages (see page 940). For both regular and context-free languages the so-called pumping lemmas imply that if any finite sequences satisfy the constraints, then so must an essentially repetitive infinite sequence.

■ **Diophantine equations.** Any algebraic equation—such as $x^3 + x + 1 == 0$—can readily be solved if one allows the variables to have any numerical value. But if one insists that the variables are whole numbers, then the problem is more analogous to the discrete constraints in the main text, and becomes much more difficult. And in fact, even though such so-called Diophantine equations have been studied since well before the time of Diophantus around perhaps 250 AD, only limited results about them are known.

Linear Diophantine equations such as $a x == b y + c$ yield simple repetitive results, as in the pictures below, and can be handled essentially just by knowing *ExtendedGCD[a, b]*.

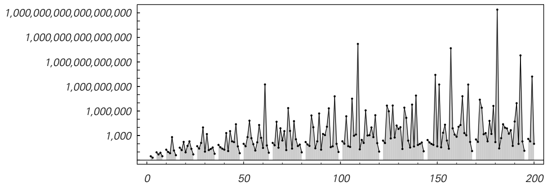


$3x = 4y$ $4x = 5y$ $3x = 4y + 1$ $4x = 5y + 3$

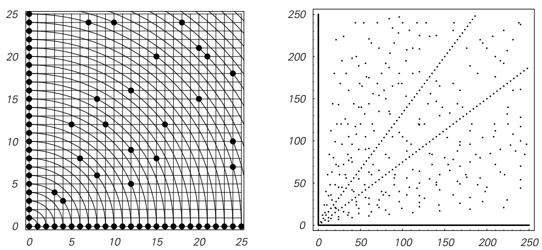
Even the simplest quadratic Diophantine equations can already show much more complex behavior. The equation $x^2 == a y^2$ has no solution except when a is a perfect square. But the Pell equation $x^2 == a y^2 + 1$ (already studied in antiquity) has infinitely many solutions whenever a is positive and not a perfect square. The smallest solution for x is given by

```
Numerator[FromContinuedFraction[
    ContinuedFraction[√a, (If[EvenQ[#], #, 2 #] &)[
        Length[Last[ContinuedFraction[√a]]]]]]]
```

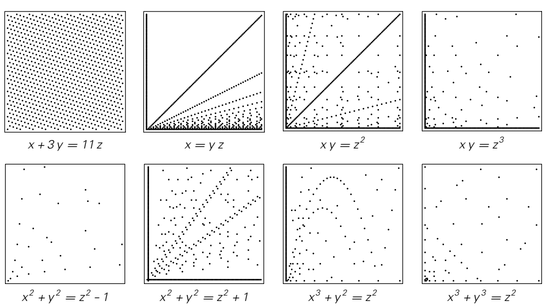
This is plotted below; complicated variation and some very large values are seen (with $a = 61$ for example $x == 1766319049$).



In three variables, the equation $x^2 + y^2 == z^2$ yields so-called Pythagorean triples $\{3, 4, 5\}$, $\{5, 12, 13\}$, etc. And even in this case the set of possible solutions for x and y in the pictures below looks fairly complicated—though after removing common factors, they are in fact just given by $\{x == r^2 - s^2,\ y == 2\, r\, s,\ z == r^2 + s^2\}$. (See page 1078.)



The pictures below show the possible solutions for x and y in various Diophantine equations. As in other systems based on numbers, nested patterns are not common—though page 1160 shows how they can in principle be achieved with an equation whose solutions satisfy $Mod[Binomial[x, y], 2] == 1$. (The equation $(2\, x + 1)\, y == z$ also for example has solutions only when z is not of the form 2^j.)

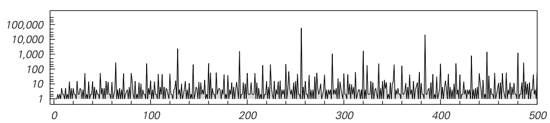


Many Diophantine equations have at most very sparse solutions. And indeed for example Fermat's Last Theorem states that $x^n + y^n == z^n$ can never be satisfied for $n > 2$. With four variables one has for example $3^3 + 4^3 + 5^3 == 6^3$, $1^3 + 6^3 + 8^3 == 9^3$—but with fourth powers the smallest result is $95800^4 + 217519^4 + 414560^4 == 422481^4$.

(See pages 791 and 1164.)

■ **Matrices satisfying constraints.** One can consider for example magic squares, Latin squares (quasigroup multiplication tables), and matrices having the Hadamard property discussed on page 1073. One can also consider matrices whose powers contain certain patterns. (See also page 805.)

■ **Finite groups and semigroups.** Any finite group or semigroup can be thought of as defined by having a multiplication table which satisfies the constraints given on page 887. The total number of semigroups increases faster than exponentially with size in a seemingly quite uniform way. But the number of groups varies in a complicated way with size, as in the picture below. (The peaks are known to grow roughly like $n\hat{}(2/27\, Log[2, n]^2)$—intermediate between polynomial and exponential.) As mentioned on page 938, through major mathematical effort, a complete classification of all finite so-called simple groups that in effect have no factors is known. Most such groups come in families that are easy to characterize; a handful of so-called sporadic ones are much more difficult to find. But this classification does not immediately provide a practical way to enumerate all possible groups. (See also pages 938 and 1032.)



■ **Constraints on formulas.** Many standard problems of algebraic computation can be viewed as consisting in finding formulas that satisfy certain constraints. An example is exact solution of algebraic equations. For quadratic equations the standard formula gives solutions for arbitrary coefficients in terms of square roots. Similar formulas in terms of n^{th} roots have been known since the 1500s for equations with degrees n up to 4, although their *LeafCount* starting at $n = 1$ increases like 6, 25, 183, 718. For higher degrees it is known that such general formulas must involve other functions. For degrees 5 and 6 it was shown in the late 1800s that *EllipticTheta* or *Hypergeometric2F1* are sufficient, although for degrees 5 and 6 respectively the necessary formulas have a *LeafCount* in the billions. (Sharing common subexpressions yields a *LeafCount* in the thousands.) (See also page 1129.)