アルゴリズム入門　＃7

地引　昌弘

2018.11.15

はじめに

今回は、配列を利用して、自然現象や社会現象をコンピュータ上で簡潔に模擬する (これを、**シミュレーション** (simulation) と呼びます) 手法の一つである**セル・オートマトン** (cellular automaton — 略称: CA) を取り上げます。

1　前回の演習問題の解説

1.1　演習 6-1 — 配列の生成

これは簡単にコードだけを示します:

```
import i2a

def arraynew1():              # (a)
  a = i2a.array.make1d(10)
  for i in range(len(a)):     # 配列の長さを返す len 関数を利用しましょう (以下同じ)
    a[i] = 10 - i
  print(a)

def arraynew2():              # (b)
  a = i2a.array.make1d(10)
  for i in range(len(a)):
    a[i] = i % 2
  print(a)

def arraynew3():              # (c)
  a = i2a.array.make1d(10)
  for i in range(len(a)):
    if (i - 4 >= 0):
      a[i] = i - 4
    else:
      a[i] = - (i - 4)
  print(a)
```

```
def arraynew4():              # (d)
  a = i2a.array.make1d(10)
  for i in range(len(a)):
    if (i >= 5):
      a[i] = 0
    else:
      a[i] = 1
  print(a)
```

(a) と (b) は簡単なクイズという感じですね。(c) については、絶対値を求める abs 関数を使う方法もあります (`a[i] = abs(i-4)`)。

1.2 演習 6-2 — 配列の取り扱い

演習 6-2 も擬似コードを省略して、Python のコードだけ記します。まず最大:

```
def arraymax(a):
  max = a[0]
  for i in range(len(a)):
    if a[i] > max:
      max = a[i]
  return(max)
```

このような形で配列を使う場合は、通常「取りあえず変数 `max` に最初の値を入れておき、より大きい値が出てきたら入れ換える」手順を採用します。配列の先頭の添字番号は、0 であることに注意して下さい。

次は最大値が何番目に出て来るかなので、「何番目か」も変数に記録しておき、最大値を更新した際は両者を同時に更新します:

```
def arraymaxno1(a):
  max = a[0]
  pos = 0
  for i in range(len(a)):
    if a[i] > max:
      max = a[i]
      pos = i
  return(pos)
```

最大値を一つだけ記録するのは、単に変数へ入れるだけで可能ですが、最大値が複数あった場合に、その位置を全部打ち出すには、1) まず最大値を求め、2) その最大値と等しいものがあれば位置を打ち出す、という形で 2 回ループを回す必要があります:

```
def arraymaxno2(a):
  max = a[0]
  for i in range(len(a)):    # search for the maximum value
    if a[i] > max:
      max = a[i]
  for i in range(len(a)):
    if a[i] == max:
      print(i)
```

平均値より小さい値を出力するのもこれと同様です:

```
def arrayavgsmaller(a):
  sum = 0.0
  for i in range(len(a)):    # calculating the average value
    sum += a[i]
  avg = sum / len(a)
  print(avg)                 # output the average value
  for i in range(len(a)):
    if a[i] < avg:
      print(a[i])
```

Python では、割り算は常に実数 (浮動小数点) として計算されますが、言語によっては (例えば、Ruby など)、sum の初期値を 0 にすると (つまり、整数)、切捨て除算 (商の小数部分を切り捨て) として扱われてしまうので、注意して下さい。

1.3 演習 6-3 — 配列を使った素数列挙

まず最初は、これまでに見つかった素数を配列に覚えておく方法です。素数が見つかるにつれ、配列を伸ばして行く必要があるので、insert 関数を利用します:

```
def isprime6(a, n):
  limit = int(math.sqrt(n))
  for i in range(len(a)):

    # 既に発見した素数で割り切れた場合は、素数ではないと判定。
    if n % a[i] == 0:
      return(False)

    # 同、割り切れなかった場合は、
    # 割り算をした素数が候補の数 n の平方根以上であれば、
    # それ以上調べても割り切れないことが分かっているので、素数と判定。
    # 平方根未満であれば、次に発見した素数を試すため、ループを継続する。
    if a[i] >= limit:
      a.insert(len(a), n)
      return(True)

  # 候補の数 n が 2 の場合（つまり、初めて isprime6 が呼び出された場合）は、
  # 素数を記録している配列 a が空なので、この処理に至る。
  # n が 3 以上の場合は、a が空ではないので、この処理に来ないはず。
  a.insert(len(a), n)
  return(True)
```

```
def primes6(n):
  start = time.process_time()
  cnt = 0
  a = []    # 事前に配列の大きさを決められないので、作成時の要素数は取りあえず 0 にしておく。
  for i in range(2, n):
    if isprime6(a, i):
      cnt += 1
  print(cnt)
  finish = time.process_time()
  print("%g" % (finish - start))
```

素数判定メソッド isprime6 の動作が少々分かりにくいので、コードの意味を説明するコメントを入れてあります (本格的なプログラムを書く際は、このようなコメントを入れます)。isprime6 は、素数の入った配列 a を受け取り、そこから順に素数 i を取り出して、候補の数 n が割り切れるかどうかを調べていきます。その処理を続け、これまでに取り出した素数で割り切れないまま、新たに取り出した素数が候補の数の平方根 limit より大きくなると、以後に取り出す素数では割り切れないことが明らかなので、即座に「素数である」という答えを返します (候補の数を素数として記録します)。これも、候補の数が 2 の場合だけは、素数の入った配列がまだ作られていない (と言うか、要素が入っていない) ので、他と区別する必要があります。この方法だと、「2 や 3 の倍数を除外」といった細かい工夫をせずとも、0.22 秒で 2 ～ 150,000 までの間に存在する素数を調べられました (Python では、配列の長さを一つずつ伸ばすという処理が、どうも遅いようです)。

最後に、「エラトステネスのふるい」はこれまでと大幅に違う方法です:

```
def primes7(n):
  # 巨大な配列を生成するのは時間が掛かり、
  # ここではアルゴリズムの実行時間を比べることが目的なので、
  # 今回は事前に配列を用意しておくこととした。
  a = i2a.array.make1d(n+1)          # 要素数+1 に注意
  start = time.process_time()
  cnt = 0
  for i in range(2, n+1):            # 終値 = n (n まで調べる → 添え字番号の最後 = n)
    if a[i] == 0:
      cnt += 1
      for j in range(i, n+1, i):     # i の全倍数に対して印を付ける (ステップ i に注意)
        a[j] = 1
  print(cnt)
  finish = time.process_time()
  print("%g" % (finish - start))
```

まず最初に、添字番号が 0～N の配列 a を作り (要素数は $N+1$)、各要素の値を 0 にしておきます。今回は、“添字番号 = 調べる数字” としたかったので、配列の要素数を一つ増やしました。次に、2 から始めて各候補の数値 i に対し、a[i] が 0 ならば素数なので記録するとともに、その素数の全倍数 k について a[k] を 1 にします。これにより、割り算をせずとも i より大きい非素数が、次々とふるい落とされて行きます。この方法は非常に高速で、150,000 以下の素数を調べるのに要した時間は、なんと 0.03 秒です[1]。最初の素朴版が数百秒のレベル、こちらが数十ミリ秒のレベルなので、両者を比較すると **10,000 倍!!!** も速くなったわけです (今回の例で言えば、20,000 倍)。日常的な世界では、「10,000 倍の差」というものはなかなかありません。我々の歩く速度はおよそ 4 km/h ですが、40,000 km/h というのは地球の重力圏を脱出するのに必要な速度です。この例に比べて、プログラムの動作速度、つまりアルゴリズムの世界は、その良し悪しにより簡単に「ものすごい差」が付いてしまいます (まあ、これを学ぼうというのが、本講義の趣旨なのですが)。

[1]因みに、配列の生成込みの時間は 0.13 秒でした。それでも、前回作成した最速のアルゴリズム (6 の倍数 ± 1…) と同じですね。

2　2 次元配列

2.1　2 次元配列の生成

これまでに出て来た配列は全て 1 次元配列 (各データが 1 列に並んでいるだけ) でしたが、2 次元配列 (配列の配列) を利用したい場合もあります。Python では、1 次元配列と同様に 2 次元配列も扱うことができます。「2 次元配列」は、「縦横 2 次元に要素が並んでいる」というイメージですが、実際には図 1 のように、配列の各要素が配列という構造になっています (とは言え、縦横 2 次元のイメージでも実用上は差し支えありません)。

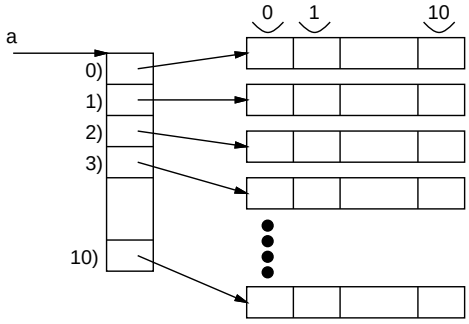

図 1: 2 次元配列

2 次元配列 a の各要素は、行の並び方向 (上下方向) の添字番号 i と、列の並び方向 (左右方向) の添字番号 j を用いて、a[i][j] と指定します。2 次元配列の生成は、1 次元配列と同じく、全ての値を直接「[[a,b], [c,d]]」のように指定することで生成できます。しかし、この方法では、配列が大きくなるととても面倒です。1 次元配列の作成では、まず i2a.array.make1d 関数を用いて配列を用意し、その後で for ループを用いて値を設定しました。2 次元配列についても同様な方法を利用しましょう[2]。但し、2 次元配列に対して配列の長さを返す len 関数を利用する場合は、注意が必要です。len 関数は配列に格納された要素数を返すのですが、2 次元配列は [[a,b], [c,d]] のように格納されるので (図 1)、2 次元配列に len 関数を適用した場合は、結果として行方向の長さが返されることになります (つまり、a[i][j] の i に該当する長さ)。多くの場合は、行方向/列方向に沿って処理をする方が見通しが良いので、下記のように変数 size_i, size_j を利用する方が無難です:

```
a = i2a.array.make2d(size_i, size_j)     # size_i 行 size_j 列の配列を作成 (各要素の値 = 0)
for i in range(size_i):
  for j in range(size_j):
    a[i][j] = …                          # 2 次元配列の各要素に値を設定
```

最後に:

配列は、様々なプログラムで利用される基本的なデータ構造です。従って、各プログラミング言語では、配列を効率的に処理するために、様々な関数や構文が用意されています。本講義の狙いから外れるため、ここではそれらを逐一取り上げることはしませんが、標準講義資料の 5.5 節「【発展】配列の様々な機能」および 5.6 節「【発展】配列とコピー」は、必ず目を通しておいて下さい。

演習 7-1 i2a.array.make2d 関数を用いて 5×5 の 2 次元配列を生成し、for ループを用いて (a)〜(d) の値を設定しなさい。

(a)

0	1	2	3	4
-1	0	1	2	3
-2	-1	0	1	2
-3	-2	-1	0	1
-4	-3	-2	-1	0

(b)

1	0	0	0	0
1	1	1	1	1
1	2	4	8	16
1	3	9	27	81
1	4	16	64	256

(c)

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

(d)

1	0	0	0	1
0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1

[2] i2a.array.make2d 関数の使用に際しては、i2a.array.make1d 関数と同様、事前に import i2a を用いて、アルゴリズム入門の演習用ライブラリを読み込んでおく必要があります。

なお、Python のプログラムで 2 次元配列を出力する際は、print 関数ではなく pprint.pprint 関数を使うと見易く出来ます。但し、これを使うには、事前に import pprint を用いて、pprint 用ライブラリを読み込んでおく必要があります。pprint.pprint 関数は、width オプションで出力を改行するまでの文字数を指定出来ます。使い方は、こんな感じ:

```
>>> import pprint
>>> a = [[0,0,0,0,0], [0,1,2,3,4], [0,2,4,6,8], [0,3,6,9,12], [0,4,8,12,16]]
>>> print(a)                        ← そのままでは見易くない
[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8], [0, 3, 6, 9, 12], [0, 4, 8, 12, 16]]
>>> pprint.pprint(a, width=20)  ← pprint.pprint を使い、横幅を適宜設定すると、
[[0, 0, 0, 0, 0],               ← 揃えてくれる (今回は 1 行 20 文字)
 [0, 1, 2, 3, 4],
 [0, 2, 4, 6, 8],
 [0, 3, 6, 9, 12],
 [0, 4, 8, 12, 16]]
```

3 セル・オートマトン

セル・オートマトンは、配列を利用して、自然現象や社会現象をコンピュータ上で簡潔にシミュレートする手法の一つです (標準講義資料にあるライフ ゲームもセル・オートマトンの一つです)。セルは “枡目” を意味し、オートマトンは “自動機械” を意味します。セル・オートマトンは、次のように振る舞います。

- 各セルは状態 (取りあえずここでは、値のようなものと考えておいて下さい) を持つ[3]。
- 各セルは、時刻 t における自セルおよび近傍 (要は、その周囲) セルの状態により、時刻 $t+1$[4]の状態が決まる。

これだけ見るとシンプルですね (まあ逆に、シンプルだからこそ、様々な自然現象や社会現象に適用出来るというわけですが)。セル・オートマトンのミソは、上記 2 項目の状態を変化させる規則です。以下では、説明を簡素化するため、状態は 0 あるいは 1 のどちらか 2 通りしかないとします。

3.1 1 次元セル・オートマトン

セル・オートマトンの中で、一番簡単な例は、セルが横一列に並んだ 1 次元セル・オートマトンです。図 2 のように、1 次元セル・オートマトンを各時刻毎に並べることで様々な状態パターンが作成されます。

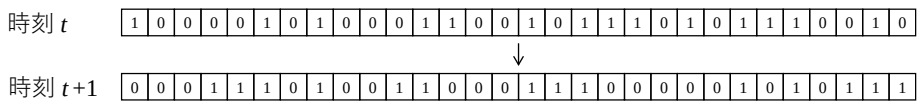

図 2: 1 次元セル・オートマトンの例

このセル・オートマトンは 1 次元なので、時刻 $t \rightarrow$ 時刻 $t+1$ での状態変化に関与するセルは、自セルとその両脇セルの計 3 セルです。これら 3 セルの状態が取り得る場合の数は、図 3 (次ページ) で示すように 8 通りです。この図では、変化後の (時刻 $t+1$ の) 状態は “?” になっていますが、実際にはここが 0 あるいは 1 になります。8 個の 0 あるいは 1 からなる場合の数は、$2^8 = 256$ 通りなので、状態変化規則は 256 種類存在します。

[3]正確に言えば、“状態” とは、例えば駅の発券窓口にいる係員を例にすると、1) お客さんが来るまでは待っている状態、お客さんが来た後は、2) 要望を聞いている状態、3) 要望に合った切符を検索している状態、切符が取れた後は、4) 料金を受け取る状態などがありますが、これら各状態に値を割り当て、その値で表すことも (例えば、“3” の状態とか “5” の状態とか) 可能なので、ここではあまり難しく考えず、値のようなものと考えておきましょう、ということです。

[4]これは、実世界での時間の流れのような連続的な時間経過ではなく、1 回目, 2 回目といった非連続な時間経過を意味しています。

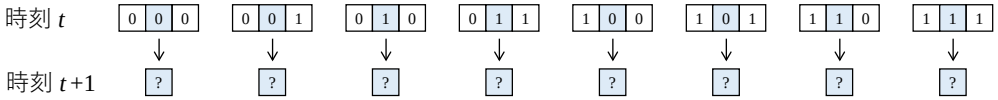

図 3: 1 次元セル・オートマトンの状態変化規則

これまでに、全 256 種類の状態変化規則から生成される状態パターンが調べられています。その結果、各状態パターンは、以下の四つに場合分け出来ることが分かりました。

クラス 1: どのような初期状態から始めても、やがて全てのセルが均質な状態になるパターン

クラス 2: セルが周期的な状態変化を繰り返すパターン

クラス 3: カオス的な (乱雑な) パターン (乱雑の中に一部、自己相似的な局所構造が現れることもある)

クラス 4: クラス 1/2 の秩序状態と、クラス 3 のカオス状態が共存するようなパターン

以下に、各クラスの状態パターン例を示しておきます。これらの図は、各時間における 1 次元セル・オートマトンの状態を、時間の経過に伴い上から下へ並べたものです。黄緑色のセルは状態 1 を、白色 (と言うか、もう灰色だね) のセルは状態 0 を表しています。必要に応じて、拡大して見てみて下さい。

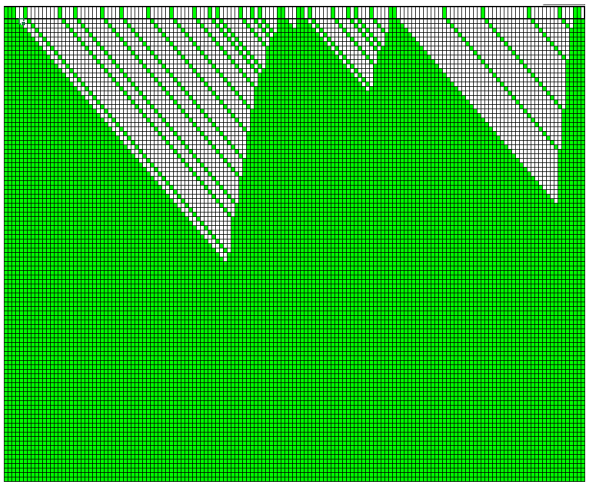

図 4: クラス 1 の状態パターン例

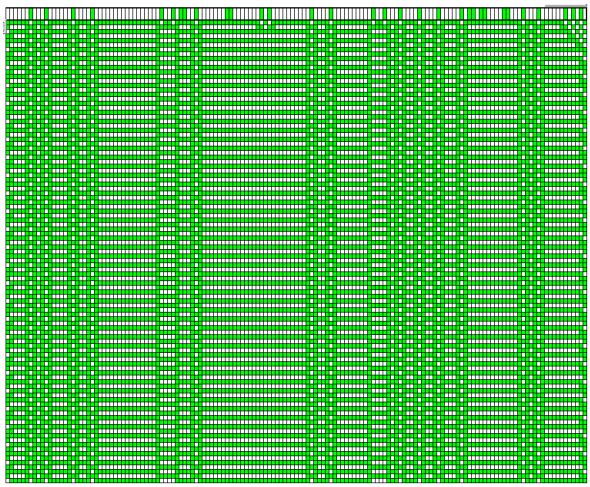

図 5: クラス 2 の状態パターン例

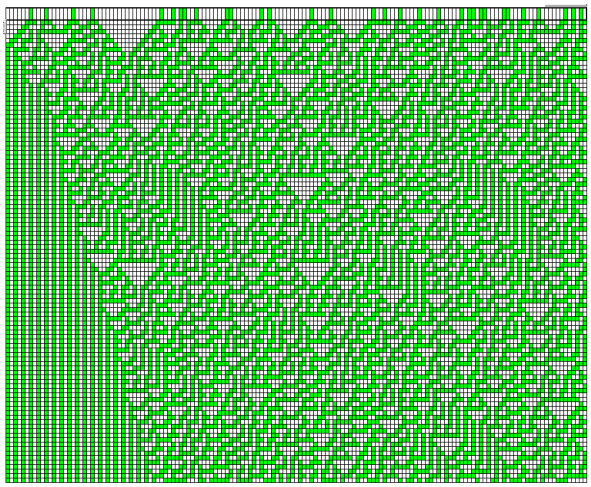

図 6: クラス 3 の状態パターン例

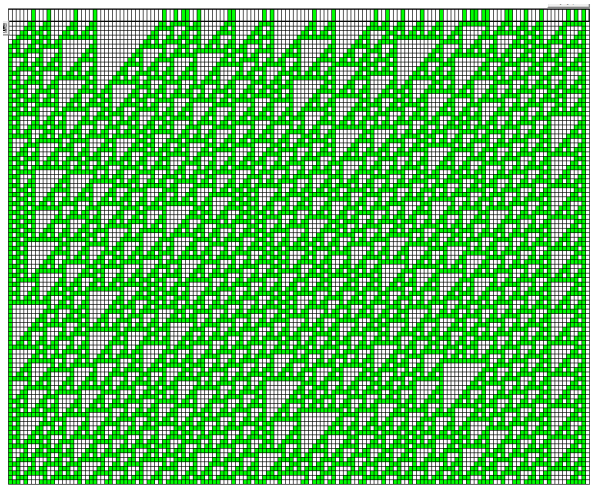

図 7: クラス 4 の状態パターン例

これらを見る限り、はぁそうですか、という感じしかしないのですが、実はこれまでに、カオスに分類されない新たなクラス 4 の状態から、大変興味深い事実が導かれています。図 7 を見てみると、様々な大きさの三角形が並んでいます。向きは揃っているのですが、大きさはバラバラだし、その出現パターンについても一見規

則性があるようには見えません。これのどこが興味深いかと言うと、実は、このパターンはプログラム言語になっているのではないか、と考えられたからです。何やら唐突な話で、ちょっと戸惑いますね。でも知りたいという人もいると思うので (私としては、いて欲しいのですが…)、簡単に紹介しておきます。

1 次元セル・オートマトンは、0/1 のパターンを渡され、それを元にまた 0/1 のパターンを生成するものです。実は、クラス 4 の規則では、渡されたビット パターンの一部 (以後、これをビット パターンと呼ぶことにします) に対して、下記の特徴を備えたビット パターンを (どこかの時刻で) 生成することができるのです。

- 渡されたビット パターンと同じものが、同じ列に生成される。
- 渡されたビット パターンと同じものが、(規則性を持って) 別の列に生成される。
- 渡された二つのビット パターンが、入れ替わる。
- 渡された二つのビット パターンから、別のパターンが生成される (生成されたパターンは規則性があり、また以後の時間経過において変化しない)。

これがプログラム言語とどのような関係があるかを考えるために、まずは、プログラムを表現するのに必要な機能とは何か、という観点から少し掘り下げてみましょう (要は、何があれば制御構造や各種演算を表せるか、ということです)。天下り的ですが、下記の動作をする subleq(a, b, X) 命令を考えます。

1. b にある値から、a にある値を引き、その結果を b にしまう。
2. ステップ 1 の引き算の結果が 0 以下であった場合、次は場所 X にある subleq 命令を実行する。
3. 同、結果が 0 より大きい場合 or X の指定がない場合は、単に次行の subleq 命令を実行する。

実は、この subleq(a, b, X) 命令が一つあれば、プログラムに必要な機能をほぼ全て用意することが出来るのです。例えば、次のような if 文は、

```
(条件分岐前の処理)
if v == 0:
  (条件成立時の処理)
else:
  (条件非成立時の処理)
(条件分岐後の処理)
```

以下のように作れます (しかしながら、相当複雑ですね…)。

```
(条件分岐前の処理)
subleq(a,a)          # a は作業領域，a-a → a より a=0
subleq(b,b)          # b も作業領域，同様に b=0
subleq(v,a,Label-1)  # a(=0)-v → a，a=(-v) ≦ 0 (つまり v ≧ 0) の時は Label-1 へ移動
subleq(a,a,Label-F)  # ここに来るのは -v>0 (つまり v ≠ 0) の時，条件非成立へ移動
Label-1:
subleq(a,b,Label-T)  # a=(-v)，b(=0)-a → b，b=(-a)=v ≦ 0 の時は Label-2 へ移動
subleq(a,a,Label-F)  # ここに来るのは v>0 (つまり v ≠ 0) の時，条件非成立へ移動
Label-2:
subleq(a,a,Label-T)  # ここに来るのは v ≧ 0 かつ v ≦ 0 (つまり v=0) の時，条件成立へ移動
Label-T:
(条件成立時の処理)
subleq(a,a,Label-E)  # 条件成立時の処理を終えたので、条件分岐後へ移動
Label-F:
(条件非成立時の処理)
subleq(a,a,Label-E)  # 条件非成立時の処理を終えたので、条件分岐後へ移動
Label-E:
(条件分岐後の処理)
```

上の処理で、subleq(a, a, Label-xx) となっている部分は、必ず Label-xx へ移動したいため、単に同じ値を引き算しているだけです (結果が 0 以上なので、Label-xx へ移動するという意)。また、Label-2 の存在は冗長で、実は Label-T へ直接移動できるのですが、分かり易さのために入れました。if 文以外にも、同様に while/for などの制御構造や各種演算を作れます (つまり、上にある “条件○○の処理” となっている部分も全て、subleq 命令だけで作ることができるわけです)。以上より、一見多機能なプログラミング言語も、極めてシンプルな命令に分解することができるというわけです (同じく、数学における様々な計算や関数も、極めてシンプルな初期関数とその帰納法により表現できます)。

このような考え方のもと、7 ページで挙げたクラス 4 による出現ビット パターンの特徴をうまく組み合わせてみたら、実はプログラミング (つまり、手順を定義して実行すること) の原始となるシステムを作り出すことができた[5]、というわけなのです (但し、subleq 命令の例で見たように、理論上は利用可能であるという意味で、例えば本講義で取り上げたプログラムなどを自在に作成することは現実的ではありません)。これが突破口となり、以後、様々な思わぬものがプログラミング言語として利用できることが分かって来ました (これも、理論上は利用可能であるという意味で、実際にプログラムを作成するという点では、やはり現実的ではありません) 。例えば、Wii U 用のゲーム ソフトであるスーパー マリオ メーカーでは、マリオの通過を邪魔する障害物の出現パターンを設定出来ます。そこで、0 を障害物 A、1 を障害物 B で表すことにして、これらの出現パターンをクラス 4 と同等のパターンにする設定を作れるようです (他にも、Minecraft やポケット モンスター、マジック・ザ・ギャザリングなども話題になっています/まあ、これらの信憑性は良くわかりませんが…)。

[5] ここまで来てこんな説明では納得しない or さらに興味を深めた人のために、もう少しだけ説明しておきます。例えば、$m+n$ を計算する “簡潔な” システムを考えます。まずは配列 `a` を用意し、数値 m は `m` 個の要素に連続して 1 が入ったものとしましょう。(今回は、`a[0]`∼`a[m-1]=1, a[m]=0, a[m+1]`∼`a[m+n]=1` です)。次に、現在参照している要素の添字番号をインデックス `i` とし、`a[i]` が演算子の左項 (`L`) or 右項 (`R`) のどちらに該当するかを状態 `s` で表すことにします。そして、`a[i]` が 1 であれば、1) `a[i]=0` に変える、2) 適当な場所 `a[X+i]` に 1 をコピーする、3) `i` を一つ増やすことにします (これを規則と呼びましょう)。また、`a[i]` が 0 であれば終了するのですが、但し、状態 `s` が `L` の場合は、1) 状態 `s` を `R` に変える、2) `i` を一つ増やすことにします。ここで、`i=0, s=L` に対して上の規則を逐次適用すると、`a[X]` 以降の要素に `m+n` 個の 1 が並び (それ以外は 0)、結果として $m+n$ の計算をしたことになります。このシステムには、配列/インデックス/状態/規則しかありません (規則と言っても、i) 状態の変更, ii) 要素に 0 or 1 を代入, iii) インデックスの変更、の三つしか行ないません)。簡潔なシステムですが、実は、計算可能なアルゴリズムは全てこのシステム上でプログラミングできる、ことが証明されています (このようなシステムを、**チューリング機械** (Turing Machine) と呼びます)。そして、7 ページに挙げたクラス 4 の特徴 (四つ) を用いることで、このチューリング機械を模倣できたというわけなのです (言い換えれば、プログラミング言語に必要な機能とは、チューリング機械の動作を真似できればよいのです)。但し、プログラムと言っても、これまで見て来たものとは大きく異なります。1 次元セル・オートマトンでは、時刻 t における `a[i]` の値は、時刻 $t+1$ では `a[i ± 1]` に影響を与えます。つまり、ビット パターン (の影響) は、斜め方向に伝わって行くわけです。これより、プログラムに必要なビット パターンは、配列 `a` の遥か前方 (`a[`$\to -\infty$`]` のどこか) or 後方 (`a[`$\to +\infty$`]` のどこか) に入れておくという形態を取ります (念のため、この脚注内の配列は、1 次元セル・オートマトンの世界におけるセルを意味しており、Python の配列とは異なります/ Python の配列では、負の添字番号はまた違った意味を持っています)。

3.2　セル・オートマトンの応用

1 次元セル・オートマトンは、その生成規則から得られる複雑な現象について探ることで、前節で紹介した話題だけではなく、秩序とカオスの混在した**複雑系** (complex system) と呼ばれる新しい分野も切り開きました。しかしながら、自然現象や社会現象をコンピュータ上で簡潔にシミュレートするという観点からは、あまり使い勝手の良いものではありません。そこで、取りあえずここは、計算とは何ぞや現象とは何ぞやという話題を少し脇に置いて (まあ、私はそちらの方が好きですけど)、セル・オートマトンを利用したシミュレーションについて考えてみることにしましょう。

セル・オートマトンは、自身および近傍の状態により、次の状態が決まるというものです。実世界も、周囲からの影響により様相を変えて行くので、セル・オートマトンは実世界を簡潔に表現できるはずです。但し、実世界では 1 次元という関係は少なく、大半は複数次元の関係を持っています[6]。そこで、まずは問題を考えるに当たり、セル・オートマトンを何次元に拡張すればよいかを考えましょう。それには、対象となる現象を説明するモデルを考える必要があります。

では、次の問題をコンピュータ シミュレーションによる解析を通じて、検討してみましょう。

> 「現時点で緑地と砂漠が混在する対象地域 A は、砂漠化の進行が懸念されています。
> A において、砂漠が今後どの程度まで進行するのか調査しなさい。」

この問題を調査するに当たり、まずは、砂漠が広がる理由 (or 仕組み) をモデル化する必要があります。これは、天候や人間の社会活動が影響していると考えられますが、これらをモデル化する方針として、因果関係を定量的 (or 解析的) に分析するミクロ的視点と、結果の定性的分析に基づくマクロ的視点の二つがあります。今回はマクロ的視点に立ち、シンプルに、

> 「対象地域 A を小さい矩形 a に分け、a の周囲に砂漠が多い場合は、a の降雨量も減る
> と想定されるので、a も砂漠化する。」

という方針を考えてみましょう。この方針では、天候が及す影響については、周囲の環境が及ぼす影響に含まれると説明できるので、モデルの対象から外すことができます。次に、社会活動が及す影響ですが、周囲が砂漠であれば、それに伴う天候からの影響により、社会活動に必要な水の確保が益々砂漠化を進めてしまう、と考えることができます。よってこれも、モデルの対象から外すことができます。以上より、今回は矩形領域 a の周囲にある土地の状況だけを考えることにします。これより、各矩形をセルとみなす 2 次元のセル・オートマトンを利用できます。

次に、先ほどの方針に基づき、2 次元のセル・オートマトンを用いた砂漠化のモデルを定義します。今回は、砂漠化の進行を示す状態変化規則を、図 8 のように定義しました。

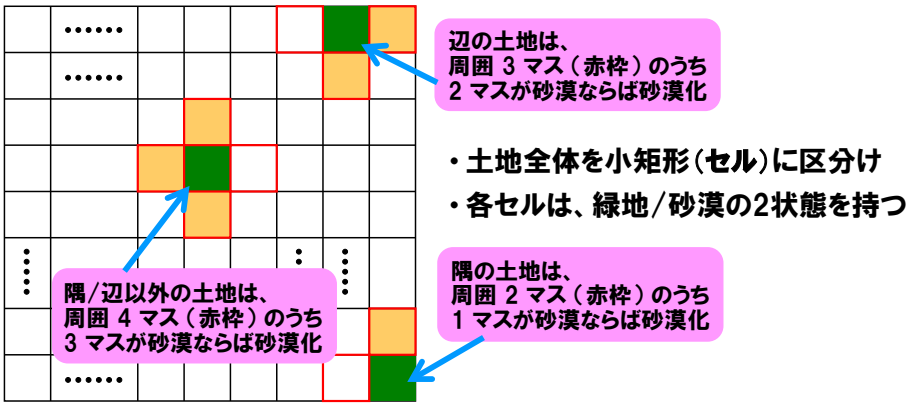

図 8: 砂漠化の進行を示す状態変化規則

[6] ここで言う次元の数は、影響を受ける相手の数に関係すると考えて下さい。この関係を、数学的な 2，3 次元で表せるのか、あるいは 4 次元以上が必要なのかを考えることになります

このモデルを元に、プログラムを砂漠化の進行をシミュレートするプログラムを作ることにします。まずは、疑似コードを考えてみましょう。

- cell_auto() — 砂漠化の進行を調べる
- 　必要な変数を用意
- 　(# 対象地域を区分けした各区画を表す 2 次元配列, 各区画が調査済みであることを示すフラグなど)
- 　区画用データの準備
- 　(# 土地データを読み込み、区画データ用の 2 次元配列に格納するなど)
-
- 　緑地⇒砂漠と変化した区画が存在する間、繰り返し (# 変化した区画の有無はフラグで表す)
- 　　フラグを True に初期化
- 　　左上セルから右下セルまで、繰り返し (# これが調査 1 回分/1 回の調査で変化させるセルは 1 個)
- 　　　もし、調査対象区画が「既に砂漠」であったら、
- 　　　　何もせず次の区画へ移動
- 　　　(枝分かれ終わり)
- 　　　もし、調査対象区画が四隅でだったら、
- 　　　　隣接する 2 区画を調べる (# どちらかが砂漠ならば砂漠化が進行)
- 　　　　調査対象区画が砂漠に変わった場合は、フラグを False に変える
- 　　　(枝分かれ終わり)
- 　　　もし、調査対象区画が四辺でだったら、
- 　　　　隣接する 3 区画を調べる (# 三つのうち、2 区画以上が砂漠ならば砂漠化が進行)
- 　　　　調査対象区画が砂漠に変わった場合は、フラグを False に変える
- 　　　(枝分かれ終わり)
- 　　　もし、調査対象区画が四隅/四辺でなかったら (# 周囲が囲まれているならば)、
- 　　　　隣接する 4 区画を調べる (#四つのうち、3 区画以上が砂漠ならば砂漠化が進行)
- 　　　　調査対象区画が砂漠に変わった場合は、フラグを False に変える
- 　　　(枝分かれ終わり)
- 　　(調査 1 回分の繰り返し終わり)
- 　(全調査の繰り返し終わり)
-
- 　結果を表示

最後は、上の疑似コードを元に Python のプログラムを作成します。緑地/砂漠の状態は、数字の 0/1 で表すことにします。また、各セルの位置関係ですが、2 次元配列を用いれば、図 9 のように扱うことができます[7]。

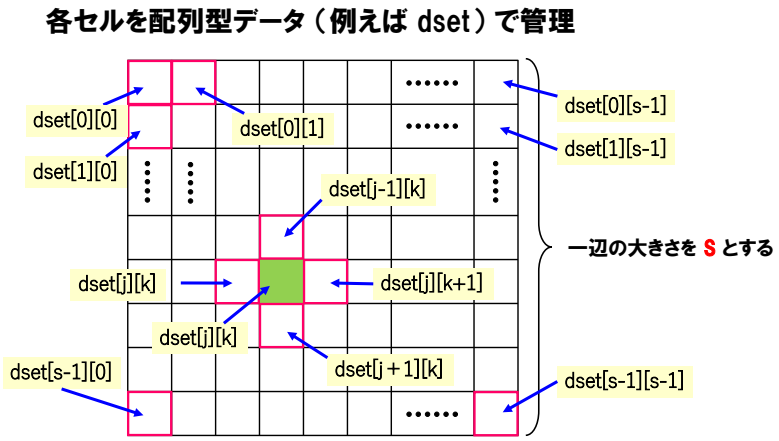

図 9: 2 次元配列を用いた各セル間の位置関係

[7]規模の大きいプログラムを作成する際は、(取りあえずは簡単なものでも構わないので) 必ず疑似コードおよび、図 8 や図 9 といったモデル図を用意しましょう。これが設計図になります。

以上の検討/設計を踏まえ作成した Python プログラムを次ページに示します (プログラム内のコメントをしっかりと読んで下さい)。また、以下では、このプログラムを作成する際に留意点について挙げておきます。

このプログラムで利用する土地データは、“land.data” に入っています。また、このプログラムによるシミュレーションは、例えば cell_auto(“land.data”) と実行します[8]。図 10 に、シミュレーションの実行結果を示します。左側が land.data に入っている初期データです。また、右側が土地の最終的な状態 (シミュレーション結果) です。最終状態にある赤字部分は、緑地 → 砂漠に変わった区画を示しています (実際の出力は赤くありません)。このプログラムを実行すると、まずは初期データを出力し、次に最終的状態を出力します。

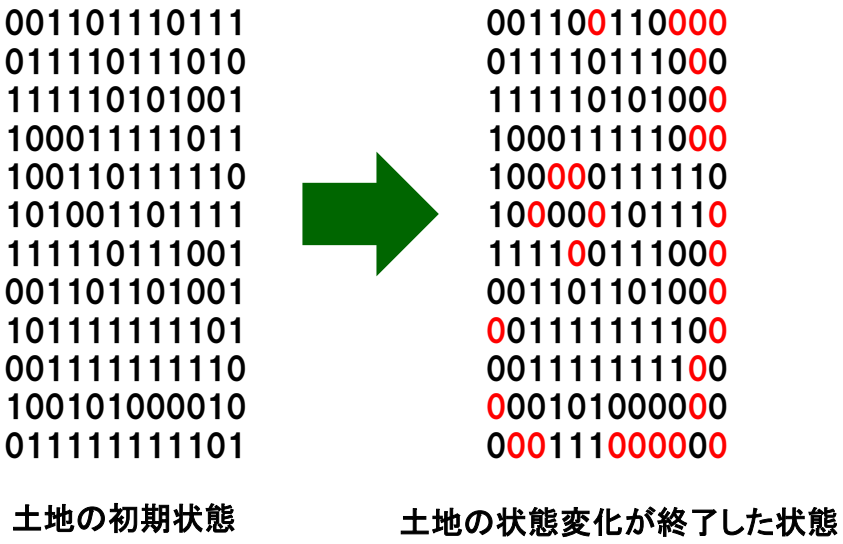

図 10: シミュレーション結果の例

緑地 → 砂漠に変わった区画があるかどうかは、フラグ `finish` に記録されます。`finish` は、全ての調査用関数が利用するため、プログラムの冒頭で大域変数として定義しています。但し、Python では関数内で大域変数を利用する場合、再度 `global` 宣言をする必要があります。これは、大域変数と関数内で定義した変数との違いを明確にするためです。大域変数として定義した変数は、各関数へ引数として渡す手間が省けますが、どれかの関数で誤用されると、(特に並列処理においては) エラー箇所の発見が難しくなるため、注意深く利用する必要があります。このプログラムでは、区画データが入った配列 `dset` も全ての調査用関数が利用します。`dset` については今回、本体である `cell_auto` 関数内で定義しました。これは大域変数ではないので、各調査用関数では利用できません。この場合は、引数として渡すことで利用できるようになります (両者を使い分ける意味は、第 6 回資料の図 14 および p.12 を参照)。このような定義する場所に応じた有効範囲を “スコープ” と呼びます。プログラムを正しく動作させるには、“スコープ” を正しく把握することが重要になります。

さて、次ページ以降にあるプログラムですが、最終的に完成すると 250 行前後の規模になります。プログラムの規模としては、決して大きくはないですが、この程度の規模でも初めて作るとなると戸惑うことも多いので、以下に要領を整理しておきます (当然ですが、標準講義資料の「6.6 節 テストとデバッグの基本」には目を通しておいて下さい)。

- 全てを間違いなく一度に作成することは不可能なので、少しずつ作成して行きます。ある機能を作成した後は、その部分のテストを行ない、これがうまく行けば、次の機能に取り掛かるというわけです。作成した部分までを実行して途中で抜けたい場合は、例えば、一時的にその箇所へ `return` 関数を入れるという方法があります。テストにより作成した機能が正しく動作することが確認できた後は、もちろんこれ (一時的な `return` 関数) の削除 (or コメント アウト) を忘れないで下さい。

[8] cell_auto 関数に渡す引数である土地データの入ったファイルについては、そのパス名に注意して下さい。土地のデータ ファイルと作成した Python プログラムが同じフォルダにある場合は、cell_auto(“land.data”) で実行できますが、違うフォルダにある場合 (例えば、土地のデータ ファイルが data フォルダにある場合) は、cell_auto(“data/land.data”) となります。

- 機能単体のテストを行なう場合は、分かり易いテスト データを用意する必要があります。今回の例で言えば、例えば四隅の調査機能をテストする場合、四隅だけが 1 で残りが 0 である試験用土地データを作成して、テストします (land.data を参考に作ればよいですね)。
- 標準講義資料にある `print` 法を実践するために、今回は `print_data` 関数を用意してあります (適宜、必要な箇所に埋め込んで利用して下さい)。また、特定の処理が正しく実行されているかどうかは、
  - その前後に print 関数でメッセージや該当データを出力させてみる。
  - デバッグ用の if 文を用意して、メッセージや該当データを出力させてみる。

  といった手法が有効です (特に、後者を使いこなせるようになれれば一人前です)。

課題: セル・オートマトンを用いて砂漠化の進行をシミュレートする Python プログラム

```
import i2a      # アルゴリズム入門の演習用ライブラリを読み込む。

SIDE = 12       # 地域全体の大きさ（一辺の大きさ）
finish = False  # 緑地 => 砂漠へと変化した区画が存在するかどうかを表わすフラグ
                # まず最初は、while ループの中に入る必要があるので、
                # 便宜上、初期値は False にしてある。
                # while ループへ入った後は、改めて初期値を設定する。

# 区画用データの表示（確認のため、適宜プログラム内に挿入/コメントアウトする）
def print_data(dset):

    # 各変数の宣言
    j = 0; k = 0

    # 動作確認用に、2 次元配列の要素を一つずつ表示
    # dset[j][k] は、(j 行, k 列) の要素（XY 座標系とは逆なので要注意）
    # print(..., end="") は "改行なし出力", print("") は "改行一つ" を表わす。
    for j in range(SIDE):
        for k in range(SIDE):
            print(dset[j][k], end="")
        print("")

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)


# 区画用データの準備
def init_data(fname, dset):

    # 各変数の宣言
    cnt = 0                 # 読み込んだデータ数（区画データの総数）
    j = 0; k = 0

    # 区画データのファイルを読み込む（open したファイルの各情報は、変数 fd に入る）
    fd = open(fname, "r")
    raw_data = fd.read()    # ファイルより生データを読み込み、配列 raw_data にしまう。
    fd.close()

    # 生データから区画データ（2 次元配列）に変換
    # 添字番号（index）の関係および境界処理に注意のこと
    for i in range(len(raw_data)):

        # 改行文字は読み飛ばす。
        if raw_data[i] == "\n":
            continue
```

```
            # 便宜上、以後の処理では、
            # 区画データを文字データではなく数値データとして利用するため、
            # int 関数で "文字 → 数値" の変換をしておく。
            dset[j][k] = int(raw_data[i])
            cnt = cnt + 1
            k = k + 1
            if k == SIDE:
                k = 0
                j = j + 1

    # 初期データを 2 次元配列全体として表示（確認用/確認後はコメントアウト）
    print_data(dset)

    # 区画データの総数を返して終了
    return(cnt)


# 調査対象区画が四隅のどれかであった場合の処理
def check_four_corners(dset, j, k):

    # 各変数の宣言
    global finish

    # 調査対象区画が左上端だった場合
    + ----------------------------------------------------------------+
    | ここの部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +----------------------------------------------------------------+

    # 同、右上端だった場合
    + ----------------------------------------------------------------+
    | ここの部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +----------------------------------------------------------------+

    # 同、左下端だった場合
    + ----------------------------------------------------------------+
    | ここの部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +----------------------------------------------------------------+

    # 同、右下端だった場合
    + ----------------------------------------------------------------+
    | ここの部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +----------------------------------------------------------------+

    # 調査対象区画が四隅のどれでもなかった場合は、何もしない。

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)
```

```
# 調査対象区画が四辺のどれかであった場合の処理
def check_four_sides(dset, j, k):

    # 各変数の宣言
    global finish
    dp = 0              # 砂漠化した区画の数

    # 調査対象区画が上辺だった場合
    if (j == 0) and ((1 <= k) and (k <= SIDE - 2)):

        # 調査対象区画の左区画/右区画/下区画を調べる。
        # 3 区画を全て必ず調べたいので、elif ではなく if を使う (要注意)。
        # もし砂漠だったら、dp にその区画数をしまっておく。
        if dset[j][k - 1] == 0:
            dp = dp + 1
        if dset[j][k + 1] == 0:
            dp = dp + 1
        if dset[j + 1][k] == 0:
            dp = dp + 1

        # 隣接する 3 区画のうち、2 区画以上が砂漠ならば、その土地も砂漠化
        if dp >= 2:
            finish = False
            dset[j][k] = 0

    # 同、左辺だった場合
    elif ((1 <= j) and (j <= SIDE - 2)) and (k == 0):
      + ------------------------------------------------------------------+
      | ここの部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。|
      +------------------------------------------------------------------+

    # 同、右辺だった場合
    elif ((1 <= j) and (j <= SIDE - 2)) and (k == SIDE - 1):
      + ------------------------------------------------ ------------------+
      | ここの部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。|
      +------------------------------------------------ ------------------+

    # 同、下辺だった場合
    elif (j == SIDE -1 ) and ((1 <= k) and (k <= SIDE - 2)):
      + ------------------------------------------------ -----------------+
      | ここの部分は未完です。調査対象区画が上辺だった場合を参考に作成して下さい。|
      +------------------------------------------------ -----------------+
```

```
    # 調査対象区画が四辺のどれでもなかった場合は、何もしない。

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)


# 調査対象区画が周囲を囲まれている場合の処理
def check_other_cells(dset, j, k):

    # 各変数の宣言
    global finish
    dp = 0              # 砂漠化した区画の数

    # 調査対象区画が四隅/四辺でない場合（周囲が囲まれている場合）
    + ----------------------------------------------------------------+
    | ここの部分は未完です。check_four_sides 関数を参考に作成して下さい。|
    +-----------------------------------------------------------------+

    # 調査対象区画の周囲が囲まれていない場合は、何もしない。

    # 返す値は特にないが、関数定義の終わりを示すために return 関数を入れておく。
    return(None)


# プログラム本体
def cell_auto(fname):

    # 各変数の宣言
    global finish
    cnt = 0             # 区画データの総数（今回は使っていない）

    # 区画データ用の領域を確保（今回は一辺が SIDE となる正方形です）
    # 区画データの意味:
    #   砂漠 = 0（文字の "0" でなく、数値の 0 として扱う点に注意）
    #   緑地 = 1（文字の "1" でなく、数値の 1 として扱う点に注意）
    dset = i2a.array.make2d(SIDE, SIDE)

    # 区画用データの準備
    cnt = init_data(fname, dset)

    # 緑地 => 砂漠と変化した区画が存在しなくなるまで調査を続ける。
    #   finish == True:  緑地 => 砂漠と変化した区画が存在しない。
    #   finish == False: 緑地 => 砂漠と変化した区画が存在する。
    while not finish:

        # 終了フラグを初期化
        finish = True
```

```
    # 調査 1 回分:
    #   左上から右下まで、(j 行, k 列) にある区画を一つずつ調べ終わると、
    #   1 回分の調査が終了する。
    + ----------------------------------------------------------------+
    | ここの部分は未完です。                                          |
    | 2 次元配列を左上から右下まで一つずつ順に巡る for ループを作成して下さい。|
    | for ループのブロックでは、各区画を dset[j][k] として参照します。|
    +----------------------------------------------------------------+

            # 調査対象区画が既に砂漠であったら、何もせず次の区画へ移動
            if dset[j][k] == 0:
                continue

            # 同、四隅の場合
            # 隣接する 2 区画のうち、どちらかが砂漠ならば砂漠化
            check_four_corners(dset, j, k)

            # 同、四辺の場合 (四隅でなかったの場合)
            # 隣接する 3 区画のうち、2 区画以上が砂漠ならば砂漠化
            check_four_sides(dset, j, k)

            # 同、周囲が囲まれている場合 (四隅/四辺でなかった場合)
            # 隣接する 4 区画のうち、3 区画以上が砂漠ならば砂漠化
            check_other_cells(dset, j, k)

    # 結果を表示
    print("")
    print_data(dset)
    return(None)
```

最後に

前の方でも述べましたが、セル・オートマトンを利用することで、様々な自然現象や社会現象を対象としたシミュレーションを実行することができます。今回は、図 8 (10 ページ) のような状態変化規則を定義しましたが、例えば、規則に適合した場合に必ずしも状態を変化させるのではなく確率的に変化させるとか、隣接セルだけではなく 2 セル以上離れたセルも距離に応じた影響を及ぼすなど、対象に合わせて様々な規則を導入することで、シミュレーションの適合範囲を広げることができます。皆さんも、実践的な技法の一つとして、是非覚えておいて下さい。