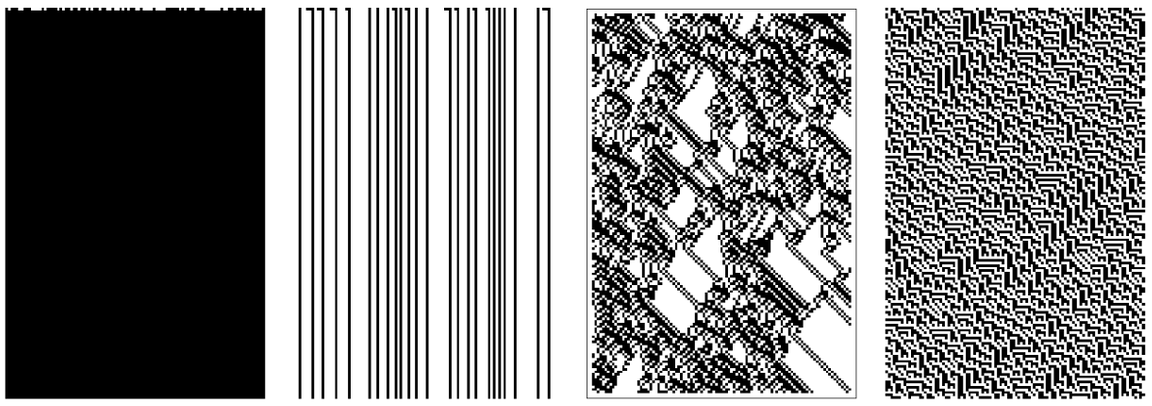
数理生物学演習

第９回　空間構造の数理モデル（２）：人工生命



野下 浩司（Noshita, Koji）

noshita@morphometrics.jp

https://koji.noshita.net

理学研究院 数理生物学研究室

第９回　空間構造の数理モデル（２）：人工生命

本日の目標

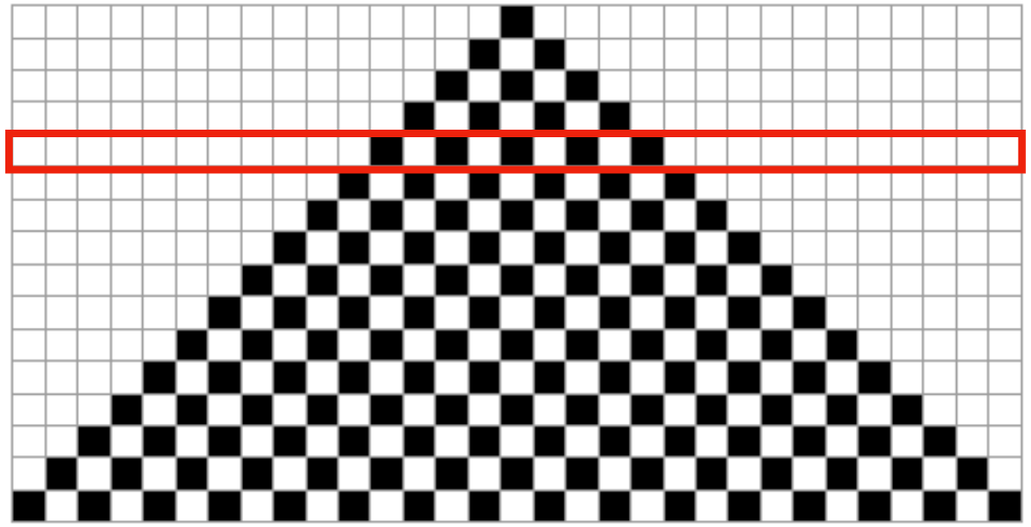
- セル・オートマトン
- ライフゲーム

1次元セル・オートマトン

仮定

- セルが１次元的に並んでいる
- セルは状態“0”または“1”のいずれかをもつ
- セルは自身と近傍の状態により次のステップでの状態が決まる

ある時間での
セルの状態



状態 “0”

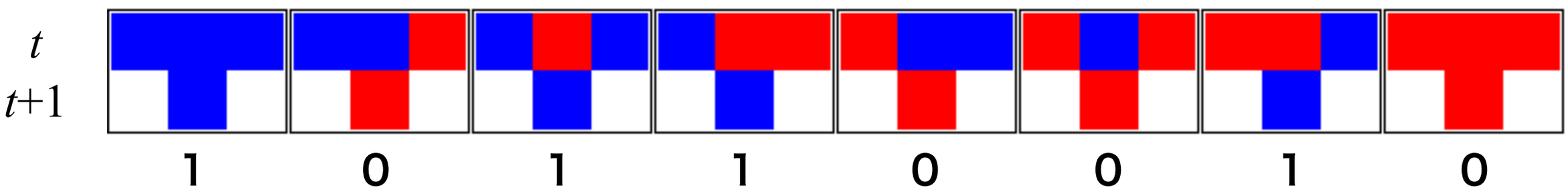
状態 “1”

時間発展を記録していく

状態 “0”

状態 “1”

遷移ルール（２近傍）



$$1\times2^7+0\times2^6+1\times2^5+1\times2^4+0\times2^3+0\times2^2+1\times2^1+0\times2^0=178$$

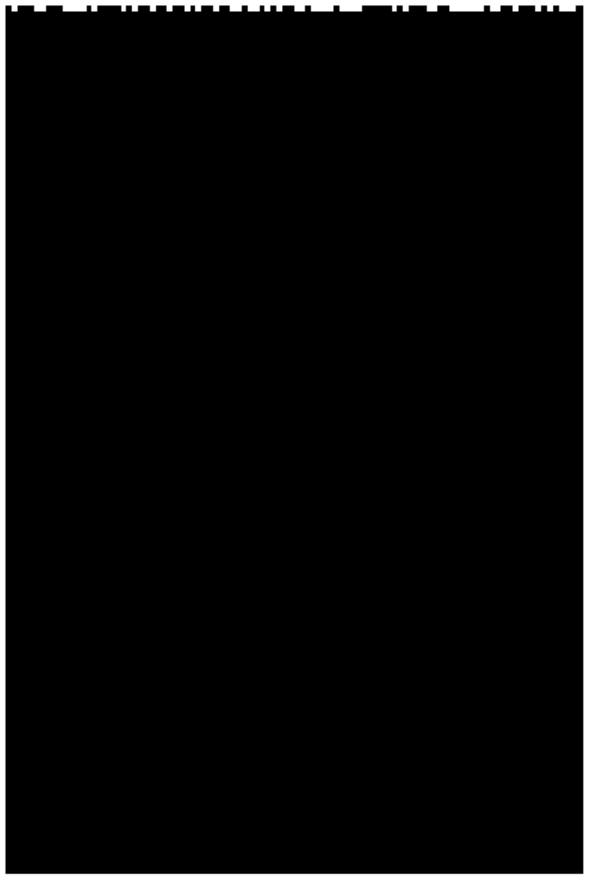
ルール178

ウルフラムのクラス

Wolfram (1983) *Rev. Mod. Phys.*

クラス1

セルの状態が
すべて同じになり，
変化が起こらない．

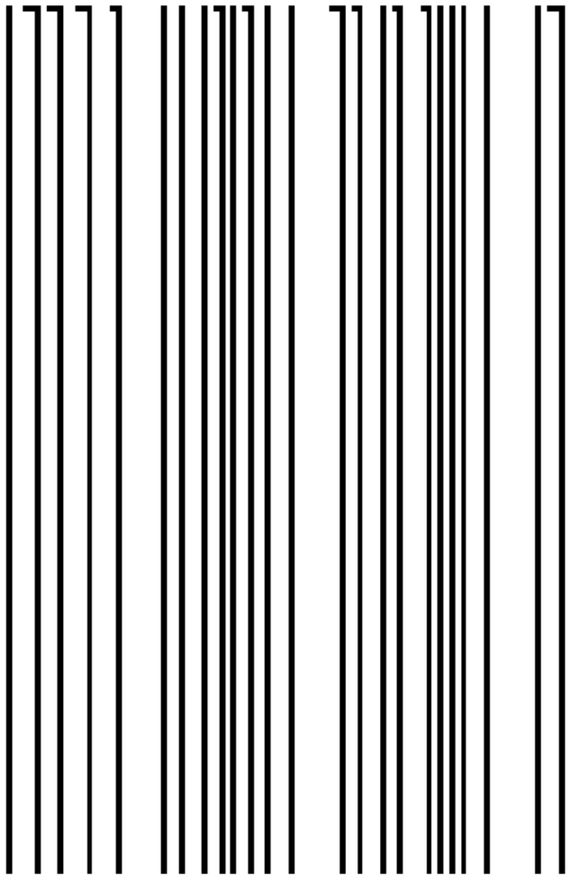


平衡点

秩序
安定

クラス2

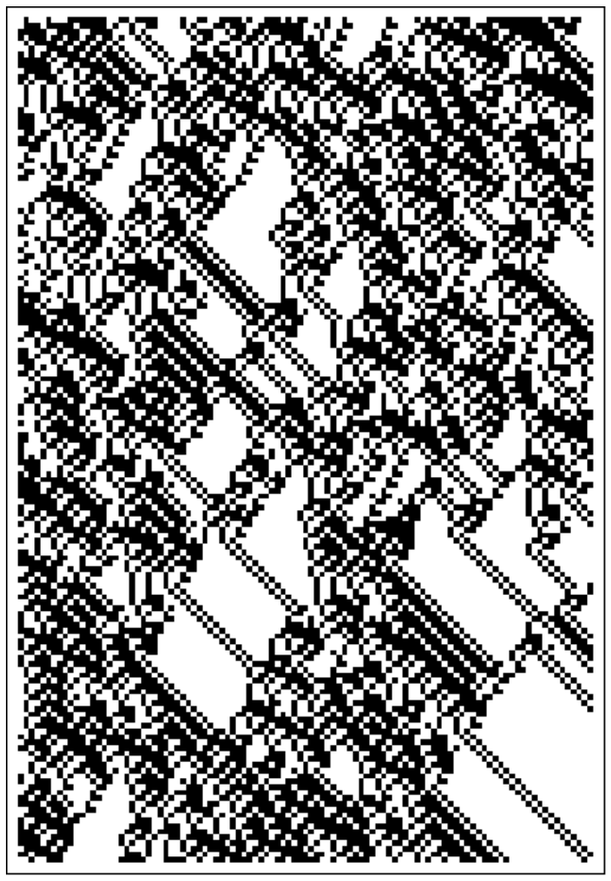
安定したパタンに落ち着き変化が周期的になる．



リミットサイクル

クラス4

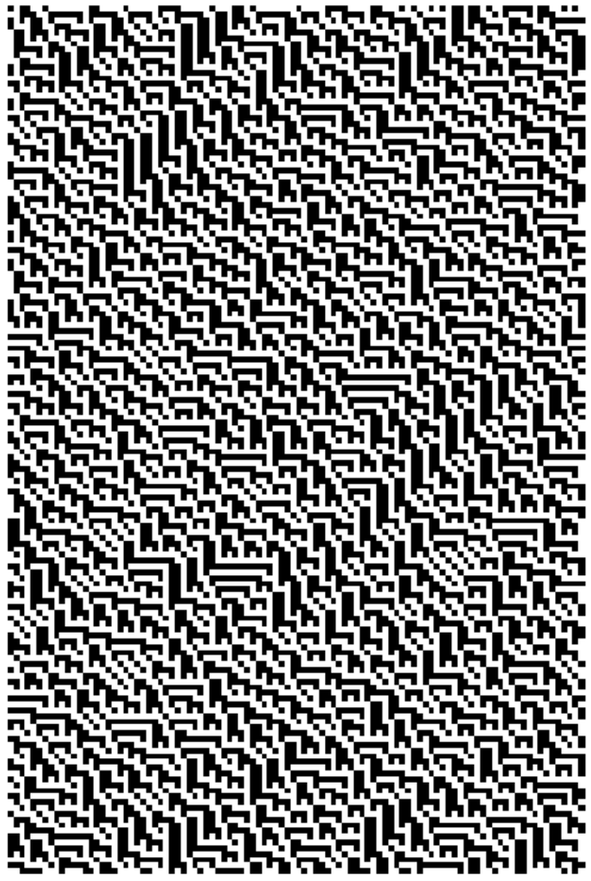
あるときは規則的なパタンを示し，あるときはランダムに振る舞う．



複雑系

クラス3

全体がランダムに振る舞う．ただし，決定論的．



カオス

無秩序
不安定

クラス4で“複雑さ”が最大になる
生命現象はここにあるのかも？

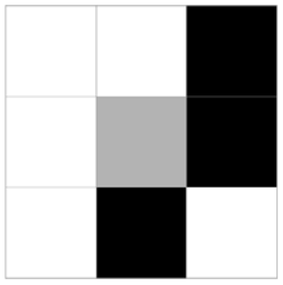
ライフゲーム Conway’s Game of Life

仮定

- **各セルは状態“生”と“死”をもつ**
- **誕生，生存，死亡のプロセスを経て，“生”と“死”の状態を更新する**
- **８近傍のセルの状態により次の状態がきまる**
- **遷移ルールは誕生，維持，過疎，過密の４つ**

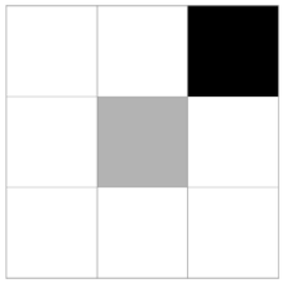
2次元のセル・オートマトンの特殊な場合．かなり色々なパタンが観察できる．

誕生



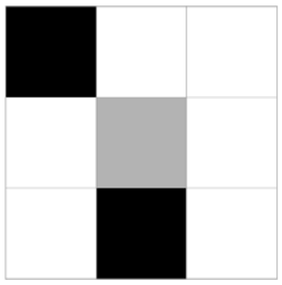
8近傍中
ちょうど3つが“生”ならば
次のステップで“生”

過疎



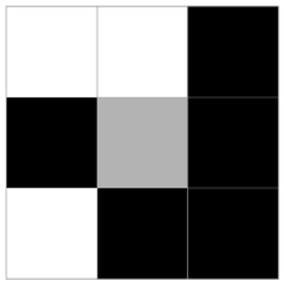
8近傍中
“生”が１つ以下ならば
次のステップで“死”

維持



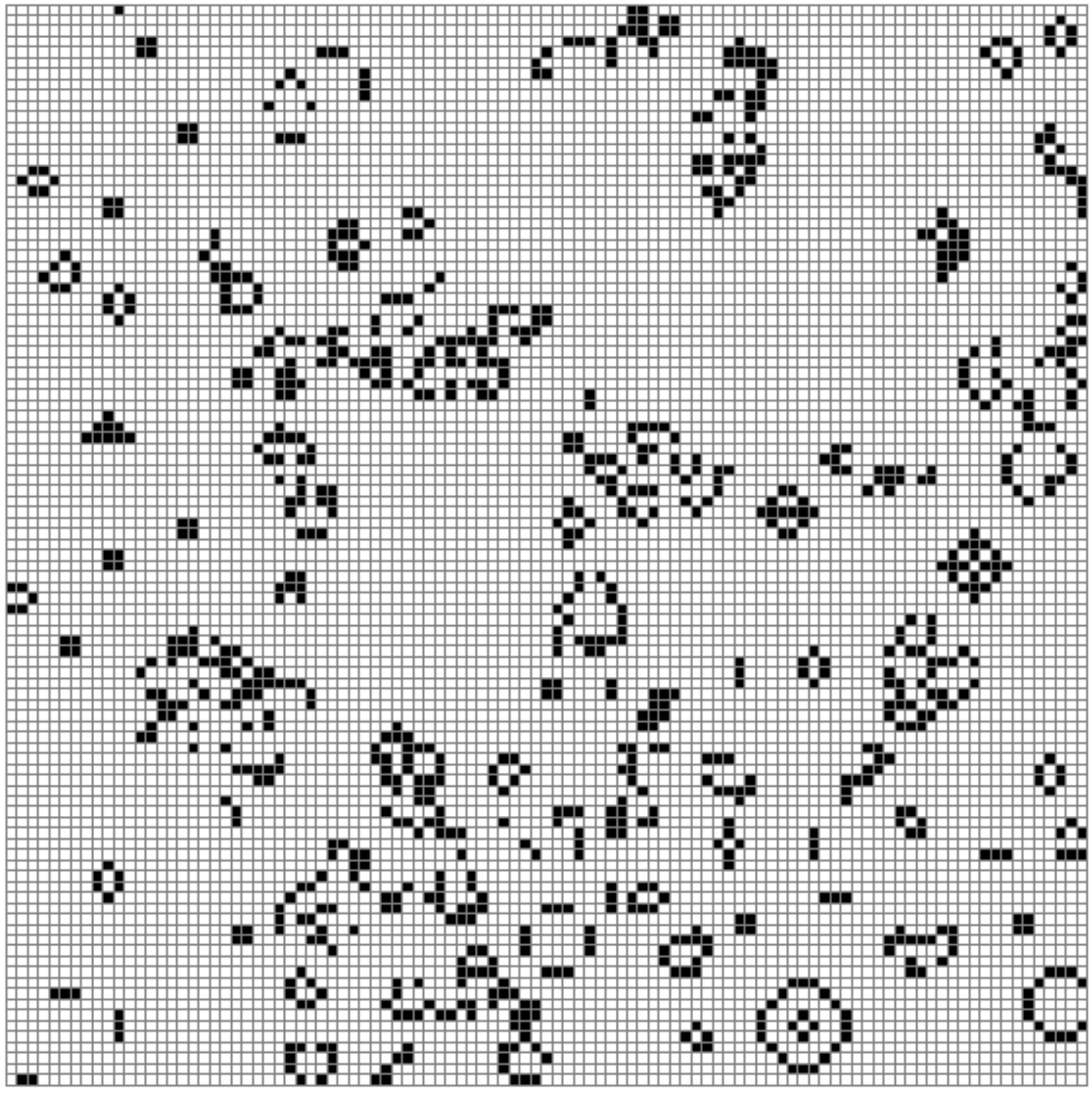
8近傍中
ちょうど２つが“生”ならば
次のステップで更新なし
（“生”ならば“生”，“死”ならば“死”）

過密



8近傍中
“生”が４つ以上ならば
次のステップで“死”

ライフゲームの実行例



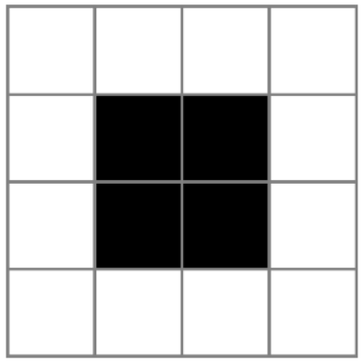
様々なパタンが生成・消失していく．生命のアナロジー？

ライフゲームにみられるパタンいろいろ（１）

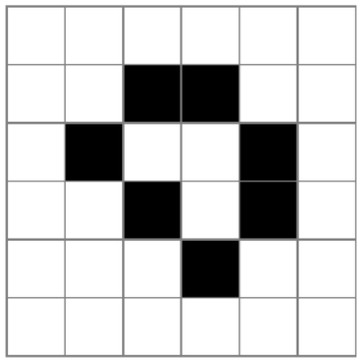
固定物体 still life

全く変化しない

ブロック block



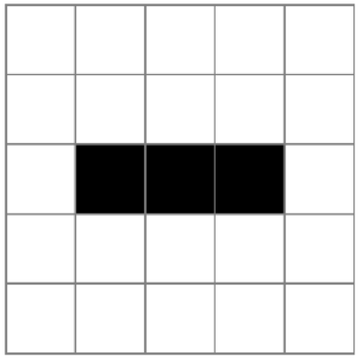
パン loaf



振動子 oscillator

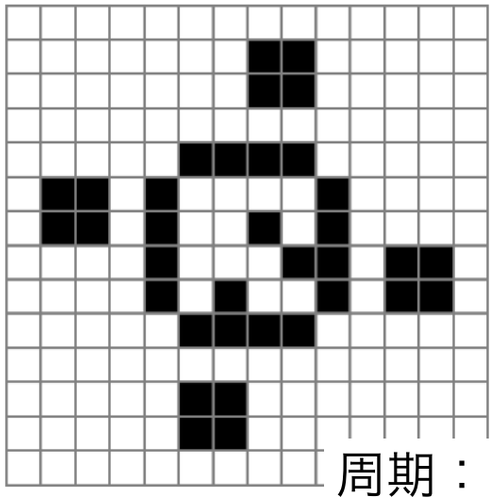
あるパタンを決まった周期で繰り返す
（特定の世代後には同じパタンに戻る）

ブリンカー blinker



周期：２

回転花火 pinwheel

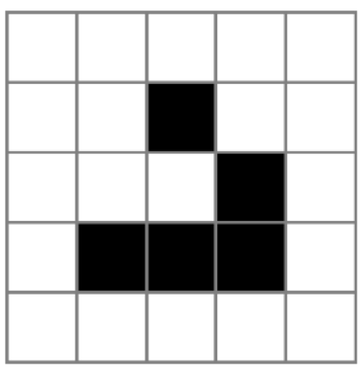


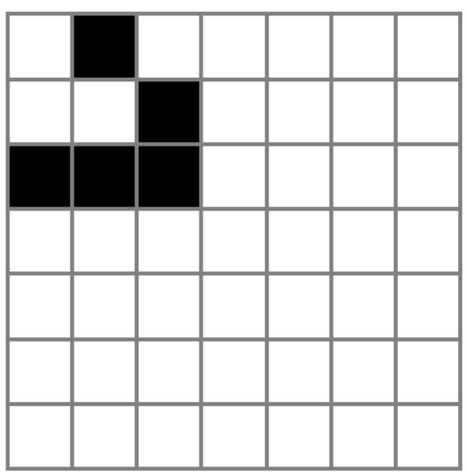
周期：４

移動物体 spaceship

あるパタンを決まった周期で繰り返すが，少なくとも１セル以上移動している.

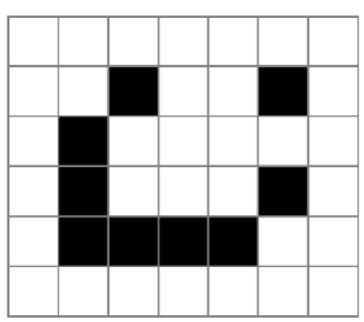
グライダー glider

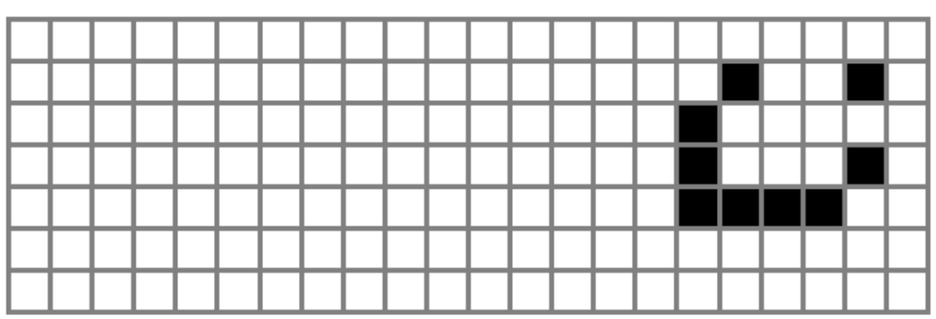




Lightweight spaceship

最も高速（$c/2$）な移動物体の一つ





cは（CAにおける）光速（１世代あたりの情報伝播の上限）

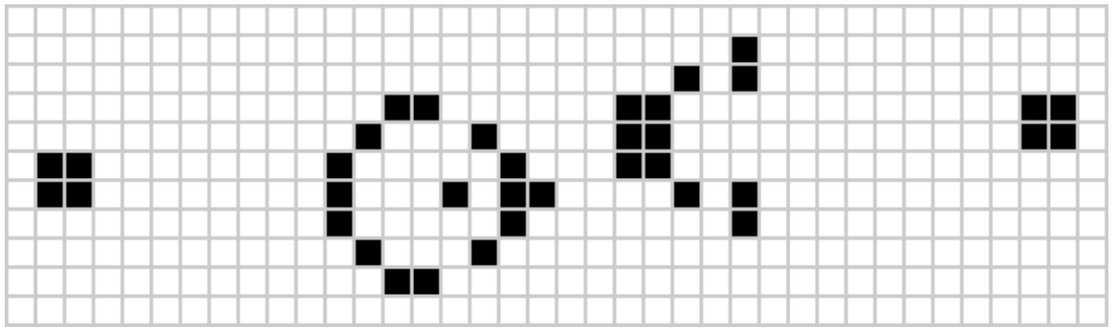
ライフゲームにみられるパタンいろいろ（2）

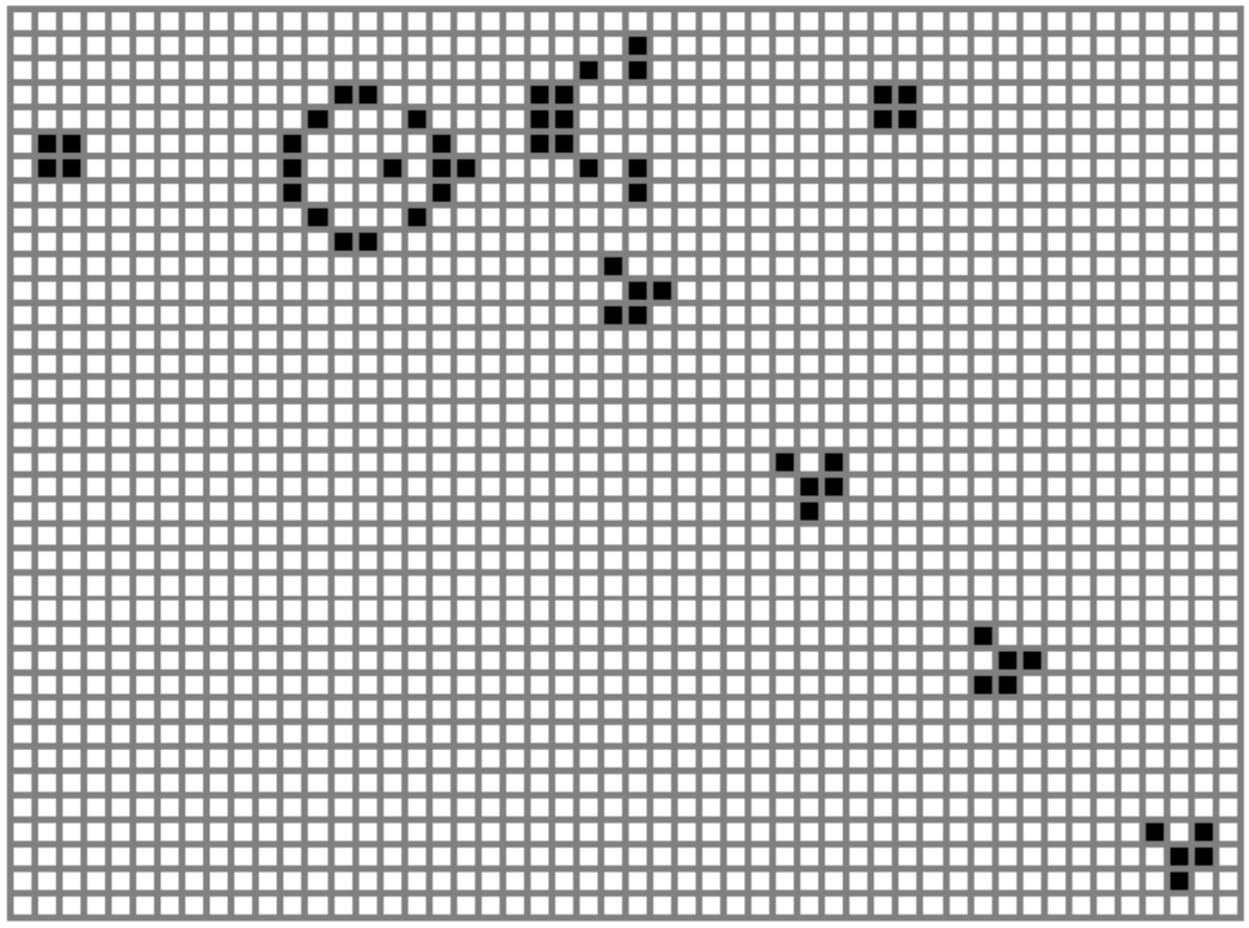
繁殖型

銃 gun

移動物体を発射し続ける安定的（周期的）なパタン

Gosperグライダー銃

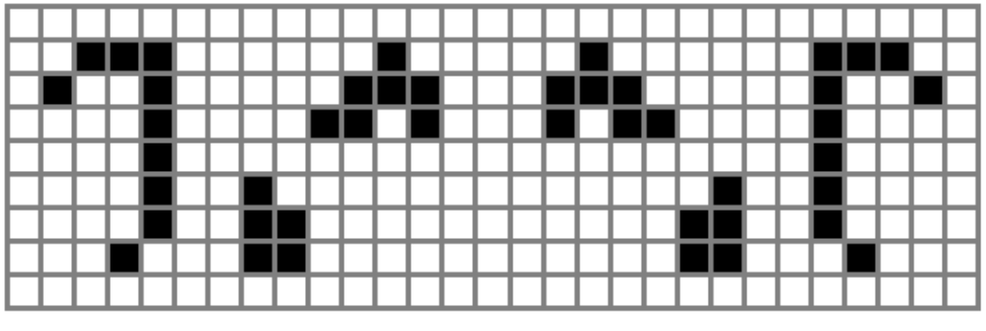


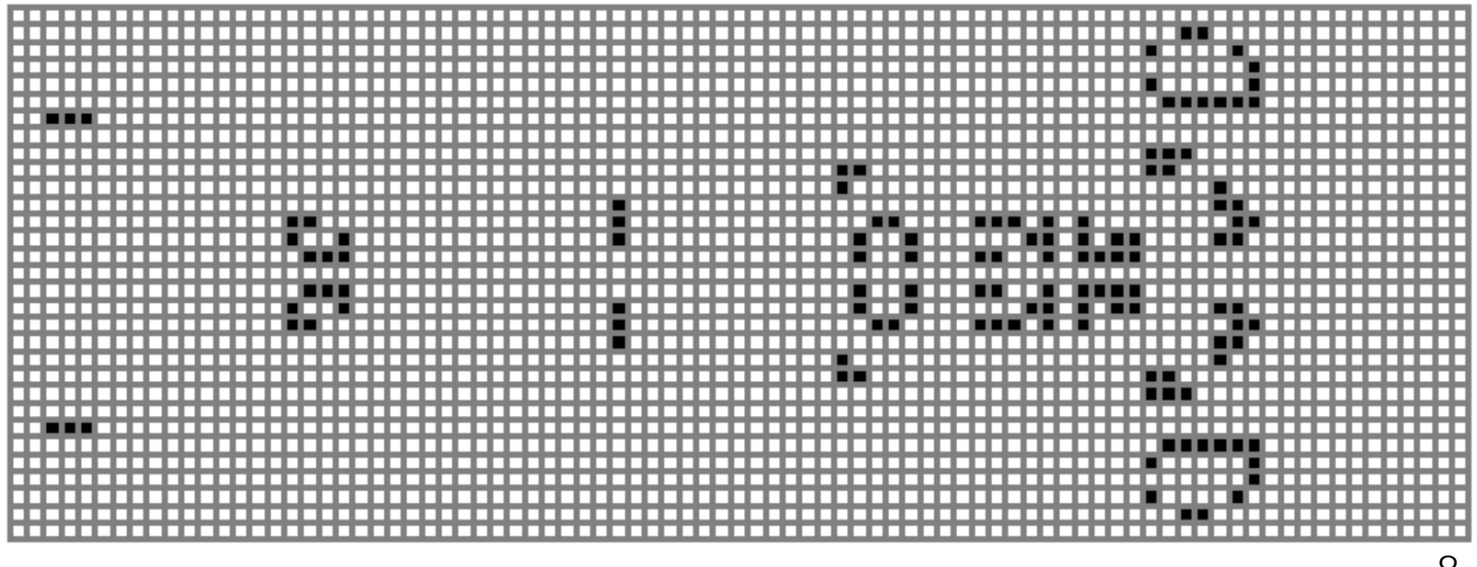


シュシュポッポ列車 puffer

固定物体や振動子などの
“デブリ”を残しながら移動する

Puffer 1





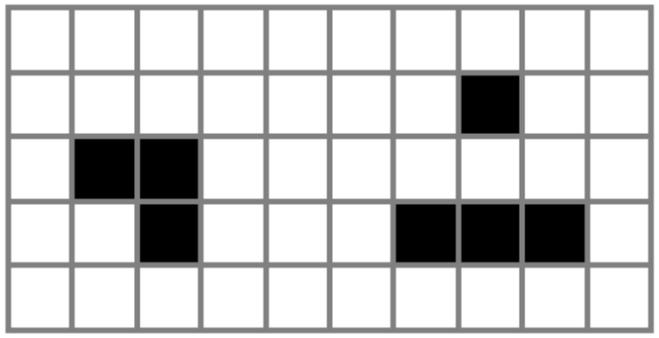
ライフゲームにみられるパタンいろいろ（3）

その他の特徴的なパタン

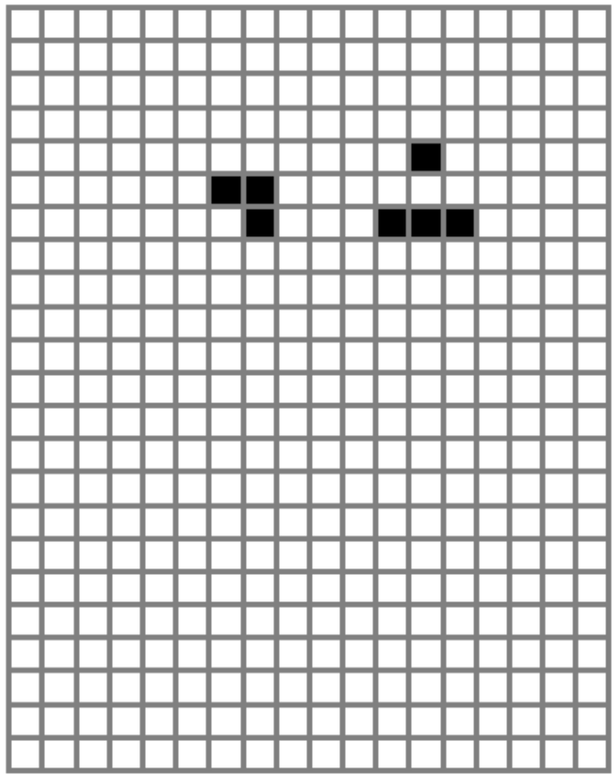
長寿 Methuselah

安定化（消失，固定，周期など）するまでに長い世代を要する

ダイハード



130世代後に消失

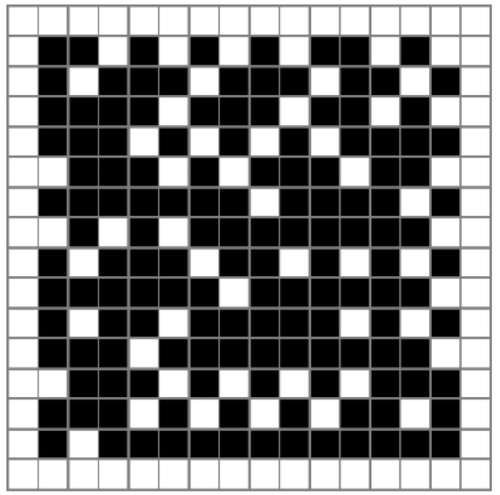


エデンの園配置 Garden of Eden

他のどのパタンからも生成できない

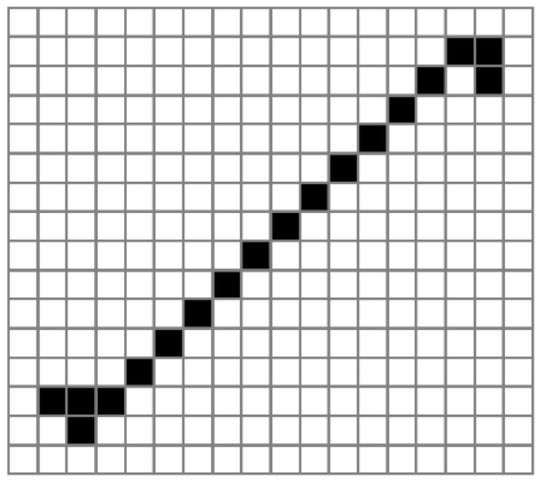
（最初にそのように配置されなければ存在し得ない）

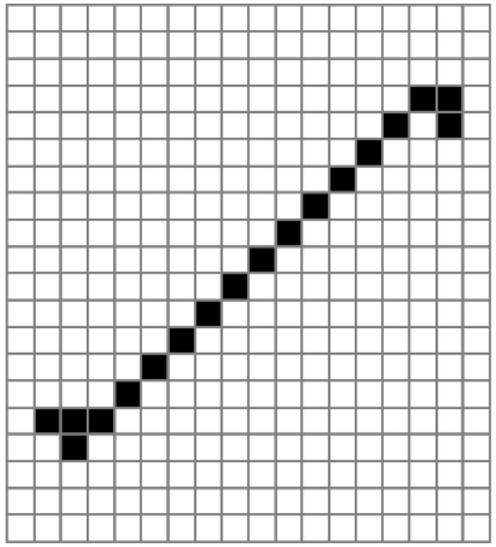
Garden of Eden 2

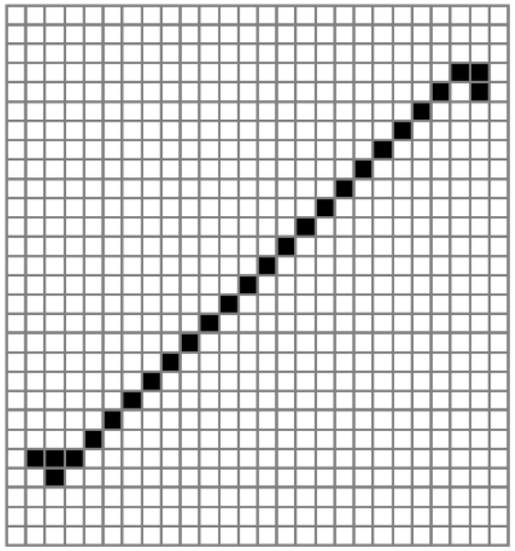


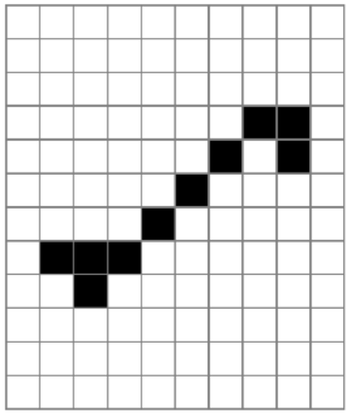
導火線 wick/fuse　安定的（固定/周期）で線的な構造を持ち，一方の端から“火をつける”ことができる

パン屋 baker









長くしたり，短くしたりもOK

もっといろいろなパタンを知りたい人はLifeWiki http://www.conwaylife.com/wikiを見てみよう

セルオートマトンと自己複製

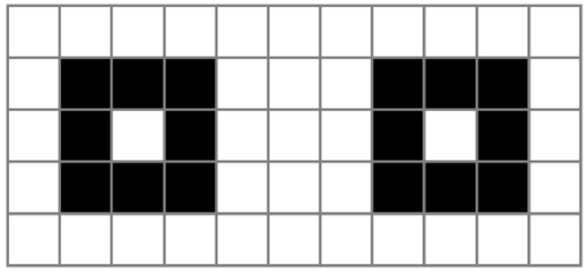
複製子 replicatorが備えるであろう性質　フォン・ノイマンの万能組立機

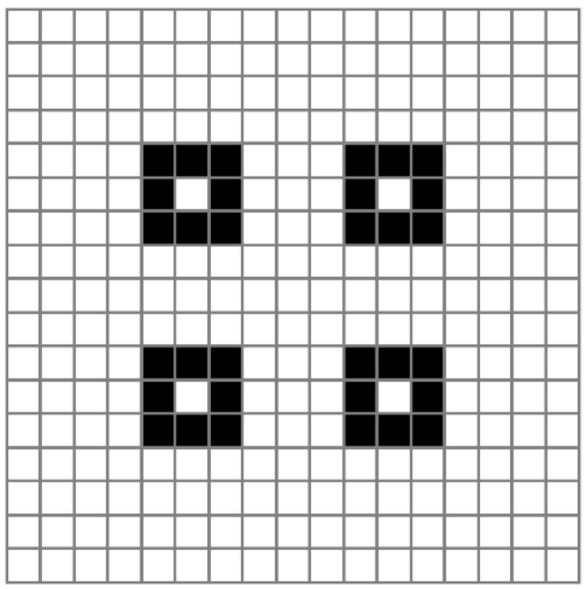
- 設計図のしての側面：次世代に受け渡し可能な自身の組み立て方を示したデータ
- 組立機としての側面：設計図に基づき次世代を組み立てることができるアルゴリズム

Conway’s Game of Lifeにみられる自己複製（？）

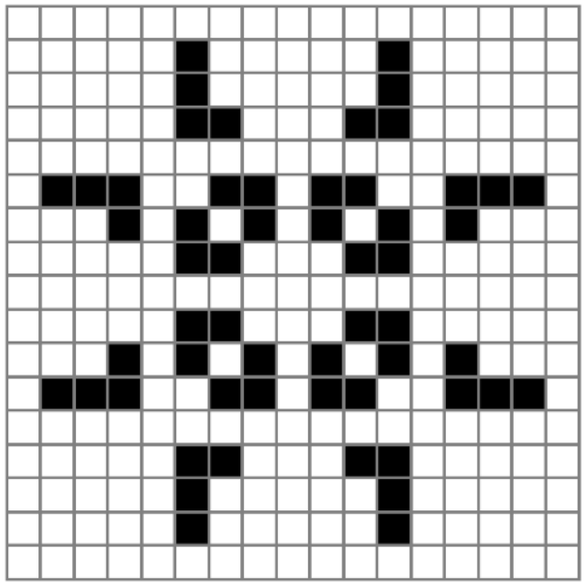
pre-pulsar

15世代後に自身のコピーを作るが，その後パルサーに移行してしまい，それ以上の増殖ができない





15世代後



パルサーは周期３の振動子

線形伝播子 linear propagator

2013年に見つかったConway’s Game of Life上で（おそらく）初の広義の自己増殖するパタン．狭義には自己増殖は２つ以上の自身のコピーを生成するものを指すが，このパタンは自身のコピーを一つだけ残す．Pufferの仲間．14826990セル×14826908セルとめちゃくちゃでかいので可視化が難しい．

http://www.conwaylife.com/wiki/Linear_propagator

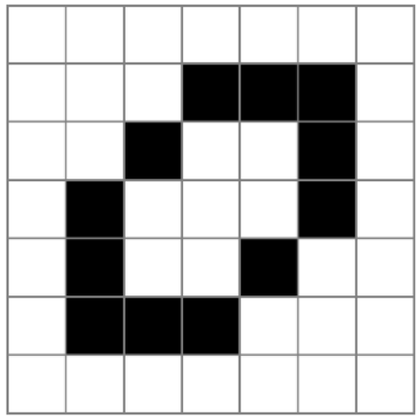
その他のルールに見られる自己複製するパタン

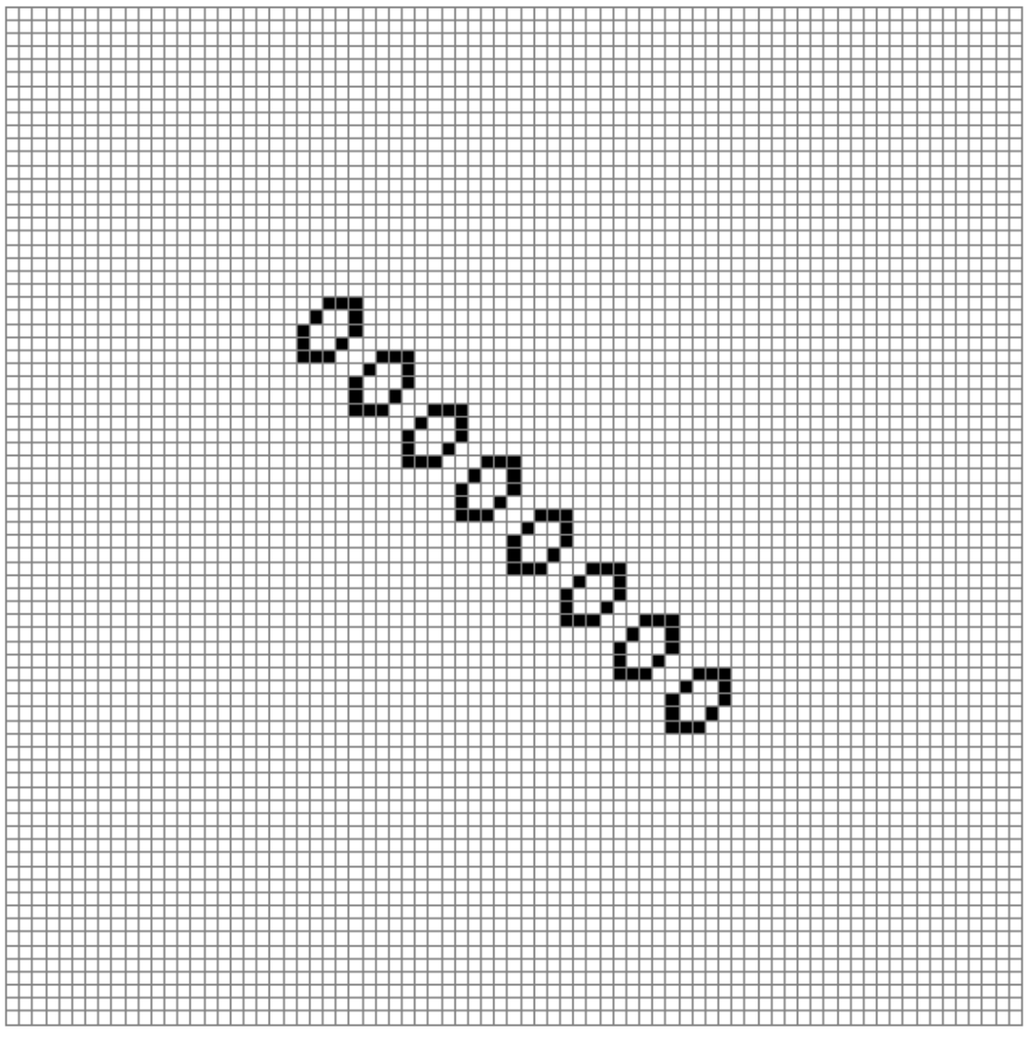
HighLife

8近傍中3 or 6で誕生，2で維持

Replicator

12(2^n-1)世代で2^n個体に増える

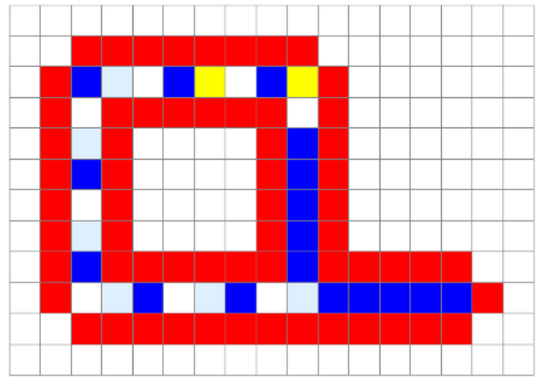


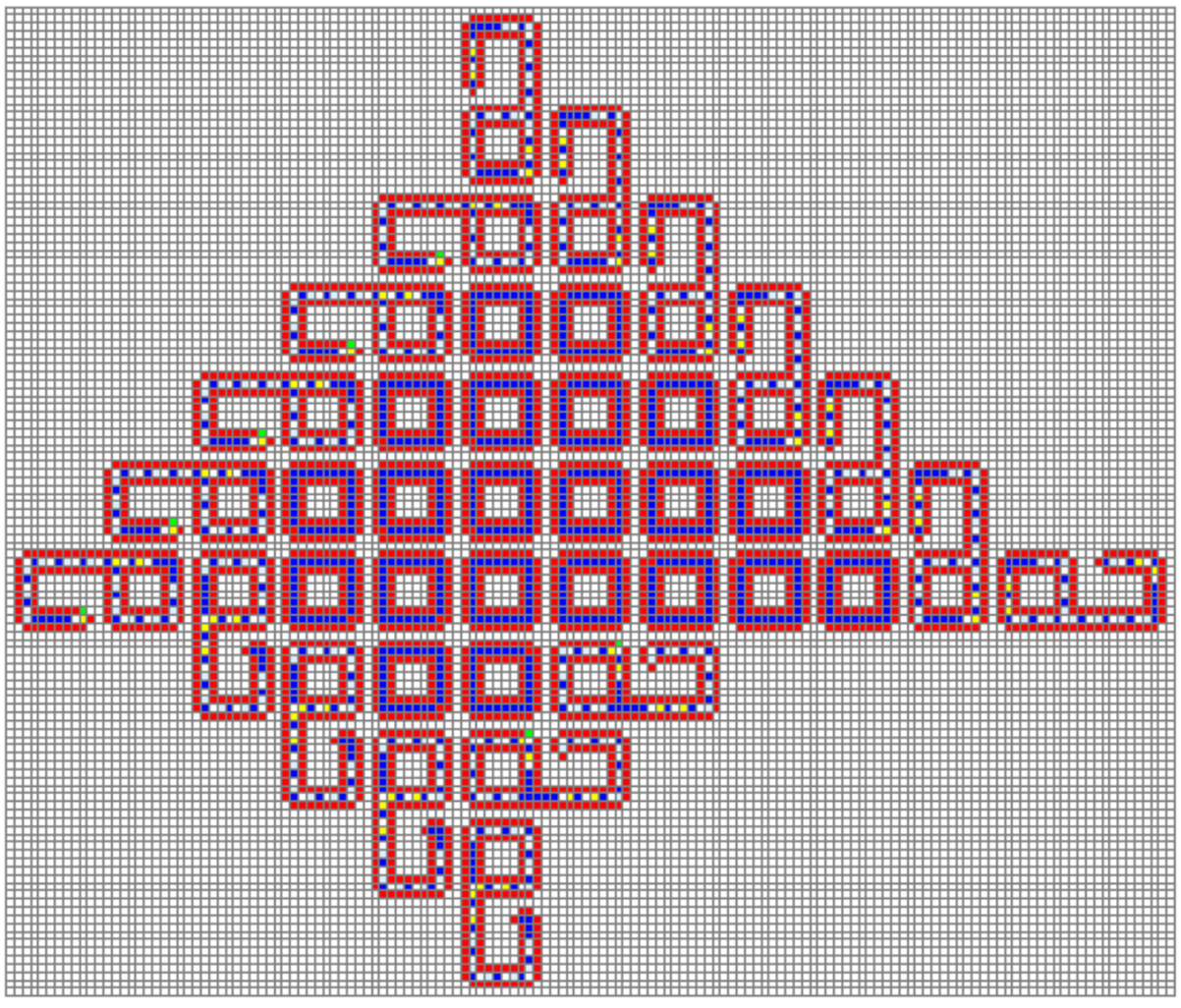


Langton's Loops

8状態，遷移ルールの詳細は
Langton (1984) *Physica D* 参照

151世代でコピーを作る
増殖に伴いコロニーを形成するが内部は増殖できず固定され，表面のみ成長を続ける



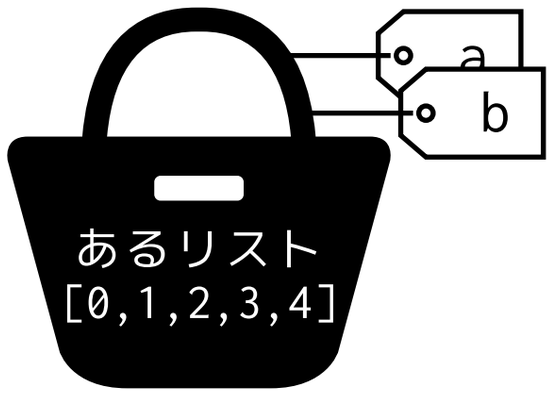


実際にプログラムを組んでみよう！

リストや配列のコピーの補足

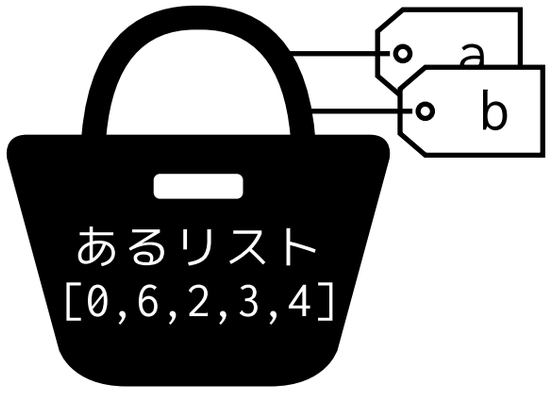
代入：オブジェクトに
ラベルをつける操作

```
a = [0,1,2,3,4]
```



別のラベルをつける

```
b = a
```

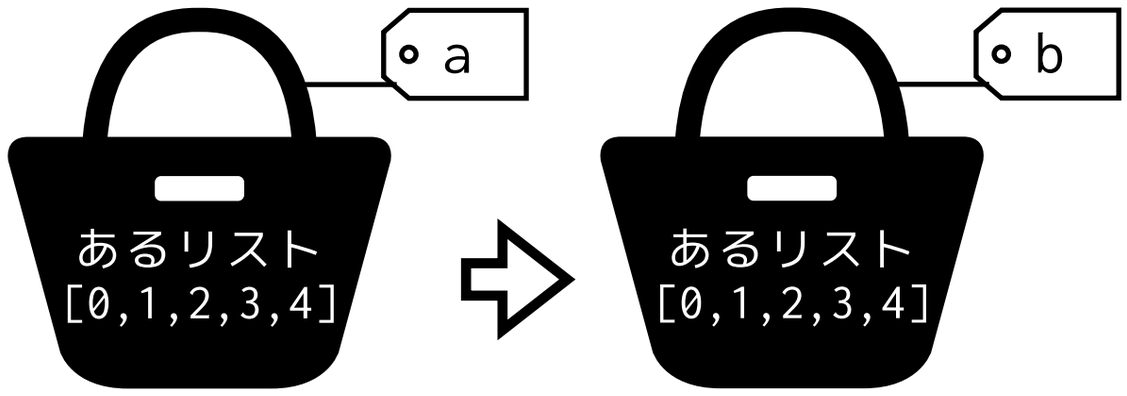


要素への代入は同じオブジェクトを
参照しているものへもの影響する

```
b[1] = 6
print(a)
→ [0,6,2,3,4]
```

ここではbの要素を書き換えたら
aにも影響が及ぶ

そこでcopy



同じ要素への参照（ラベル）を
もつ別のオブジェクトを作る

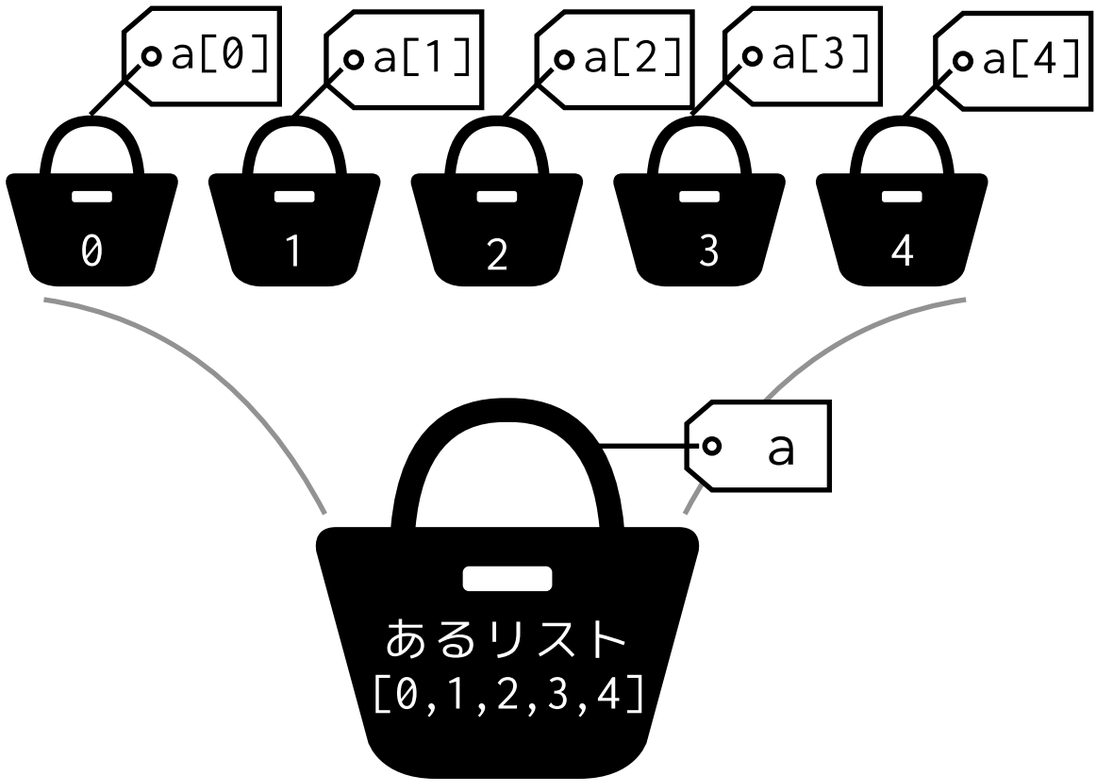
```
a = [0,1,2,3,4]
b = copy.copy(a)
```

もしくはdeepcopy

```
a = [0,1,2,3,4]
b = copy.deepcopy(a)
```

copyは浅いコピー（要素は参照）
deepcopyは深いコピー（要素も含め完全な別物）
詳しく知りたい人は公式ドキュメント参照
https://docs.python.org/ja/3/library/copy.html

リストや配列のコピーの補足（2）

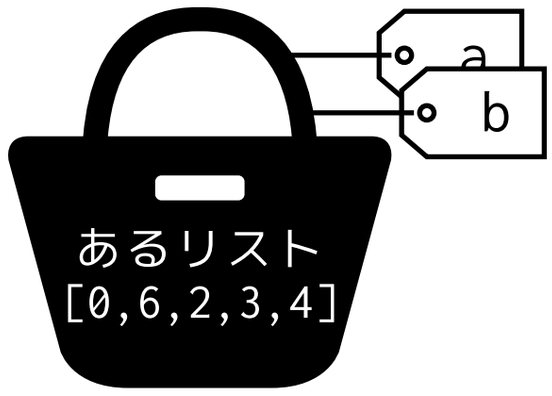


リストの要素についても同様

```
a = [0,1,2,3,4]
```

別のラベルをつける

```
b[1] = a[1]
```



要素への代入は同じオブジェクトを参照しているものへもの影響する

```
b[1] = 6
print(a)
→ [0,6,2,3,4]
```

ここではbの要素を書き換えたらaにも影響が及ぶ

NumPyでの行列計算の復習（１）：第７回から一部抜粋

NumPyのndarray（配列）を
ベクトルや行列と見立てて計算する

- １次元配列 → ベクトル
- 2次元配列 → 行列

```
# ndarray

import numpy as np

# ndarrayの定義
a = np.array([1,2,3])
b = np.array([6, 3.3, 1])
C = np.array([[1, 5, 6],
              [7, 8, 9],
              [4, 2, 3]])
D = np.array([[2.3, 4, 7.2],
              [7, 9, 1],
              [11, 2, 9]])
```

```
# ベクトル・行列計算

## ベクトルの基本演算
print("a+b: ", a + b)
print("a-b: ", a - b)
print("3*a: ",3 * a)

## ベクトルの内積・外積
print("a.b, np.dot(a,b): ", np.dot(a,b))
print("axb, np.cross(a,b): ", np.cross(a,b))

## 行列の基本演算
print("C+D: \n", C + D)
print("C-D: \n", C - D)
print("2*C: \n", 2 * C)

## 行列の乗算
print("C.a, np.dot(C,a): ", np.dot(C,a))
print("C.D, np.dot(C,D): \n", np.dot(C,D))
print("D.C, np.dot(D,C): \n", np.dot(D,C))
```

$\mathbf{a}+\mathbf{b}$ $3\mathbf{a}$ $\mathbf{a}\cdot\mathbf{b}$ $\mathbf{a}\times\mathbf{b}$ $\mathbf{C}-\mathbf{D}$ $2\mathbf{C}$ $\mathbf{Ca}$ $\mathbf{CD}$ $\mathbf{DC}$

$$\mathbf{a} = (1,2,3) \qquad \mathbf{b} = (6,3.3,1)$$

$$\mathbf{C} = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 8 & 9 \\ 4 & 2 & 3 \end{pmatrix} \qquad \mathbf{D} = \begin{pmatrix} 2.3 & 4 & 7.2 \\ 7 & 9 & 1 \\ 11 & 2 & 9 \end{pmatrix}$$

NumPyでの行列計算の復習（２）：第７回から一部抜粋

NumPyのndarray（配列）を
ベクトルや行列と見立てて計算する

- １次元配列 → ベクトル
- 2次元配列 → 行列

$$\mathbf{a} = (1,2,3)$$
$$\mathbf{b} = (6,3.3,1)$$
$$\mathbf{e} = (1.0,2.0,3.0)$$

$$\mathbf{C} = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 8 & 9 \\ 4 & 2 & 3 \end{pmatrix} \quad \mathbf{D} = \begin{pmatrix} 2.3 & 4 & 7.2 \\ 7 & 9 & 1 \\ 11 & 2 & 9 \end{pmatrix}$$

$$\mathbf{F} = \begin{pmatrix} 2 & 4 & 7 \\ 7 & 9 & 1 \\ 11 & 2 & 9 \end{pmatrix}$$

```
# 線形代数向け関数
# 転置行列
print("C^T, C.transpose(): ", C.transpose())
print("C^T, np.transpose(C): ", np.transpose(C))
# 行列式
print("|D|, np.linalg.det(D): ", np.linalg.det(D))
# 逆行列
print("F^-1, np.linalg.inv(F): ”, np.linalg.inv(F))
# 固有値・固有ベクトル
print(“np.linalg.eig(C): ", np.linalg.eig(C))
print("固有値のみ, np.linalg.eigvals(C): ”, np.linalg.eigvals(C))
```

$\mathbf{C}^T$

$|\mathbf{D}|$

$\mathbf{F}^{-1}$

固有値： $\lambda_1, \lambda_2, \lambda_3$
固有ベクトル： $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$

NumPyの配列操作：インデックス，スライス（１）

インデックスやスライスをもちいて要素や配列の集まりへのアクセスができる
のはリストと同様だが，より高機能なアクセスが可能

```
# 01-01. 配列
import numpy as np

a = np.array([1, 2, 3])
b = np.array([6, 3.3, 1])
C = np.array([[1, 5, 6], [7, 8, 9], [4, 2, 3]])
D = np.array([[2.3, 4, 7.2], [7, 9, 1], [11, 2, 9]])
```

$$\mathbf{a} = (1,2,3) \qquad \mathbf{C} = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 8 & 9 \\ 4 & 2 & 3 \end{pmatrix}$$

$$\mathbf{b} = (6,3.3,1)$$

$$\mathbf{D} = \begin{pmatrix} 2.3 & 4 & 7.2 \\ 7 & 9 & 1 \\ 11 & 2 & 9 \end{pmatrix}$$

```
# 01-02. インデックスの基礎
print(a[0])
print("aの末尾からのカウント： ", a[-1])
print("２次元 列へのアクセス： ", C[0])
print("2次元 要素へのアクセス", C[0, 1])
```

```
# 出力
1
aの末尾からのカウント：  3
２次元 行へのアクセス：  [1 5 6]
2次元 要素へのアクセス 5
```

```
# 01-03. スライスの基礎
print("0~(2-1)まで1ステップ刻み: ", b[0:2:1])
print("2次元 0~(2-1)まで1ステップ刻み 列: \n", D[0:2])
print("2次元 0~(2-1)まで1ステップ刻み 行と列両方: \n", D[0:2, 0:2])
```

```
# 出力
0~(2-1)まで1ステップ刻み:  [6.  3.3]
2次元 0~(2-1)まで1ステップ刻み 列:
 [[2.3 4.  7.2]
 [7.  9.  1. ]]
2次元 0~(2-1)まで1ステップ刻み 行と列両方:
 [[2.3 4. ]
 [7.  9. ]]
```

NumPyの配列操作：インデックス，スライス（2）

インデックスやスライスをもちいて要素や配列の集まりへのアクセスができる
のはリストと同様だが，より高機能なアクセスが可能

$$\mathbf{C} = \begin{pmatrix} 1 & 5 & 6 \\ 7 & 8 & 9 \\ 4 & 2 & 3 \end{pmatrix}$$

Cの要素や行，列を取り出してみよう

```
# 01-04. リストを使った行列の一部へのアクセス
print("0,1行と0,1列: \n", C[[0, 1]][:, [0, 1]])
```

```
# 出力
0,1行と0,1列:
 [[1 5]
 [7 8]]
```

```
# 01-05. インデックスとスライスの組み合わせ
print("２次元 列へのアクセス： ", C[:, 0])
```

```
# 出力
２次元 列へのアクセス：  [1 7 4]
```

```
# 01-06. リストを使ったインデックスアクセス
print("0,1行: \n", C[[0, 1]])
print("0,1列: \n", C[:, [0, 1]])
```

```
# 出力
0,1行:
 [[1 5 6]
 [7 8 9]]
0,1列:
 [[1 5]
 [7 8]
 [4 2]]
```

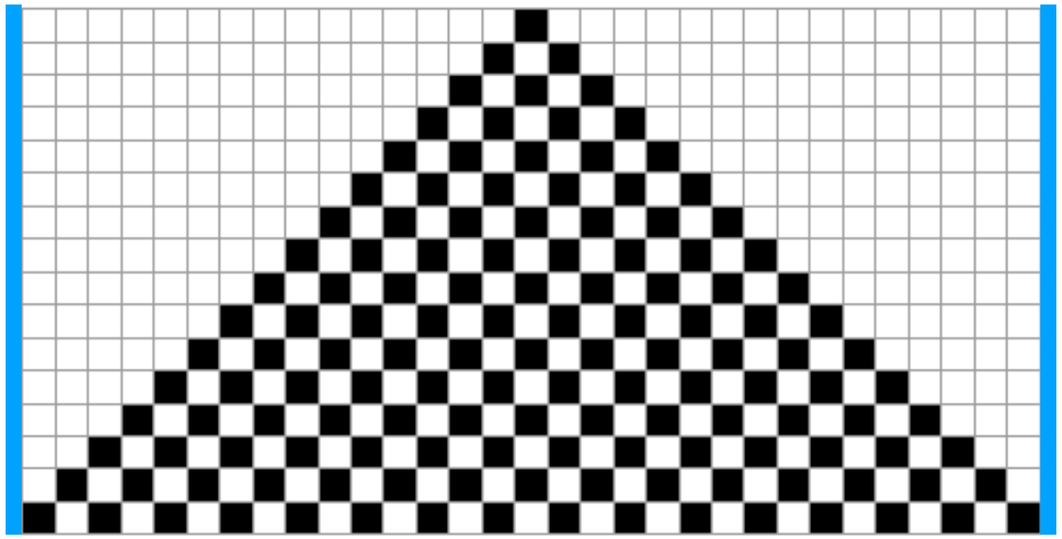
ライフゲームのプログラミングの際にも利用してみよう！

境界条件

プログラムを組むときも，
この部分の処理は注意！

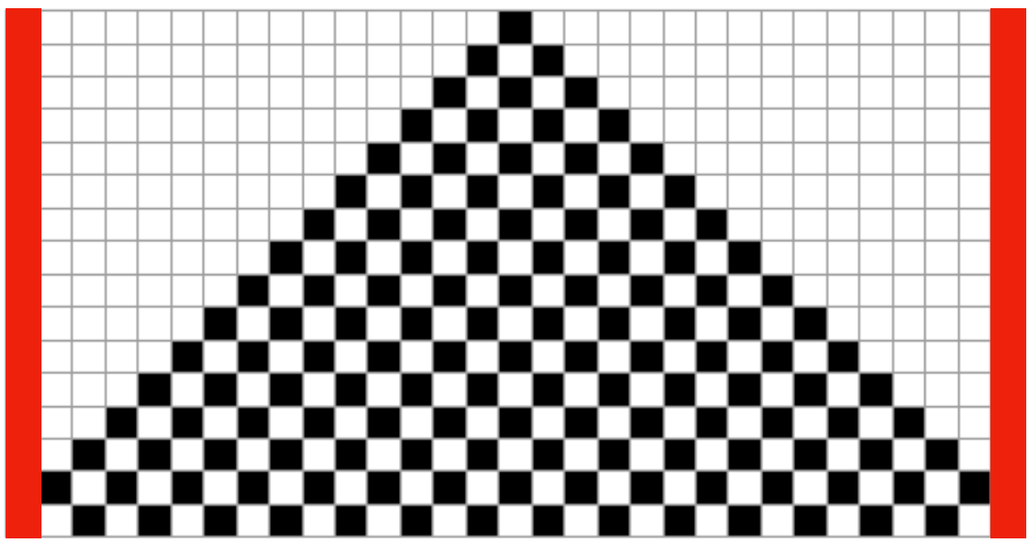
周期境界条件

“端”同士が張り合わされていると考える.



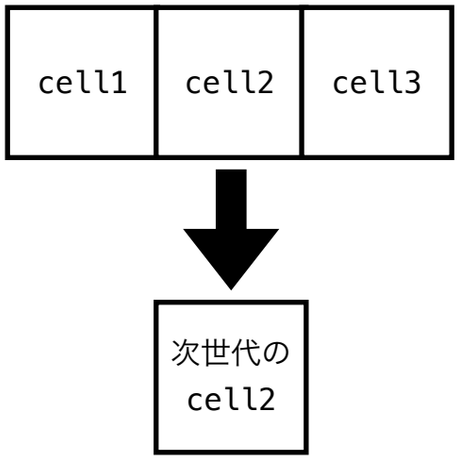
固定端

“端”の値を与えて，変動しないとする.



例えば，この端で常に状態“０”

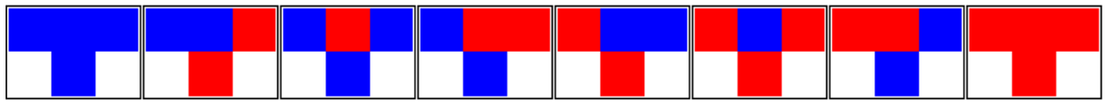
02-01.ルール178を実装してみよう



自身とその両近傍の状態に応じ，
次世代の状態が決まる

状態 “0”
状態 “1”

ルール178



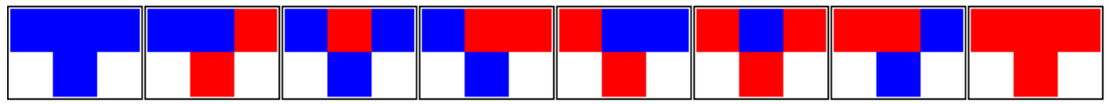
```python
# 02-01-01. ルール178
def rule178(cell1, cell2, cell3):
    if cell1 == 0:
        if cell2 == 0:
            if cell3 == 0:
                return 0
            elif cell3 == 1:
                return 1
        elif cell2 == 1:
            if cell3 == 0:
                return 0
            elif cell3 == 1:
                return 0
    elif cell1 == 1:
        if cell2 == 0:
            if cell3 == 0:
                return 1
            elif cell3 == 1:
                return 1
        elif cell2 == 1:
            if cell3 == 0:
                return 0
            elif cell3 == 1:
                return 1
```

```python
# 確認
for cell1 in range(2):
    for cell2 in range(2):
        for cell3 in range(2):
            print(cell1, cell2, cell3, ": ", rule178(cell1, cell2, cell3))
```

```
0 0 0 : 0
0 0 1 : 1
0 1 0 : 0
0 1 1 : 0
1 0 0 : 1
1 0 1 : 1
1 1 0 : 0
1 1 1 : 1
```

あまり賢いルールの定義の仕方ではない．
余裕のある人はもっと良い実装方法を考えてみよう．

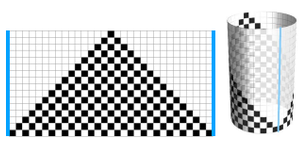
02-01.ルール178を実装してみよう



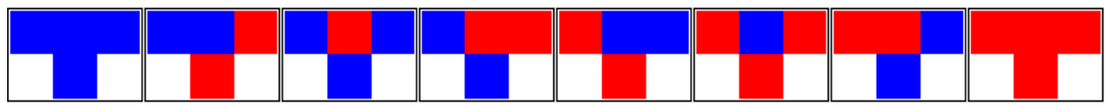

```python
# 02-01-02. セルオートマトンの実行
# パッケージの読み込み等
import numpy as np
import matplotlib.pyplot as plt

num_cell = 100  # セルの数
steps = 50  # 何ステップまで計算するか

cell_list = []  # 各世代のセルの情報を記録するリスト
cell = np.zeros(num_cell, dtype=int)  # セルの初期化

cell[int(num_cell / 2)] = 1  # 一つのセルだけに1を代入
cell_list.append(cell)
for t in range(steps):
    # -- 状態遷移 --
    cell_tmp = np.ones(num_cell, dtype=int)
    # 境界条件処理その１
    cell_tmp[0] = rule178(cell[num_cell - 1], cell[0], cell[1])
    # メイン
    for i in range(1, num_cell - 1, 1):
        cell_tmp[i] = rule178(cell[i - 1], cell[i], cell[i + 1])
    # 境界条件処理その２
    cell_tmp[num_cell - 1] = rule178(cell[num_cell - 2], cell[num_cell - 1], cell[0])

    # 情報の更新
    cell = np.copy(cell_tmp)
    cell_list.append(cell)
```

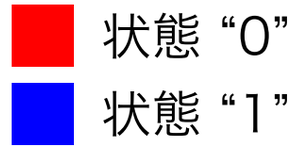
次世代の情報を
一時的に記録するセル

周期境界条件

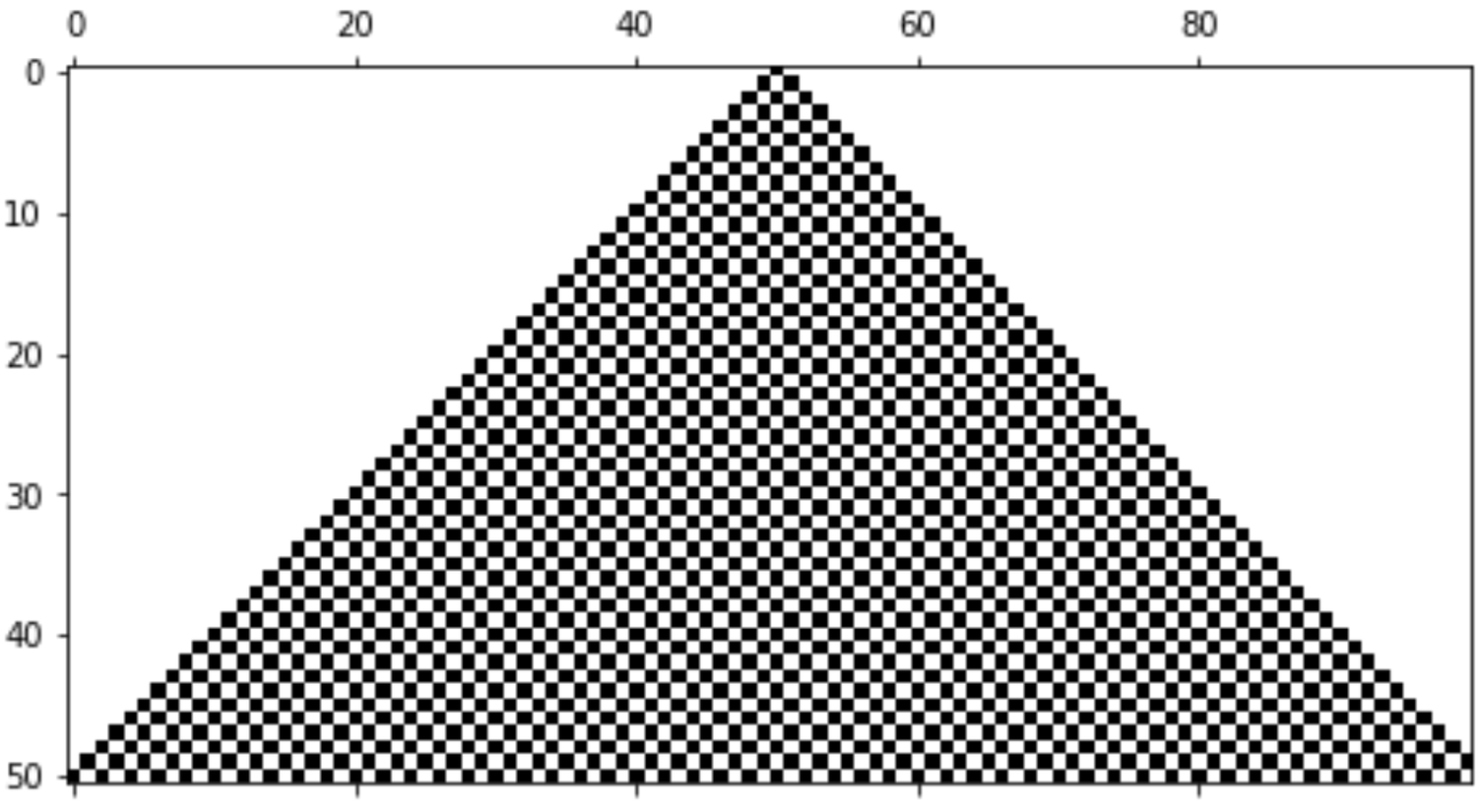


02-01.ルール178を実装してみよう





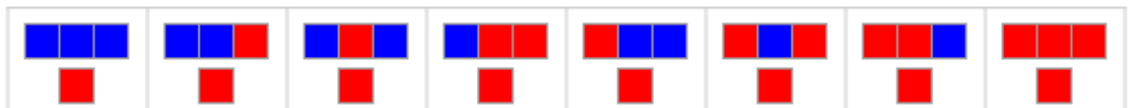

```
# 02-01-03. セルオートマトンの可視化
plt.figure(dpi=300)
plt.matshow(cell_list[:], cmap="binary")
```

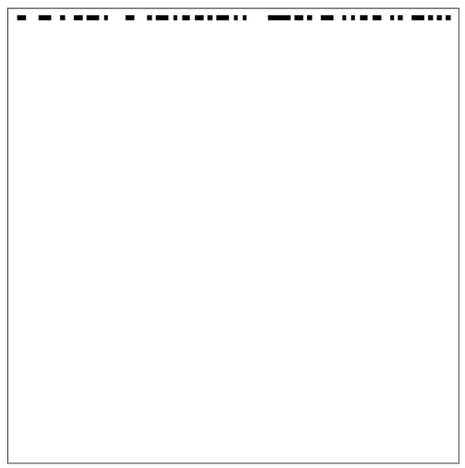


- matplotlib.pyplot
  - matshow(リストや配列)
    リストや配列の可視化

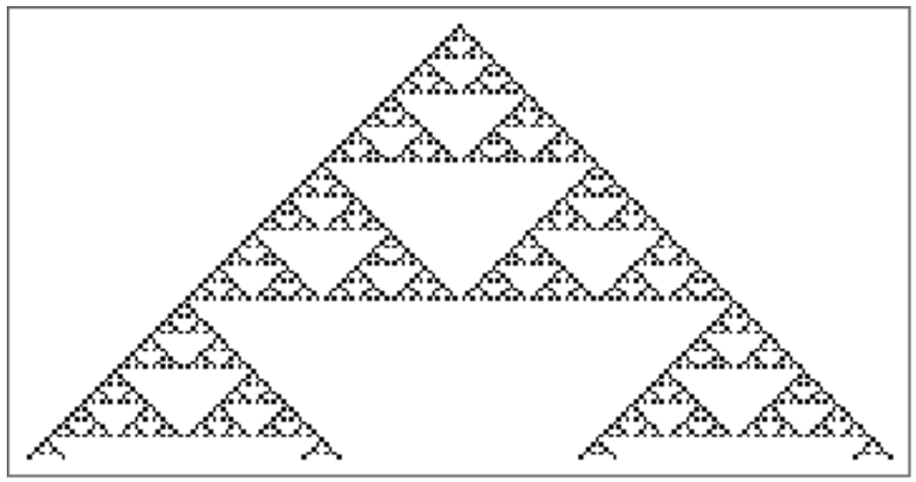
02-02
別のルールを実装してみよう

ルール0（クラス１）

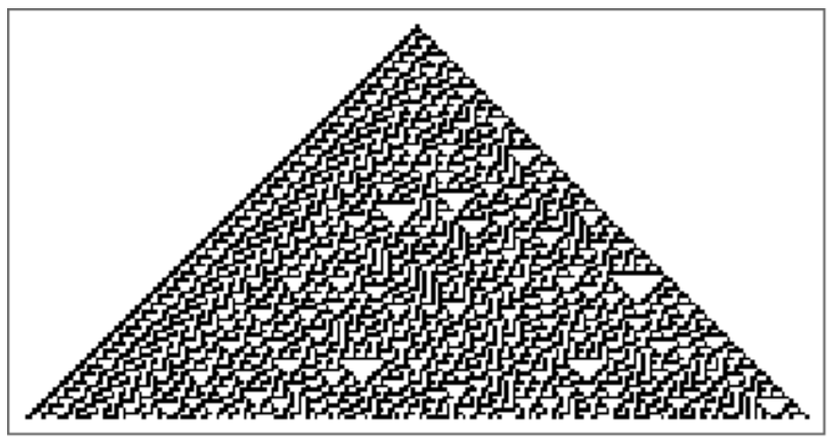




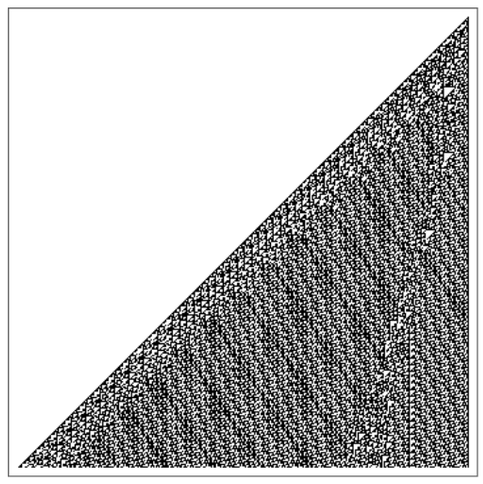
ルール90（クラス２）



ルール30（クラス３）



ルール110（クラス４）



状態 “0”

状態 “1”

興味のある人は4近傍へもチャレンジしてみよう

ライフゲーム

03.
ライフゲームのプログラムを組んでみてください.

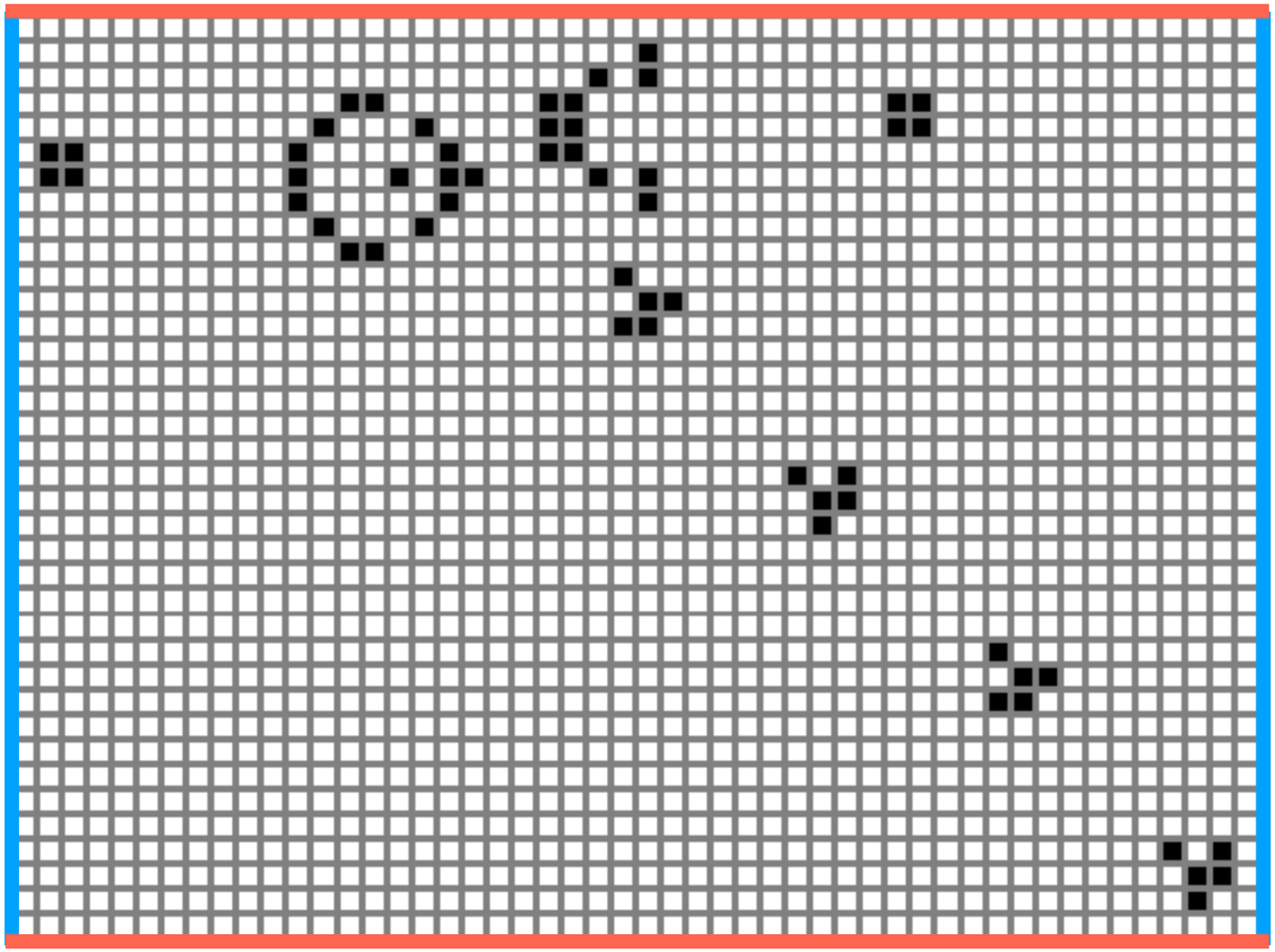
- 境界条件は周期または固定のいずれか好きな方を採用して良い
  （ただし固定端の場合は境界はすべて“死”）
- 格子のサイズは50×50で作る（もっと大きくしても良い）

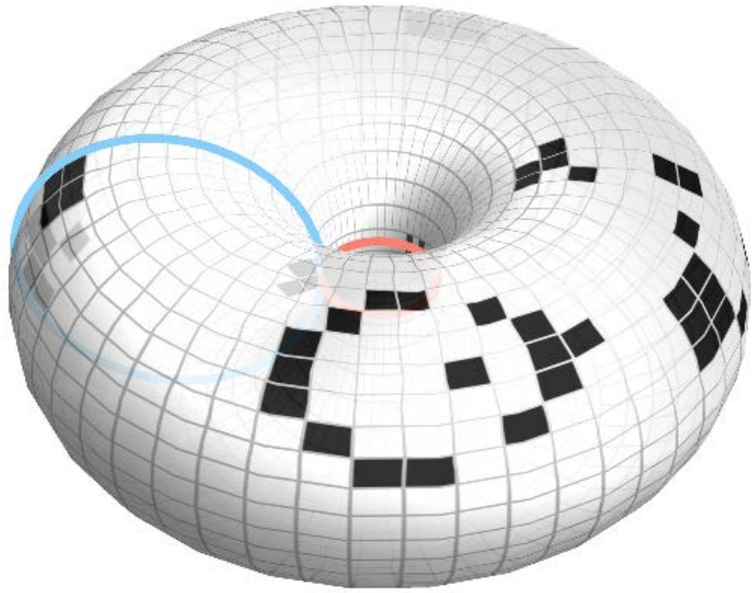
やるべきこと
- 初期条件の設定
- 状態遷移ルールの実装
- 境界条件の処理
- 結果の記録
- データの可視化

境界条件 ２次元（１）

周期境界条件

“端”同士が張り合わされていると考える.



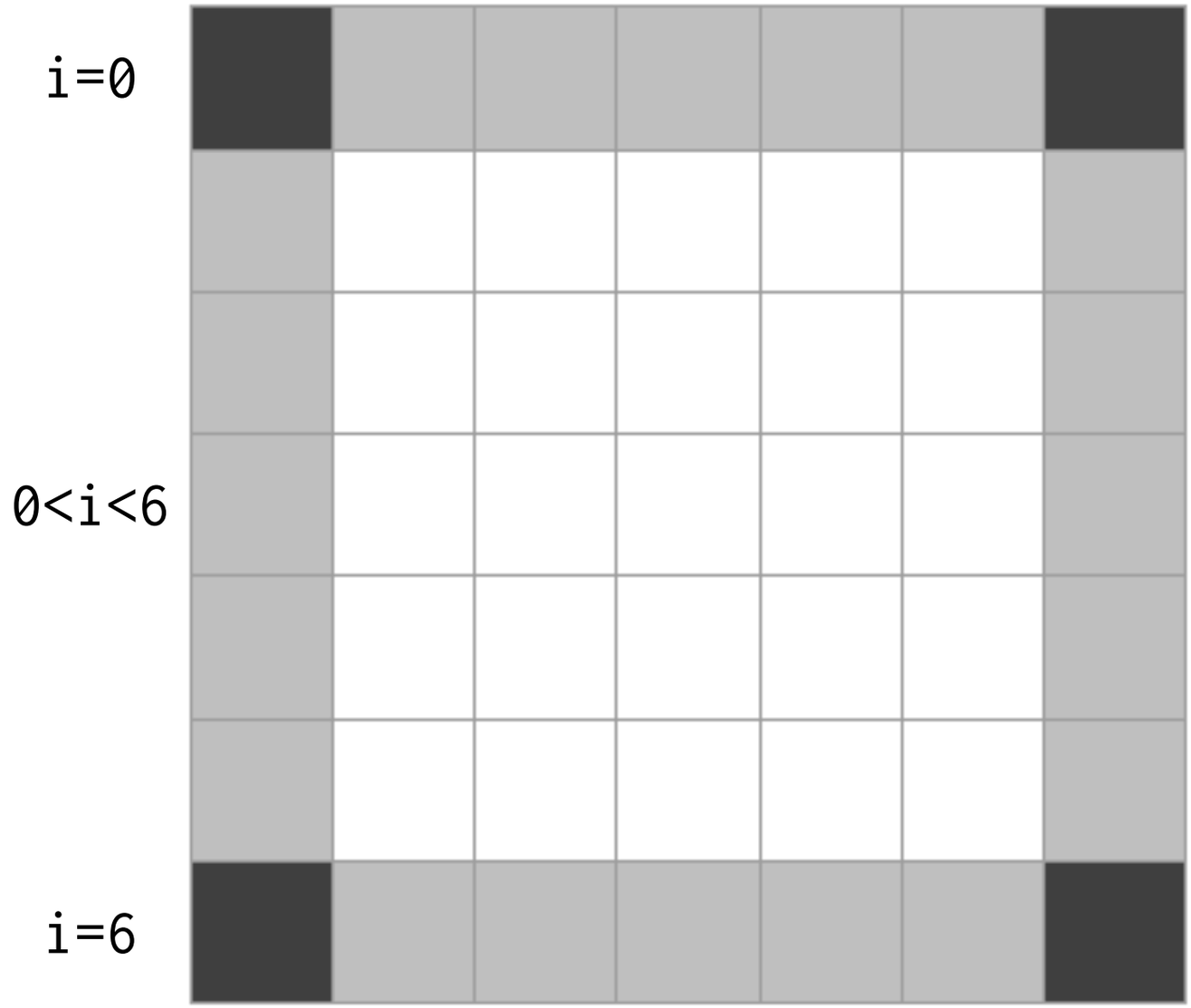


2次元の場合の単純な周期境界条件はトーラス状の構造を考えていることになる

固定端は2次元でも平面のままなので省略.

境界条件 ２次元（2）

n_i=7, n_j=7
の場合



ライフゲームの場合，次世代の“状態”はそのセル自身と８近傍（Moore近傍）に依存する.
そのため，境界条件の処理は上図のように８つ考えればよい.
（結果的に処理は境界条件処理８つ＋メイン１つの計９つに分岐する．）

ライフゲーム　作成方針の例（１）：遷移ルールの定義

擬似的なコードを提示しているので，参考にして自分でプログラムを組もう．

```
# 03-01. ライフゲーム遷移ルール
def 遷移ルール(3x3の配列（注目するセルとその８近傍）):
    近傍の“生”（1）のセルの数を計算する（np.sumを使ってみよう）

    近傍の“生”のセルがちょうど２つならば:
        次世代の（中心の）セルの状態は更新なし
    （２ではなく）近傍の“生”のセルがちょうど3つならば:
        次世代の（中心の）セルの状態は“生”
    それ以外の場合（過疎，過密）：
        次世代の（中心の）セルの状態は“死”

    return 次世代の（中心の）セルの状態
```

どうやって場の全体から部分的な3x3の配列を取ってくるか？
→ インデックス，スライスを活用

ここに挙げているのはあくまで一例．自分がやりやすい方法で実装してOK.

ライフゲーム　作成方針の例（2）：実行部分

```
# 03-02. ライフゲームの実行
# 初期条件の設定
場のサイズを設定（縦，横にそれぞれ何個セルが配置されるか？）
何世代目まで計算するか？
場を記録するリストの準備
場の初期値の設定（例．ランダム，特定パタンの配置など）

for 世代:
    # 状態遷移
    次世代の場を記録する配列の用意

    for i in 場のサイズ（縦）:
        for j in 場のサイズ（横）:
            # 境界条件等による分岐
            # 全部で9通り（メイン+境界条件）
            もしi==0ならば:
                もしj==0ならば:
                    次世代の場[i,j] = 遷移ルール(場[[-1,0,1],:][:,[-1,0,1]])
                （j==0でなく）もし0<j<場のサイズ（横）-1ならば:
                    次世代の場[i,j] = 遷移ルール(場[[-1,0,1],:][:,[j-1,j,j+1]])
                （j<場のサイズ（横）-1でなく）もしj==場のサイズ（横）-1ならば:
                    …
            （i==0でなく）もし0<i<場のサイズ（縦）-1ならば:
                …
            （i<場のサイズ（縦）-1でなく）もしi==場のサイズ（縦）-1ならば:
                …

    場の更新（次世代を現世代にコピー，np.copyを使ってみよう）
    場の記録（リストへ追加）
```

周期境界条件を採用した場合

ここに挙げているのはあくまで一例．自分がやりやすい方法で実装してOK.

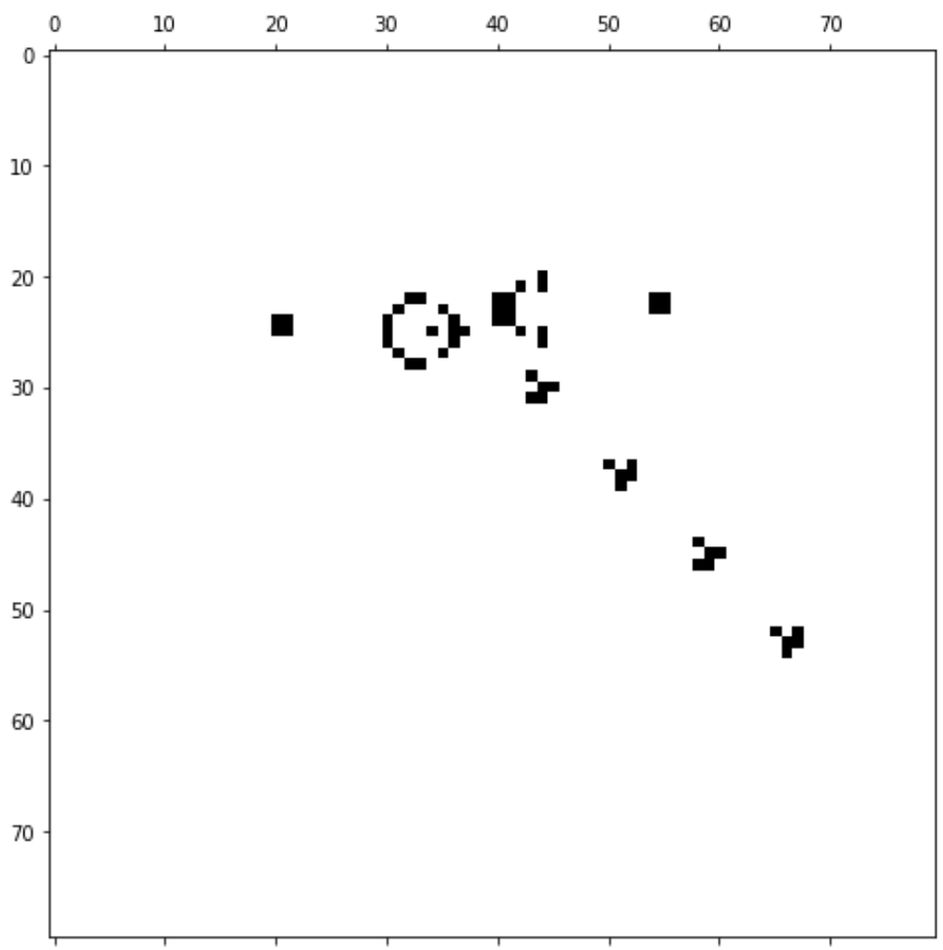
ライフゲーム　作成方針の例（３）：可視化１

ここではfield_listにライフゲームの結果が保存されているとする
field_listは各ステップでの状態を記録した配列のリストとする

特定の世代をプロット

```
# 03-03. セルオートマトンの可視化
plt.figure(figsize=(8, 8))
plt.matshow(field_list[50], fignum=1, cmap="binary")
```

field_list[50]に記録された場の配列をプロット



ライフゲーム　作成方針の例（４）：可視化２

ここではfield_listにライフゲームの結果が保存されているとする
field_listは各ステップでの状態を記録した配列のリストとする

アニメーションの作成

```
# 03-04. セルオートマトンの可視化：アニメーション
from matplotlib import animation, rc
from IPython.display import HTML

fig, ax = plt.subplots(dpi=150)
plt.close()

artists = []

for i in range(len(field_list)):
    im = ax.matshow(field_list[i], interpolation="none", cmap="binary")

    ax.set_aspect("equal")
    artists.append([im])

# アニメーションへの変換
ani = animation.ArtistAnimation(fig, artists, interval=100)

# アニメーションの表示（JavaScriptとして埋め込み）
rc("animation", html="jshtml")
ani
```

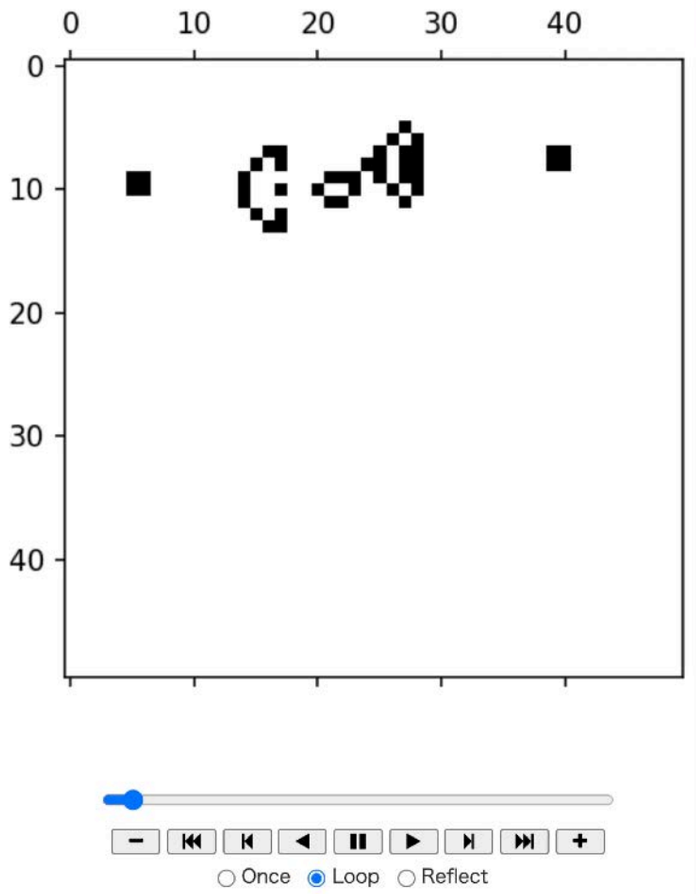
アニメーション作成に必要な
モジュールのimport

各フレームを格納するリスト

アスペクト比の設定

フレームのリストへの格納

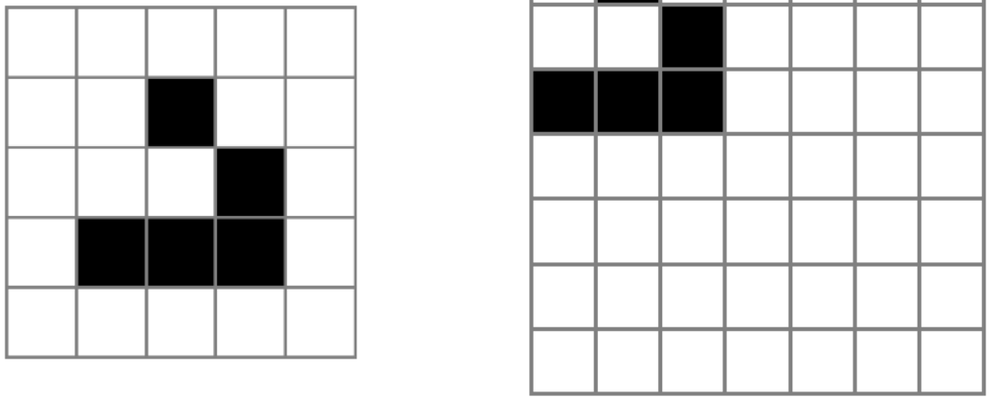
intervalで各フレームが表示される
時間（ms）を指定



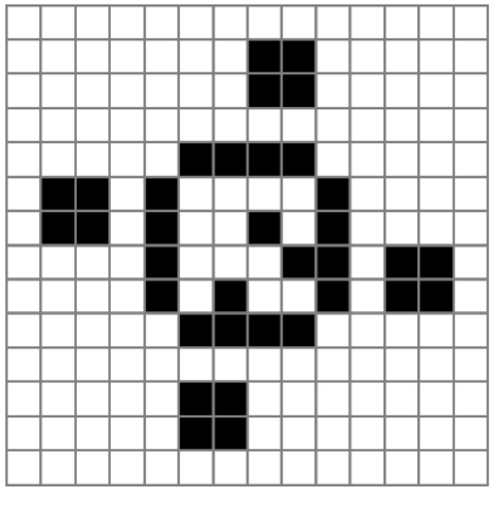
ライフゲーム：パターンの例（１）

```python
# グライダー
ptn_glider = np.array([[1, 1, 1], [1, 0, 0], [0, 1, 0]], dtype=int)

# 回転花火
ptn_pinwheel = np.array(
    [
        [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
        [1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
        [1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1],
        [0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1],
        [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
    ],
    dtype=int,
)
```

グライダー glider



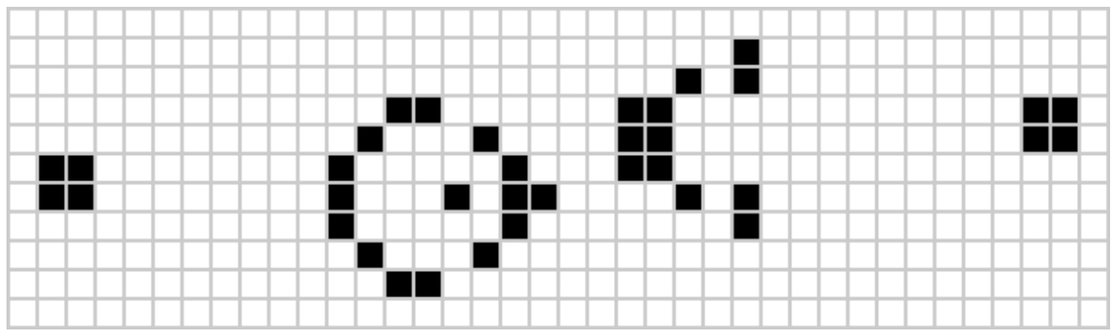
回転花火 pinwheel



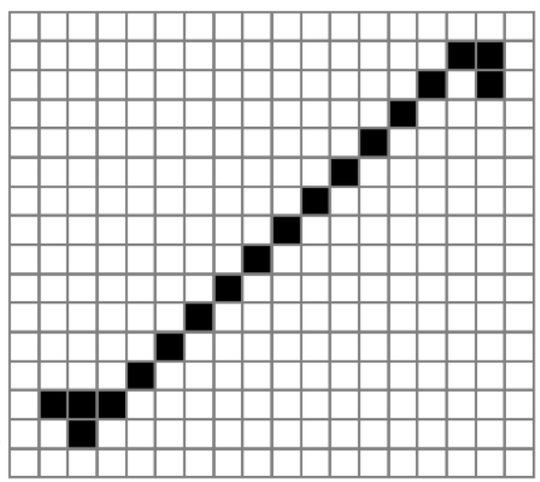
ライフゲーム：パターンの例（2）

```
# グライダー銃
ptn_glidergun = np.array([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
], dtype=int)

# パン屋
ptn_baker = np.array([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
], dtype=int)
```

Gosperグライダー銃



パン屋 baker



もっといろいろなパタンを知りたい人はLifeWiki http://www.conwaylife.com/wikiを見てみよう

本日の課題

ノーマル：１つ選ぶ

ハード：２つ以上

1. １次元セル・オートマトンのクラス1, 2, 3, 4をそれぞれに相当する遷移ルールを見つけ出し，示せ．また，どういった時に各クラスが出現するか考察せよ.
2. ライフゲームで固定物体，振動子，移動物体，繁殖のパタンをそれぞれ探しだし，その遷移課程を示せ.
3. 1次元セル・オートマトン，ライフゲームのいずれかについて，どういった生物学的解釈が出来るか？自分なりに考察せよ.
4. 質問，意見，要望等をどうぞ.

課題をノートブック（.ipynbファイル）にまとめて，Moodleにて提出すること

ファイル名は[回数，01~15]_[難易度，ノーマル nかハード h].ipynb．例．10_nh.ipynb

次回予告

第10回：数理生物学は役に立つのか？（２）：
時間生物学における数理モデリング

6月24日

復習推奨

- 微分方程式の数値解法