鑑賞するフラクタル

吉永 塁 *

2023 年 2 月 1 日

1　はじめに

問題を解いたり証明を考えたりするだけでなく，「鑑賞」することも，数学との向き合い方の一つとしてあっても良いのではないかと思います．本稿では，題材としてフラクタルとセルオートマトンを取り上げ，いくつかの図形をプログラムで描くことを目的とします．理論的な厳密さは不十分かもしれませんが，自分が書いたプログラムによって図形が実際に描かれていく楽しさを感じてもらえればと思います．ソースコードは付録として掲載しました．実際に動かしてみたり，改良したりして遊んでいただければ幸いです．

1.1　Processing について

今回，図形を描くのに Processing[1] [*1] を用います．Processing は，Java をベースとしたグラフィクスに特化したプログラミング言語です[*2] ．図形描画関数などが予め定義されていて，短い記述で簡単に図形や絵を描くことができるのが特徴です．煩雑なことをせずに視覚的な出力が得られるので，プログラミングに馴染のない方でも親しみやすいのではないかと思います．本稿のプログラムは全て Processing で記述されています．

プログラミングの最低限の基礎知識は前提としていますが，それほど高度なことをする訳ではないので，変数・配列・関数・条件分岐（`if` 文）・繰り返し（`for` 文）あたりの内容を知っていれば問題なく読み進められると思います．その他，Processing の構文や関数などについて，詳しくは公式 Web ページのリファレンス等を適宜参照してください．p5.js 全般についての文献として，[4, 5] 等が参考になるかと思います．その他，[6] では，数学的な概念の可視化を題材としています．

2　フラクタル・再帰

フラクタル図形とは，図形の一部が全体の相似形となっているような図形のことをいいます[*3]．また，ある関数の中で，その関数自身を呼び出すような構造を持つ関数のことを再帰関数と呼びます．この章では，フラクタル図形を描くとき中心となる再帰関数を示し，フラクタルと再帰という 2 つの概念が対応していることを見ていきます．

2.1　Sierpinski のカーペット

Sierpinski のカーペット[*4] とは，図 1 のような，正方形の中心を取り除くという操作[*5] を繰り返すことで得られる[*6] フラクタル図形です．

* 情報工学専攻 2 年 m1622047@edu.kit.ac.jp

*1 掲載するプログラムは，Processing 4.0b3 で作成したものです．

*2 Java ベースが基本ですが，Python ベースの Processing.py[2] や，JavaScript ベースでブラウザ上で実行可能な p5.js[3] などの派生もあります．

*3 フラクタル図形について，詳しくは [7, 8, 9, 10] などを参照してください．

*4 ポーランドの数学者 Wacław Sierpiński（1882-1969）に因みます．

*5 ここでは「取り除く」という操作を，「黒色で塗る」という操作として表現しています．

*6 厳密には，この「中心を取り除く」の様な操作を無限回繰り返すことで得られる図形のことを，フラクタル図形と呼びます．今回の目的は，フラクタル図形を視覚的に確認することなので，図形の変化が判別できなくなる程度の有限回の繰り返しで近似します．

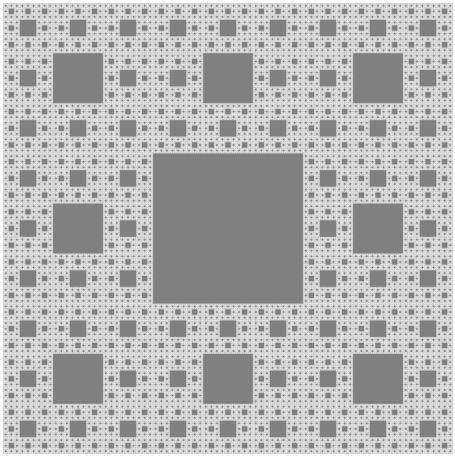
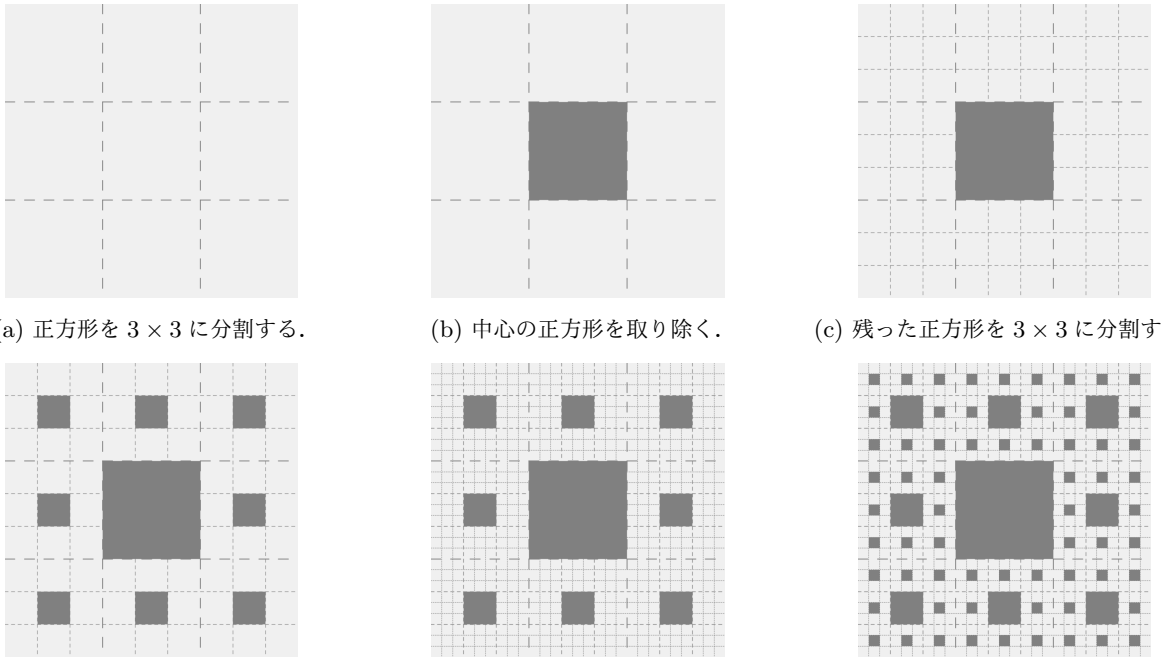


図 1: Sierpinski のカーペット

Sierpinski のカーペットは，次の手順 1-3 を繰り返し実行することによって得られます．

手順 1： 与えられた正方形を 3×3 の小正方形に分割する．
手順 2： 分割してできた 9 個の小正方形のうち，中心にあるものを取り除く．
手順 3： 残りの 8 個の小正方形を手順 1 の正方形とみる．



(a) 正方形を 3×3 に分割する．　(b) 中心の正方形を取り除く．　(c) 残った正方形を 3×3 に分割する．

(d) 各正方形の中心を取り除く．　(e) 残った正方形を 3×3 に分割する．　(f) 各正方形の中心を取り除く．

図 2: Sierpinski のカーペットを描く過程

この操作を再帰関数として記述するとソースコード 1 のようになります[*7]．手順 2 の正方形の中心を取り除く操作が，正方形を描く関数 `square`（8 行目）として表されていること，また，手順 3 の残った 8 つの正方形をそれぞれひとつの正方形として見る操作が，関数 `carpet` の再帰呼び出し（9-16 行目）として表されています．プログラム全体は A.1 節に記載しました．出力は図 1 のようになります．

[*7] p5.js では原点が画面左上で y 軸が下向き正として取られていることに注意してください．

ソースコード 1: Sierpinski のカーペットを描く再帰関数

```
1  void carpet(float x, float y, float l, int n){  /* 左上座標(x, y), 辺lの正方形；再帰の深さn */
2      if( n <= 0 ){
3          return ;
4      }
5
6      float sqlen = l / 3;  // 小正方形の辺の長さ
7
8      square(x+sqlen, y+sqlen, sqlen);  // 中心
9      carpet(x        , y        , sqlen, n-1);  // 上左
10     carpet(x+sqlen  , y        , sqlen, n-1);  // 上中
11     carpet(x+2*sqlen, y        , sqlen, n-1);  // 上右
12     carpet(x        , y+sqlen  , sqlen, n-1);  // 中左
13     carpet(x+2*sqlen, y+sqlen  , sqlen, n-1);  // 中右
14     carpet(x        , y+2*sqlen, sqlen, n-1);  // 下左
15     carpet(x+sqlen  , y+2*sqlen, sqlen, n-1);  // 下中
16     carpet(x+2*sqlen, y+2*sqlen, sqlen, n-1);  // 下右
17 }
```

Sierpinski のカーペットは，正方形の一部を取り除くという操作により得られる 2 次元的な図形[*8] でした．これに対し，線分について同様の操作を行うことで得られる 1 次元的な図形を，Cantor 集合（図 13），立方体について同様の操作を行うことで得られる 3 次元的な図形を，Menger のスポンジ（図 14）と呼びます．

問 2.1. Sierpinski のカーペットを描くプログラム（ソースコード 13）を実行して，図 1 が出力されることを確認してください．また，再帰呼び出しの深さ（ソースコード 13 の 11 行目，関数 `carpet` の第 4 引数の値）を変更すると，出力される図はどのように変化するか確かめてください．

問 2.2. Sierpinski のカーペットの再帰関数 `carpet` の，再帰呼び出し（ソースコード 13 の 22-29 行目）のいくつかの行を適当にコメントアウトして，出力される図がどのように変化するか確かめてください．

問 2.3. 問 2.2 を参考に，Vicsek フラクタル（図 15, 図 16）を描くプログラムを作成してください．ヒント：Sierpinski のカーペットを描くプログラム（ソースコード 13）の，中心を塗りつぶす関数 `square`（21 行目）を，`if` 文の中（15-16 行目の間）へ移動して，代わりに中心部分の再帰呼び出し `carpet` を加える．

問 2.4. Cantor 集合（図 13）を描くプログラムを作成してください．

[*8] ここでは，2 次元図形である正方形から始めるという意味で，Sierpinski のカーペットを「2 次元的」と表現しています．

2.2 Sierpinski のギャスケット

Sierpinski のギャスケット[*9] とは，図 3 のような，三角形の中心を取り除くという操作を繰り返すことで得られるフラクタル図形です．2.1 節でみた Sierpinski のカーペットの三角形版です．

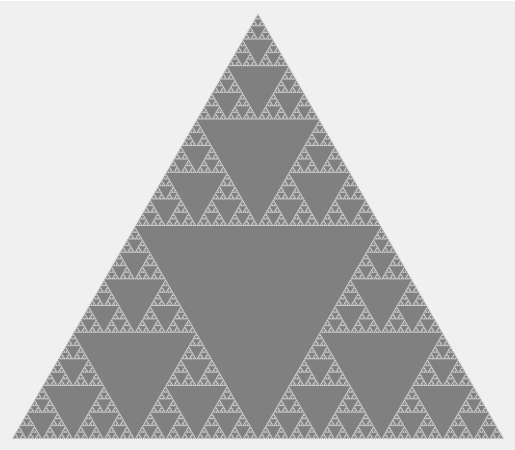


図 3: Sierpinski のギャスケット

Sierpinski のギャスケットは，Sierpinski のカーペットとほぼ同様の手順で得られます．Sierpinski のカーペットを得る手順において，正方形を正三角形と読み替えるだけです．Sierpinski のギャスケットを描く再帰関数はソースコード 2 のようになります．Sierpinski のカーペットの場合と同様に，正三角形の中心を取り除く操作が，三角形を描く関数 `triangle`（14 行目）と対応し，残り 3 つの正三角形をそれぞれひとつの正三角形として見る操作は，関数 `gasket` の再帰呼び出し（15-18 行目）と対応します．プログラム全体は A.2 節に記載しました．出力は図 3 のようになります．

ソースコード 2: Sierpinski のギャスケットを描く再帰関数

```
1  void gasket(float tx, float ty, float lx, float ly, float rx, float ry, int n) {
2      /* 上頂点(tx, ty), 左下頂点(lx, ly), 右下頂点(rx, ry)の正三角形；再帰の深さn */
3      if ( n <= 0 ) {
4          return ;
5      }
6
7      float lcx = (tx + lx) / 2;  // 上頂点と左下頂点の中点のx座標
8      float lcy = (ty + ly) / 2;  // 上頂点と左下頂点の中点のy座標
9      float rcx = (tx + rx) / 2;  // 上頂点と右下頂点の中点のx座標
10     float rcy = (ty + ry) / 2;  // 上頂点と右下頂点の中点のy座標
11     float bcx = (lx + rx) / 2;  // 左下頂点と右下頂点の中点のx座標
12     float bcy = (ly + ry) / 2;  // 左下頂点と右下頂点の中点のy座標
13
14     triangle(lcx, lcy, rcx, rcy, bcx, bcy);  // 中心
15     gasket(tx , ty , lcx, lcy, rcx, rcy, n-1);  // 上
16     gasket(lcx, lcy, lx , ly , bcx, bcy, n-1);  // 左下
17     gasket(rcx, rcy, bcx, bcy, rx , ry , n-1);  // 右下
18 }
```

問 2.5. 正多角形を描く関数（ソースコード 3）を参考に，Pentaflake（図 17），Hexaflake（図 18）を描くプログラムを作成してください．ヒント：再帰の深さが n のときの $\{5 \mid 6\}$ 角形と，$n-1$ のときの $\{5 \mid 6\}$ 角形の大きさの比率を考える．

[*9] Sierpinski の三角形とも呼ばれます．

ソースコード 3: 正多角形を描く関数

```
1 void polygon(float x, float y, float l, int n) {  /* 中心(x, y)から各頂点までの長さがlの正n角形 */
2     beginShape();
3     for ( float t = 0; t < TWO_PI; t += TWO_PI/n ) {
4         vertex(x+l*cos(t), y+l*sin(t));
5     }
6     endShape(CLOSE);
7 }
```

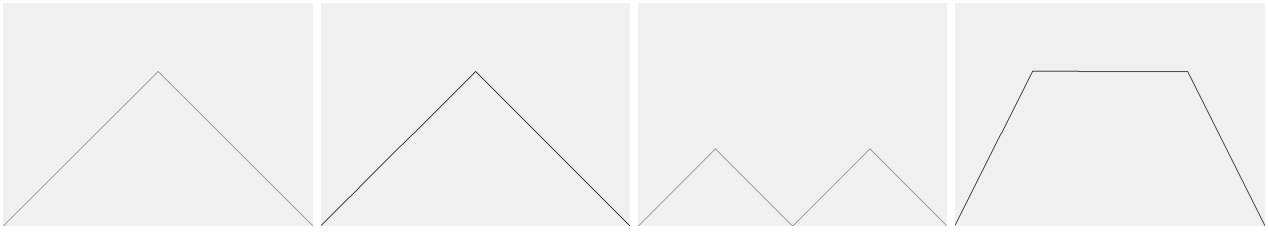
2.3 高木曲線

高木曲線[*10] とは，図 4 のような，三角波の足し合わせによって得られるフラクタル図形です．

図 4: 高木曲線

高木曲線は，次の手順 1-3 を繰り返し実行することによって得られます．

手順 1： 与えられた区間の中点を頂点とする三角波を考える．
手順 2： 手順 1 の三角波を，これまでに得た全ての三角波を足し合わせたものに加える。
手順 3： 区間を中心で 2 分割し，左右の小区間を手順 1 の区間とみる．

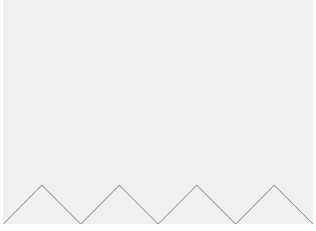


(a) 区間の中点を頂点とする三角波を考える．

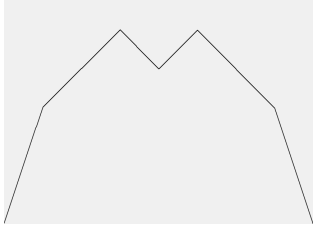
(b) (a) の三角波を初期状態（高さ 0 の波）に足し合わせる．

(c) (a) の区間を 2 等分してそれぞれ三角波を考える．

(d) (c) の三角波を (b) に足し合わせる．

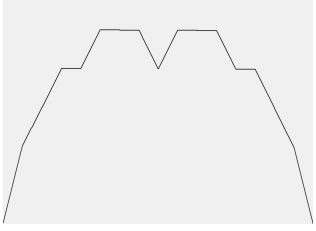


(e) (c) の各区間を 2 等分してそれぞれ三角波を考える．



(f) (e) の三角波を (d) に足し合わせる．

(g) (e) の各区間を 2 等分してそれぞれ三角波を考える．



(h) (g) の三角波を (f) に足し合わせる．

図 5: 高木曲線を描く過程

[*10] 日本の数学者高木貞治（1875-1960）に因みます．特徴的な形からブラマンジェ曲線とも呼ばれます．

この操作を再帰関数として記述するとソースコード 4 のようになります．ys は，これまでに得た全ての三角波を足し合わせた曲線の，位置 x における高さを保持する配列です．手順 1-2 の三角波の足し合わせが，10-15 行目の for 文の処理と対応し，手順 3 の左右の小区間をそれぞれひとつの区間として見る操作は，関数 takagicurve の再帰呼び出し（16-17 行目）と対応します．プログラム全体は A.3 節に記載しました．出力は図 4 のようになります．

ソースコード 4: 高木曲線を描く再帰関数

```
1  void takagicurve(int lx, int rx, int n) {  /* 区間[lx, rx]；再帰の深さn */
2      if( n <= 0 ){
3          return ;
4      }
5
6      int cx = (lx + rx) / 2;  // 区間の中点
7      int y = 0;               // 三角波の高さ
8
9      // 三角波の足し合わせ（ysは位置xにおける高さを保持する配列）
10     for ( int x = lx; x < cx; x++ ) {
11         ys[x] += y++;
12     }
13     for ( int x = cx; x < rx; x++ ) {
14         ys[x] += y--;
15     }
16     takagicurve(lx, cx, n-1);  // 左
17     takagicurve(cx, rx, n-1);  // 右
18 }
```

高木曲線は，連続且つ至る所で微分不可能であるという性質を持ちます．

問 2.6. 高木曲線の他にも，連続且つ至る所で微分不可能という性質を持つ関数として，Weierstrass 関数があります．Weierstrass 関数について調べてみてください。

2.4 Koch 曲線

Koch 曲線[*11] とは，図 6 のような，線分を 3 分割し，その中央の部分を山形にするという操作を繰り返すことで得られるフラクタル図形です．

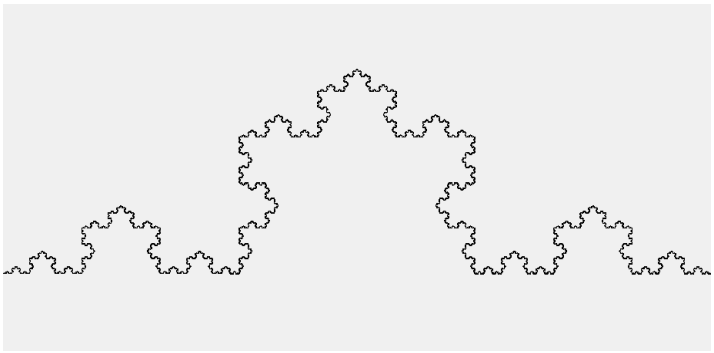
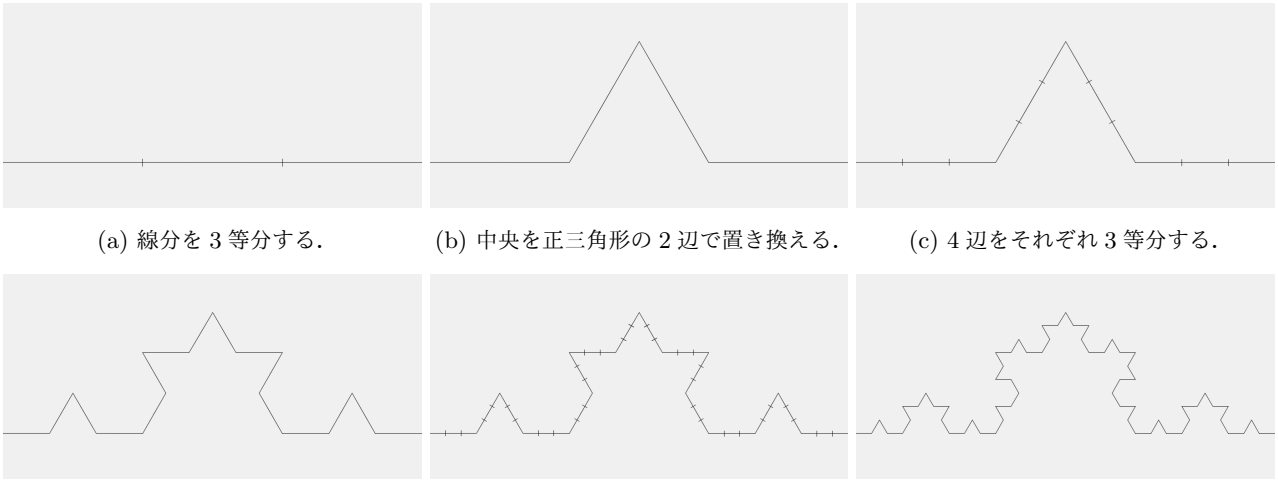


図 6: Koch 曲線

Koch 曲線は，次の手順 1-3 を繰り返し実行することによって得られます．

手順 1： 与えられた線分を 3 つの小線分に 3 等分する．
手順 2： 中央の小線分を，小線分を 1 辺とするような正三角形の残りの 2 辺と置き換える．
手順 3： 4 つの小線分を手順 1 の線分とみる．

[*11] スウェーデンの数学者 Helge von Koch（1870-1924）に因みます．



(a) 線分を 3 等分する．　(b) 中央を正三角形の 2 辺で置き換える．　(c) 4 辺をそれぞれ 3 等分する．

(d) 中央を正三角形の 2 辺で置き換える．　(e) 各辺をそれぞれ 3 等分する．　(f) 中央を正三角形の 2 辺で置き換える．

図 7: Koch 曲線を描く過程

この操作を再帰関数として記述するとソースコード 5 のようになります．手順 3 の 4 つの線分をそれぞれひとつの線分として見る操作が，関数 kochcurve の再帰呼び出し（13-16 行目）と対応します．2-3 つ目の再帰呼び出し（14-15 行目）が，手順 2 の正三角形の 2 辺と対応していることに注意してください．プログラム全体は A.4 節に記載しました．出力は図 6 のようになります．

ソースコード 5: Koch 曲線を描く再帰関数

```
1  void kochcurve(float lx, float ly, float rx, float ry, int n) {  /* 左端(lx, ly), 右端(rx, ry)の線分 ;
2                                                                     再帰の深さn */
3      if ( n <= 0 ) {
4          line(lx, ly, rx, ry);
5          return ;
6      }
7
8      float dx = (rx - lx) / 3;  // 小線分x成分
9      float dy = (ry - ly) / 3;  // 小線分y成分
10     float cx = (lx + rx) / 2;  // 線分の中点のx座標
11     float cy = (ly + ry) / 2;  // 線分の中点のy座標
12
13     kochcurve(lx             , ly             , lx+dx          , ly+dy          , n-1);  // 左
14     kochcurve(lx+dx          , ly+dy          , cx-sqrt(3)/2*dy, cy+sqrt(3)/2*dx, n-1);  // 中左
15     kochcurve(cx-sqrt(3)/2*dy, cy+sqrt(3)/2*dx, rx-dx          , ry-dy          , n-1);  // 中右
16     kochcurve(rx-dx          , ry-dy          , rx             , ry             , n-1);  // 右
17 }
```

Koch 曲線も高木曲線と同様，連続且つ至る所で微分不可能であるという性質を持ちます．また，図 19 の様に，3 つの Koch 曲線を星形のように繋げた図形を Koch 雪片といいます．

問 2.7. Koch 曲線の長さは無限大であることを示してください．

問 2.8. 最初の正三角形の 1 辺の長さを l として，Koch 雪片の面積を求めてください．

2.5　再帰的定義

この章では，再帰関数によってフラクタル図形を描いてきました．再帰関数（再帰的定義）は，数学やプログラミングにおいて，フラクタル以外の場面でも頻繁に登場します．身近な例では，総和演算子 Σ は，次のように再帰的に

定められます．

$$\sum_{i=0}^{n} x_i \coloneqq \begin{cases} x_0 & (n=0) \\ x_n + \displaystyle\sum_{i=0}^{n-1} x_i & (n \geq 1) \end{cases}$$

これを p5.js の再帰関数として書くと，ソースコード 6 のようになります．プログラム全体は A.5 節に記載しました．

ソースコード 6: 総和を計算する再帰関数

```
/* 配列xsの要素の総和を求める関数 */
int recursive_sum(int[] xs, int n){  /* 配列xs={x0,x1,...,xn}；再帰の深さn */
    if( n == 0 ){
        return xs[n];
    }
    return xs[n] + recursive_sum(xs, n-1);
}
```

問 2.9. 総乗演算子 Π，階乗演算子 ! も再帰的に定義されます．配列 `xs` の総和を計算する再帰関数（ソースコード 6）を参考に，配列 `xs` の総乗を計算する再帰関数 `int recursive_prod(int[] xs, int n)`，n の階乗を計算する再帰関数 `int recursive_factorial(int n)` を作成してください．

問 2.10. 次のように定義される数列 a_n を Fibonacci 数列と言います．

$$a_n \coloneqq \begin{cases} 1 & (n=0,1) \\ a_{n-1} + a_{n-2} & (n \geq 2) \end{cases}$$

Fibonacci 数列の第 n 項を計算する再帰関数 `int recursive_fib(int n)` を作成してください．

問 2.11. $n \in \mathbb{N}$, $0 \leq k \leq n$ に対して，二項係数 $\binom{n}{k}$ [*12] は次のように定義されます．

$$\binom{n}{k} \coloneqq \begin{cases} 1 & (k=0,n) \\ \binom{n-1}{k} + \binom{n-1}{k-1} & (k \neq 0,n) \end{cases}$$

二項係数 $\binom{n}{k}$ を計算する再帰関数 `int recursive_binom(int n, int k)` を作成してください．

問 2.12. 図 8 のように，二項係数をピラミッド状に並べたものを，Pascal の三角形[*13] と言います．Pascal の三角形を，偶数・奇数で色を塗り分けると，どのような模様が現れるでしょうか．

$$\begin{array}{c} \binom{0}{0} \\ \binom{1}{0}\ \binom{1}{1} \\ \binom{2}{0}\ \binom{2}{1}\ \binom{2}{2} \\ \binom{3}{0}\ \binom{3}{1}\ \binom{3}{2}\ \binom{3}{3} \\ \binom{4}{0}\ \binom{4}{1}\ \binom{4}{2}\ \binom{4}{3}\ \binom{4}{4} \\ \vdots \end{array} = \begin{array}{c} 1 \\ 1\quad 1 \\ 1\quad 2\quad 1 \\ 1\quad 3\quad 3\quad 1 \\ 1\quad 4\quad 6\quad 4\quad 1 \\ \vdots \end{array}$$

図 8: Pascal の三角形

[*12] ${}_nC_k$ と表記される場合もあります．
[*13] Pascal の三角形や二項係数について，詳しくは [11] などを参照して下さい．

3　セルオートマトン

セルオートマトンとは，空間内に固定配置された多数のセルが，局所的に作用し合うことで状態を変化させていくようなシステムのことをいいます．セルオートマトンの例として，John Horton Conway（1937-2020）が考案したライフゲームが有名です．

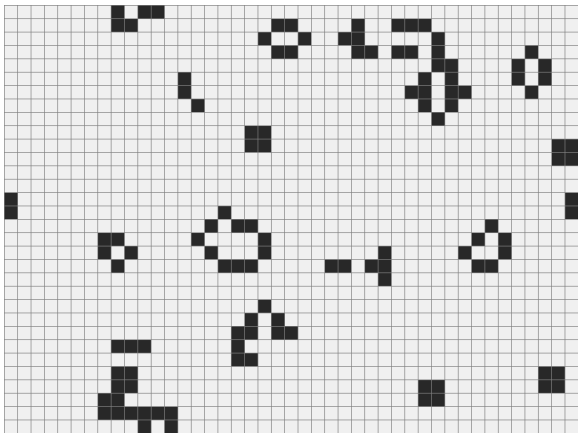


図 9: ライフゲーム：黒いマスが生きたセル，白いマスが死んだセルを表す．周囲のセルの状態によってセルの生存・死滅が決定する．周期的に振動するものや移動するものなど様々なパターンが現れる．

セルが直線状（1 次元上）に配置されたセルオートマトンを，1 次元セルオートマトンと呼びます[*14]．また，1 次元のセルオートマトンでセルの状態が 2 値 $\{0,1\}$ のいずれかをとり，セルの次の状態が自身と隣接する 2 つのセルの状態によって決定されるものを，基本セルオートマトンと呼びます[*15]．この章では，基本セルオートマトンについて見ていきます．

3.1　状態遷移と Wolfram コード

基本セルオートマトンでは，あるセルの時刻 $t+1$ の状態は，そのセルと隣接する 2 つ（左隣・右隣）のセルの時刻 t の状態によって決定されます．

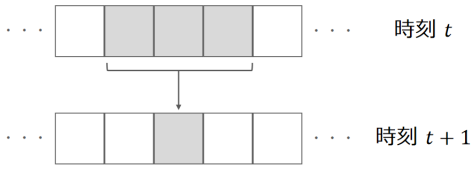


図 10: 基本セルオートマトンの遷移：時刻 $t+1$ のセル（下列灰色のセル）の状態は，時刻 t におけるそのセル自身の状態と隣接する 2 つのセル（上列灰色のセル）の状態によって決定される．

3 つのセルの現在の状態から，セルの次の状態を決定するための遷移規則について考えましょう．今，セルの状態が 2 通りの基本セルオートマトンを考えているので，遷移前の 3 つのセルの状態の組み合わせは全部で 2^3 通り存在します．また，遷移後のセルの状態は，$\{0,1\}$ の 2 通りです．遷移前の状態が 8 通り，遷移後の状態が 2 通りあるので，遷移の規則は全部で 2^8 通り存在します．

この 256 通りの遷移規則を区別するため，それぞれの遷移規則に番号を割り当てます．遷移後のセルの状態を順に並べ，その数字列を 2 進数として見たときの数を，それぞれの規則に割り当てます．表 1 の遷移規則を例として説明

[*14] セルが平面上に配置されるライフゲームは 2 次元セルオートマトンです．図 9 は，2 次元のセルオートマトンの 1 時刻の状態を表した図です．1 次元セルオートマトンの 1 時刻の状態を上から順に並べて遷移を可視化した図 12 との違いに注意してください．

[*15] 特徴を列挙して，1 次元 2 状態 3 近傍セルオートマトンと呼ぶこともあります．

します．

表 1: 遷移規則（ルール 150）

自身及び隣接するセルの状態	111	110	101	100	011	010	001	000
遷移後のセルの状態	1	0	0	1	0	1	1	0

この遷移規則における遷移後の状態（表 1 下行）を，左から順に並べると $(1,0,0,1,0,1,1,0)$ となっています．これを 2 進数とみると $10010110_{(2)} = 1\times 2^7 + 1\times 2^4 + 1\times 2^2 + 1\times 2^1 = 150$ となるので，表 1 の遷移規則をルール 150 と呼びます．この他の遷移規則についても同様に番号を割り当てます．このようにして割り当てられた遷移規則の番号を Wolfram コードと呼びます[*16]．

例 3.1. 表 2 のような遷移規則は，ルール 90 と呼ばれます．

表 2: 遷移規則（ルール 90）

自身及びと隣接するセルの状態	111	110	101	100	011	010	001	000
遷移後のセルの状態	0	1	0	1	1	0	1	0

例 3.2. 現在のセルの状態が"01011100"であるような基本セルオートマトンの状態遷移を考えます．遷移規則はルール 150 として，両端は繋がっているとします．

表 3: ルール 150 に従う遷移計算の例

現在のセルの状態	0	1	0	1	1	1	0	0
自身及び隣接するセルの状態	001	010	101	011	111	110	100	000
遷移後のセルの状態	1	1	0	0	1	0	1	0

各セルの状態遷移は表 3 のように計算されます．遷移後のセルの状態は"11001010"となります．

最後に，遷移規則をプログラムで表すことを考えます．ここではルール 150 に従った状態遷移，つまり表 1 を関数として実装します．関数はソースコード 7 のように書けます．

ソースコード 7: ルール 150 に基づく遷移を計算する関数

```
1 int updateState(int l, int t, int r) {  /* ルール150に従い，3つのセルの状態(l, t, r)から次の状態を計算 */
2     int binnum = 4*l + 2*t + 1*r;              // 3つのセルの状態の列を2進数とみたときの数 [0,1,..,7]
3     int[] rule150 = {0, 1, 1, 0, 1, 0, 0, 1};  // ルール150における遷移後の状態を持つ配列
4     return rule150[binnum];
5 }
```

基本セルオートマトンでは，状態は $\{0,1\}$ の 2 通りなので，遷移前の 3 つのセルの状態の並びを 2 進数として見ると，0 から 7 の自然数となります（2 行目の変数 `binom`）．また，遷移後の状態を，配列 `rule150` の `binom` 番目に配置しています（3 行目）[*17]．例えば，遷移前の状態が `(l,t,r)=(1,0,1)` の場合，`binom=5` となり，配列 `rule150` の `binom=5` 番目の値を参照すれば，遷移後の状態 `rule150[5]=0` が得られます．

3.2 実装

ここからは，いくつかの関数を実装して，基本セルオートマトンのプログラムを完成させます．以降の説明において，n 回状態遷移を行ったセルを第 n 世代のセルと呼ぶことにします．

[*16] この名前は Stephen Wolfram に因みます．Stephen Wolfram はセルオートマトンの解析の他に，Mathmatica や Wolfram Alpha[12] の開発でも有名です．

[*17] 配列は 0 番目から順に格納されるので，表 1 下行とは逆順で数が格納されることに注意してください．

3.2.1 大域変数

定義する大域変数をソースコード 8 に示します．セルの描画サイズを SQSIZE として与えています．これはセルの描画の他に，セルの個数（配列 cells の要素数）・世代数の決定で使われることになります．また，cells はセルの状態を保持する配列です．セルの世代更新はこの配列が持つ値の更新として行います．

ソースコード 8: 大域変数

```
1 final float SQSIZE = 8.0;  // セルの描画サイズ（定数）
2 int[] cells;               // セルの状態を保持する配列
```

3.2.2 セルの初期化

セルの初期化を行う関数をソースコード 9 に示します．この関数では，セルの状態を保持する配列 cells に初期状態（第 0 世代の状態）を与えます．初期状態は，中央のセルのみを 1，それ以外は 0 としています．また，セルの個数は，描画時に画面に入る最大の数としています[*18]．

ソースコード 9: セルの初期化関数

```
1 void initCells() {
2     cells = new int[(int)(width/SQSIZE)];
3     for ( int i = 0; i < cells.length; i++ ) {
4         cells[i] = ( i == cells.length/2 )? 1 : 0;  // 中央のセルのみ1，それ以外は0で初期化
5     }
6 }
```

3.2.3 世代更新

世代更新（状態遷移）を行う関数をソースコード 10 に示します．5-7 行目で遷移前のセルの状態を配列 cpcells に保存（コピー）した後，9-13 行目で配列 cpcells の値と関数 updateState により，セルの状態（配列 cells の値）を更新しています[*19]．またこのとき，両端（cell.length-1 番目のセルと 0 番目のセル）は繋がっているとして，遷移を行います[*20]．

ソースコード 10: 世代更新関数

```
1  void updateCells() {
2      int[] cpcells = new int[cells.length];  // 遷移前のセルの状態を保持する配列
3
4      /* 遷移前のセルの状態の保存 */
5      for ( int i = 0; i < cells.length; i++ ) {
6          cpcells[i] = cells[i];
7      }
8      /* 世代更新（全セルの状態遷移）*/
9      for ( int i = 0; i < cells.length; i++ ) {
10         cells[i] = updateState(cpcells[(i-1+cells.length)%cells.length],
11                                cpcells[i],
12                                cpcells[(i+1)%cells.length]);
13     }
14 }
```

[*18] Processing において，変数 width はウィンドウ（図形が表示される場所）の幅を保持するシステム変数です．また，対になる変数 height はウィンドウの高さを保持します．いま，セルの描画サイズは SQSIZE で与えられているので，ウィンドウ（幅）に入る最大のセル個数は int(width/SQSIZE) で得られます．

[*19] この関数では，遷移前の状態を保持しておかないと世代更新は正しく行われません．状態遷移は添字が小さいほうから順に行われていくので，例えば，i = 1 の場合，cells[1] = updateState(cells[0], cells[1], cells[2]); としてしまうと，cells[0] は遷移後の状態を持ち，cells[1] の遷移が正しく行われなくなります．

[*20] 添字の剰余計算を利用します．i = 0 の場合に添字が負数になることを防ぐために，左隣のセルの添字のみ cells.length を加算してから計算しています．

3.2.4　セルの描画

セルの描画を行う関数をソースコード 11 に示します．この関数は，世代数 gen を引数に取り，ウィンドウの上から gen 行目[*21] に配列 cells が持つセルの状態を描画します．このとき，各セルについて，状態が 1 であれば黒，そうでなければ（0 であれば）白でセルを描画します．

ソースコード 11: セル描画関数

```
1 void drawCells(int gen) {
2     for ( int i = 0; i < cells.length; i++ ) {
3         fill((cells[i] == 1)? 0 : 255);        // セルの状態が1ならば黒，そうでなければ白
4         square(i*SQSIZE, gen*SQSIZE, SQSIZE);  // セルの描画
5     }
6 }
```

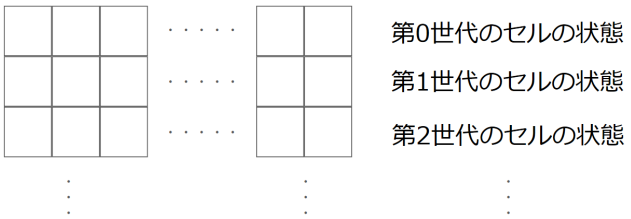


図 11: セルの描画：画面上から n 行目に第 n 世代のセルの状態を描画する．

3.2.5　setup・draw 関数

最後に，ここまで実装してきた関数を組み立てプログラムを完成させます．setup 関数と draw 関数をソースコード 12 に示します．setup 関数ではプログラム全体の設定を行います．draw 関数では，始めにセルの初期化を行い（8 行目），その後，セルの描画と世代更新を第 0 世代から最大の世代まで繰り返します（9-12 行）．また，最大の世代は，ウィンドウに入る最大数（height/SQSIZE）に設定しています[*22]．

ソースコード 12: draw 関数

```
1  void setup(){
2      size(800, 400);  // ウィンドウのサイズ設定：横800x縦400
3      noLoop();        // draw関数を一度だけ実行
4      stroke(128);     // 図形の輪郭線の色の設定
5  }
6
7  void draw() {
8      initCells();  // セルの初期化
9      for ( int i = 0; i < height/SQSIZE; i++ ) {
10         drawCells(i);   // セルの描画
11         updateCells();  // セルの世代更新
12     }
13 }
```

プログラム全体は A.6 節に掲載しました．このプログラムを実行すると図 12 のような出力が得られます．

[*21] このプログラムでは，図 11 のように，セルの状態を世代順に並べたものを描くので，セルの y 座標（4 行目の square 関数の第 2 引数）は，世代数に対応した位置（gen*SQSIZE）にしています．

[*22] ウィンドウ関連については*18 を参照してください．

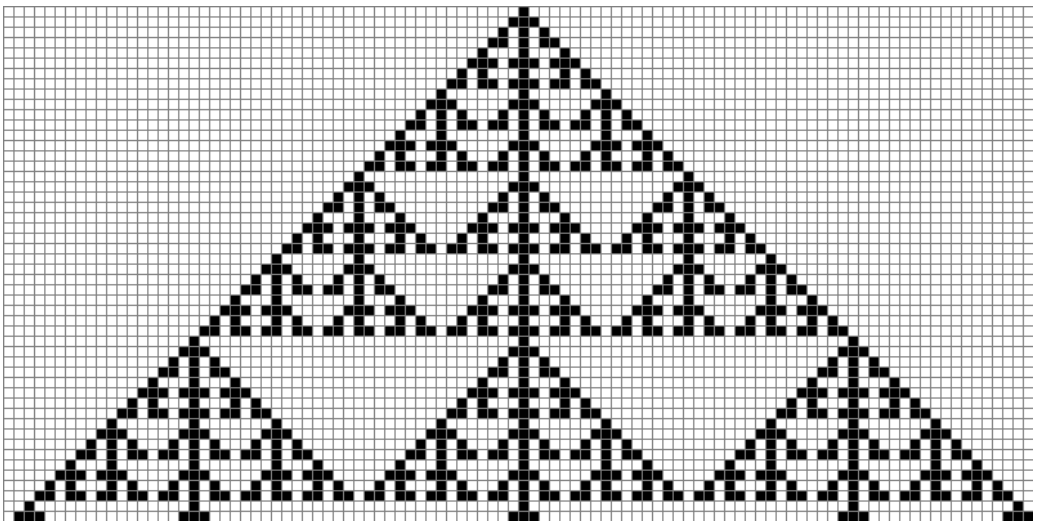


図 12: 基本セルオートマトン（ルール 150）

3.3 余談: Wolfram Alpha

最後に余談として，Wolfram Alpha[12] の紹介をしておきます．Wolfram Alpha は，Wolfram Research が開発した計算知識エンジンで，Web サービスとして公開されています．面倒な計算[*23] やグラフの描画なども行ってくれるので，高性能な電卓として利用できます．計算が合わずに困ったときや，関数のグラフの概形を確かめたいといったときに利用してみると良いかもしれません．

今回取り上げた，基本セルオートマトンにも対応しています．例えば，検索窓に「ルール 150」と入れると，ルール 150 の基本セルオートマトンの情報や基本的な性質について回答が返ってきます．今回作成したプログラムの出力（基本セルオートマトンの世代ごとの遷移の様子）と同様の図まで懇切丁寧に出してくれます．問 3.2 の確認として利用してみてください．

3.4 問

問 3.1. 引数によって遷移規則を変更できるように，ソースコード 7 で示した関数 `updateState` を拡張してください．つまり，3 つのセルの状態と Wolfram コードを引数として取り，セルの次の状態を返す関数 `int updateState(iny l, iny t, iny r, int wcode)` を実装してください．正しく実装ができていれば，第 4 引数 `wcode` を 150 としたときにソースコード 7 で示した関数 `updateState` と同じ挙動を示すはずです．

問 3.2. この章で作成した，ルール 150 に従う基本セルオートマトンのプログラムに，問 3.1 で作成した `updateState` 関数を組み入れて，ルール 150 以外の規則にも対応するように拡張してください．作成したプログラムを実行して，ルール 150 以外の規則ではどのような出力が得られるのか確認してください．特に，ルール 90 ではどのような図形が描かれるでしょうか．

問 3.3. この章で作成した基本セルオートマトンのプログラムでは，セルの初期状態は中央のセルのみ 1，それ以外は 0 としていました．`initCells` 関数の `for` 文の内容を書き換えて，初期状態（$\{0, 1\}$ の値）がランダムに割り当てられるようにしてください．`random` 関数が有用です．

問 3.4. ライフゲームを作成してください．ライフゲームは次のような規則を持ちます．

- セルは 2 次元平面上に配置される．各セルは，生か死のいずれかの状態を持つ．
- あるセルの次の状態は，隣接する 8 つのセルの状態によって決定する．

[*23] 積分計算や代数計算まで計算してくれます．すごい．

- 死んでいるセルは，そのセルに隣接する生きているセルの数が 3 つの場合，次の状態を生とする．隣接する生きているセルの数が 3 つ以外の場合は，セルの状態は死のままとする．
- 生きているセルは，そのセルに隣接する生きているセルが存在しない場合，または隣接する生きているセルの数が 1 つの場合，次の状態を死とする．
- 生きているセルは，そのセルに隣接する生きているセルの数が 2 つまたは 3 つの場合，次の状態を生とする．
- 生きているセルは，そのセルに隣接する生きているセルの数が 4 つ以上の場合，次の状態を死とする．

参考文献

[1] Processing.org `https://processing.org/`

[2] Processing.py `https://py.processing.org/`

[3] p5.js `https://p5js.org/`

[4] マット・ピアソン，久保田晃弘（監修），沖啓介（翻訳）「ジェネラティブ・アート Processing による実践ガイド」ビー・エヌ・エヌ新社 ISBN978-4-86100-963-1 2014 年

[5] 近藤邦雄・田所淳（編）「Processing による CG とメディアアート」講談社 ISBN978-4-06-512974-6 2018 年

[6] 巴山竜来「 数学から創るジェネラティブアート Processing で学ぶ かたちのデザイン」技術評論社 ISBN978-4-297-10463-4 2019 年

[7] ケネス・ファルコナー，服部久美子（訳）「岩波科学ライブラリー 291 フラクタル」岩波書店 ISBN978-4-00-029691-5 2020 年

[8] B. マンデルブロ，広中平祐（監訳）「フラクタル幾何学 上」筑摩書房 ISBN978-4-480-09356-1 2011 年

[9] B. マンデルブロ，広中平祐（監訳）「フラクタル幾何学 下」筑摩書房 ISBN978-4-480-09357-8 2011 年

[10] 山口昌哉「カオスとフラクタル」筑摩書房 ISBN978-4-480-09337-0 2010 年

[11] 吉田武「新装版 オイラーの贈物 人類の秘宝 $e^{i\pi} = -1$ を学ぶ」東海大学出版会 ISBN978-4-486-01863-6 2010 年

[12] Wolfram|Alpha `https://www.wolframalpha.com/`

付録 A　プログラム掲載

本文中に登場したプログラムの全体を掲載します.

A.1　Sierpinski のカーペット

ソースコード 13: Sierpinski のカーペット

```
/* Sierpinski carpet */
void setup() {
    size(800, 800);   // 表示ウィンドウのサイズ設定
    background(240);  // 背景色の設定
    noLoop();         // draw関数を一度だけ実行
    noStroke();       // 図形の輪郭線なし
    fill(128);        // 図形色の設定
}

void draw() {
    carpet(0, 0, width, 7);
}

void carpet(float x, float y, float l, int n) {  /* 左上座標(x, y), 辺 lの正方形；再帰の深さn */
    if( n <= 0 ){
        return ;
    }

    float sqlen = l / 3;  // 小正方形の辺の長さ

    square(x+sqlen, y+sqlen, sqlen);  // 中心
    carpet(x        , y        , sqlen, n-1);  // 上左
    carpet(x+sqlen  , y        , sqlen, n-1);  // 上中
    carpet(x+2*sqlen, y        , sqlen, n-1);  // 上右
    carpet(x        , y+sqlen  , sqlen, n-1);  // 中左
    carpet(x+2*sqlen, y+sqlen  , sqlen, n-1);  // 中右
    carpet(x        , y+2*sqlen, sqlen, n-1);  // 下左
    carpet(x+sqlen  , y+2*sqlen, sqlen, n-1);  // 下中
    carpet(x+2*sqlen, y+2*sqlen, sqlen, n-1);  // 下右
}
```

A.2 Sierpinski のギャスケット

ソースコード 14: Sierpinski のギャスケット

```
/* Sierpinski gasket */
void setup() {
    size(870, 760);   // 表示ウィンドウのサイズ設定
    background(240);  // 背景色の設定
    noLoop();         // draw関数を一度だけ実行
    noStroke();       // 図形の輪郭線なし
    fill(128);        // 図形色の設定
}

void draw() {
    float cx = width / 2;  // 正三角形の中心のx座標
    float cy = 500;        // 正三角形の中心のy座標
    float l = 480;         // 正三角形の中心から各頂点までの長さ
    gasket(cx, cy-l, cx-l*sqrt(3)/2, cy+l/2, cx+l*sqrt(3)/2, cy+l/2, 8);
}

void gasket(float tx, float ty, float lx, float ly, float rx, float ry, int n) {
    /* 上頂点(tx, ty), 左下頂点(lx, ly), 右下頂点(rx, ry)の正三角形；再帰の深さn */
    if ( n <= 0 ) {
        return ;
    }

    float lcx = (tx + lx) / 2;  // 上頂点と左下頂点の中点のx座標
    float lcy = (ty + ly) / 2;  // 上頂点と左下頂点の中点のy座標
    float rcx = (tx + rx) / 2;  // 上頂点と右下頂点の中点のx座標
    float rcy = (ty + ry) / 2;  // 上頂点と右下頂点の中点のy座標
    float bcx = (lx + rx) / 2;  // 左下頂点と右下頂点の中点のx座標
    float bcy = (ly + ry) / 2;  // 左下頂点と右下頂点の中点のy座標

    triangle(lcx, lcy, rcx, rcy, bcx, bcy);  // 中心
    gasket(tx , ty , lcx, lcy, rcx, rcy, n-1);  // 上
    gasket(lcx, lcy, lx , ly , bcx, bcy, n-1);  // 左下
    gasket(rcx, rcy, bcx, bcy, rx , ry , n-1);  // 右下
}
```

A.3　高木曲線

ソースコード 15: 高木曲線

```
/* Takagi curve */
int[] ys;  // 各点の高さを格納する配列

void setup() {
    size(860, 620);     // 表示ウィンドウのサイズ設定
    background(240);    // 背景色の設定
    noLoop();           // draw関数を一度だけ実行
    strokeWeight(1.6);  // 輪郭線の太さの設定
}

void draw() {
    reverseY();                 // y軸を上向きに取る
    initPoints();               // 配列ysの初期化
    takagicurve(0, width, 10);  // 高木曲線
    plot();                     // 曲線のプロット
}

void reverseY() {
    translate(width/2, height/2);
    rotate(PI);
    translate(-width/2, -height/2);
}

void initPoints() {
    ys = new int[width];
    for ( int x = 0; x < ys.length; x++ ) {
        ys[x] = 0;  // 各点の高さを0で初期化
    }
}

void takagicurve(int lx, int rx, int n) {  /* 区間[lx, rx] */
    if( n <= 0 ){
        return ;
    }

    int cx = (lx + rx) / 2;  // 区間の中点
    int y = 0;               // 三角波の高さ

    // 三角波の足し合わせ
    for ( int x = lx; x < cx; x++ ) {
        ys[x] += y++;
    }
    for ( int x = cx; x < rx; x++ ) {
        ys[x] += y--;
    }
    takagicurve(lx, cx, n-1);  // 左
    takagicurve(cx, rx, n-1);  // 右
}

void plot() {
    for ( int x = 0; x < ys.length-1; x++ ) {
        line(x, ys[x], x+1, ys[x+1]);
    }
}
```

A.4 Koch 曲線

ソースコード 16: Koch 曲線

```
/* Koch curve */
void setup() {
    size(920, 450);   // 表示ウィンドウのサイズ設定
    background(240);  // 背景色の設定
    noLoop();         // draw関数を一度だけ実行
}

void draw() {
    reverseY();                        // y軸を上向きに取る
    kochcurve(0, 100, width, 100, 7);  // Koch曲線
}

void reverseY() {
    translate(width/2, height/2);
    rotate(PI);
    translate(-width/2, -height/2);
}

void kochcurve(float lx, float ly, float rx, float ry, int n) {  /* 左端(lx, ly), 右端(rx, ry)の線分；
                                                                    再帰の深さn */
    if ( n <= 0 ) {
        line(lx, ly, rx, ry);
        return ;
    }

    float dx = (rx - lx) / 3;  // 小線分x成分
    float dy = (ry - ly) / 3;  // 小線分y成分
    float cx = (lx + rx) / 2;  // 線分の中点のx座標
    float cy = (ly + ry) / 2;  // 線分の中点のy座標

    kochcurve(lx             , ly             , lx+dx          , ly+dy          , n-1);  // 左
    kochcurve(lx+dx          , ly+dy          , cx-sqrt(3)/2*dy, cy+sqrt(3)/2*dx, n-1);  // 中左
    kochcurve(cx-sqrt(3)/2*dy, cy+sqrt(3)/2*dx, rx-dx          , ry-dy          , n-1);  // 中右
    kochcurve(rx-dx          , ry-dy          , rx             , ry             , n-1);  // 右
}
```

A.5　再帰関数による総和計算

ソースコード 17: 再帰関数による総和計算

```
1  void setup(){
2      int[] xs = {1, 2, 3, 4, 5, 6, 7, 8, 9};
3      int s = recursive_sum(xs, 8);
4      println(s);
5  }
6
7  /* 配列xsの要素の総和を求める関数 */
8  int recursive_sum(int[] xs, int n) {  /* 配列xs=[x0,x1,...,xn]；再帰の深さn */
9      if ( n == 0 ) {
10         return xs[n];
11     }
12     return xs[n] + recursive_sum(xs, n-1);
13 }
```

注：これは総和計算の結果の数値（3 行目で計算される s の値）を出力するだけのプログラムです．ウィンドウには何も表示されないことに注意してください．

A.6　基本セルオートマトン

ソースコード 18: 基本セルオートマトン（ルール 150）

```
/* Elementary Cellular Automaton */
final float SQSIZE = 8.0;  // セルの描画サイズ（定数）
int[] cells;               // セルの状態を保持する配列

void setup() {
    size(800, 400);
    noLoop();
    stroke(128);
}

void draw() {
    initCells();
    for ( int i = 0; i < height/SQSIZE; i++ ) {
        drawCells(i);
        updateCells();
    }
}

void initCells() {
    cells = new int[(int)(width/SQSIZE)];
    for ( int i = 0; i < cells.length; i++ ) {
        cells[i] = ( i == cells.length/2 )? 1 : 0;  // 中央のセルのみ1，それ以外は0で初期化
    }
}

int update(int l, int t, int r) {  /* ルール150に従い，3つのセルの状態(l, t, r)から次の状態を計算 */
    int[] rule150 = {0, 1, 1, 0, 1, 0, 0, 1};  // ルール150における遷移後の状態を持つ配列
    int binnum = 4*l + 2*t + 1*r;              // 3つのセルの状態の列を2進数とみたときの数 [0,1,..,7]

    return rule150[binnum];
}

void updateCells() {
    int[] cpcells = new int[cells.length];  // 遷移前のセルの状態を保持する配列

    /* 遷移前のセルの状態の保存 */
    for ( int i = 0; i < cells.length; i++ ) {
        cpcells[i] = cells[i];
    }
    /* 世代更新（全セルの状態遷移）*/
    for ( int i = 0; i < cells.length; i++ ) {
        cells[i] = update(cpcells[(i-1+cells.length)%cells.length],
                          cpcells[i],
                          cpcells[(i+1)%cells.length]);
    }
}

void drawCells(int gen) {
    for ( int i = 0; i < cells.length; i++ ) {
        fill((cells[i] == 1)? 0 : 255);        // セルの状態が1ならば黒，そうでなければ白
        square(i*SQSIZE, gen*SQSIZE, SQSIZE);  // セルの描画
    }
}
```

付録 B　その他のフラクタル

今回は，比較的簡単に描けるフラクタル図形を取り上げましたが，この他にもフラクタル図形は数多く存在します．特に，Mandelbrot 集合（図 29）は有名なので，見覚えのある方もも多いのではないかと思います．この節では，代表的なフラクタル図形を，解説を省いた図のみではありますが，掲載しておきます．興味がある方は自分の手でコードが書けるか挑戦してみてください．以下の図は全て Processing で描いています．

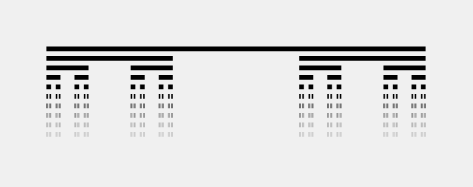


図 13: Cantor 集合

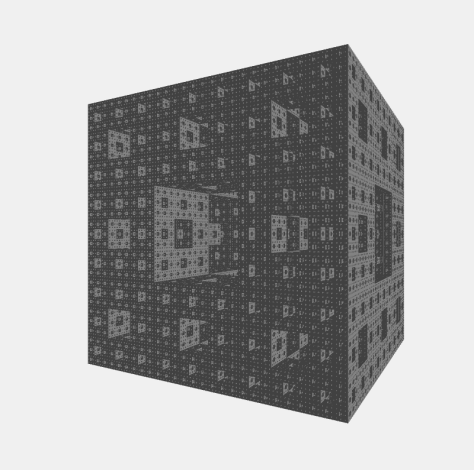


図 14: Menger のスポンジ

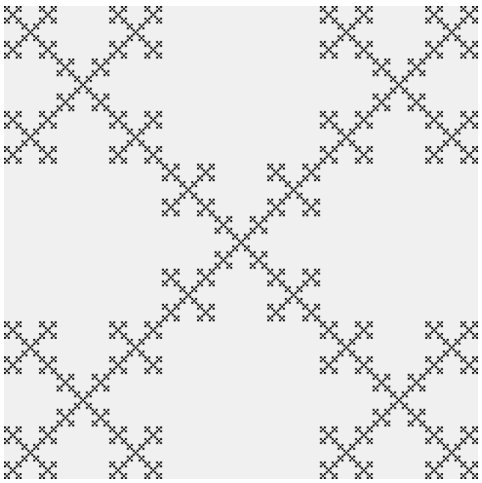


図 15: Vicsek フラクタル (X 字)

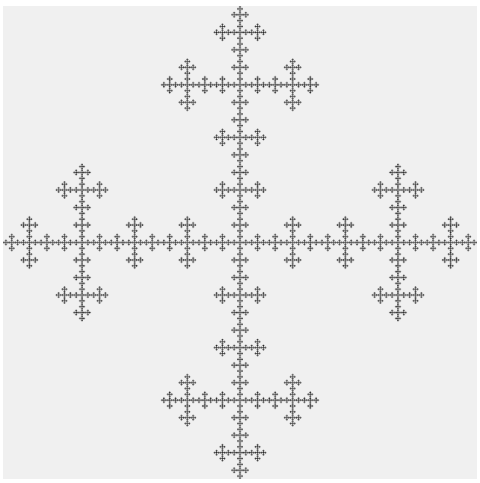


図 16: Vicsek フラクタル (十字)

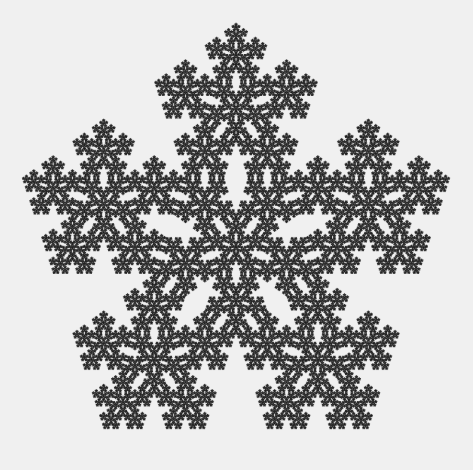


図 17: Pentaflake

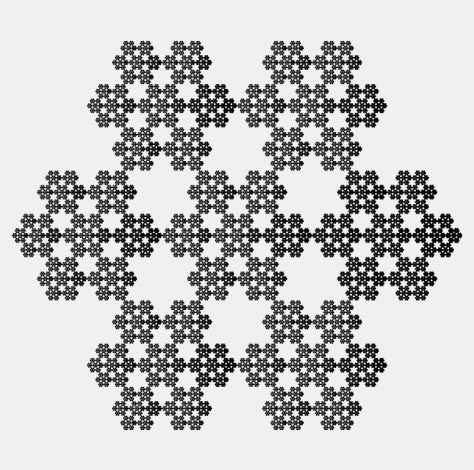


図 18: Hexaflake

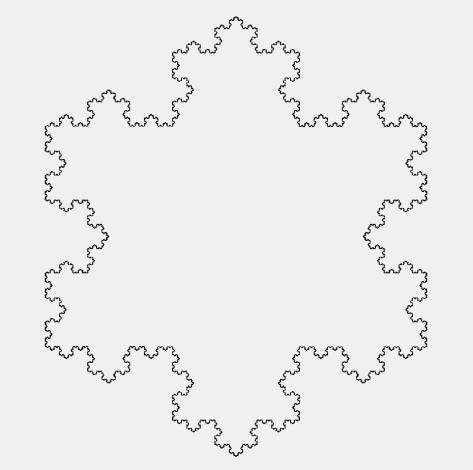


図 19: Koch 雪片

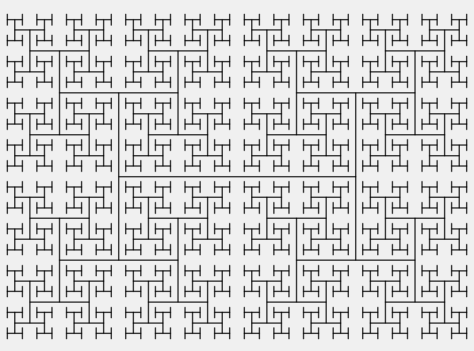


図 20: H 木

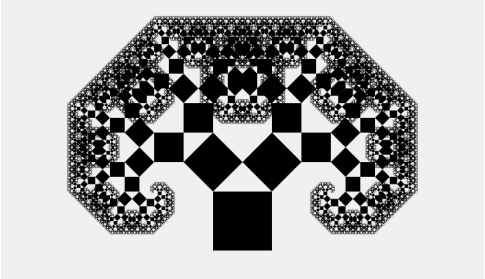


図 21: Pythagoras の木 (対称)

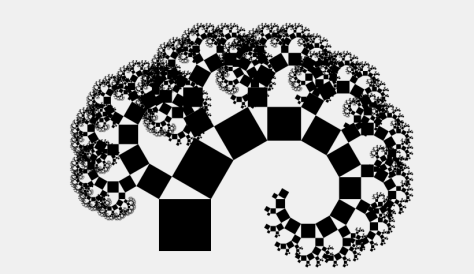


図 22: Pythagoras の木 (非対称)

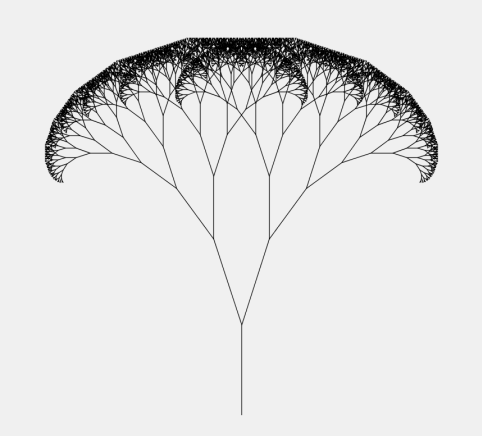


図 23: 樹木曲線 (対称)

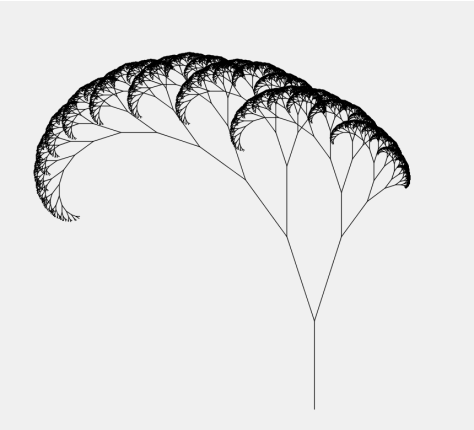


図 24: 樹木曲線 (非対称)

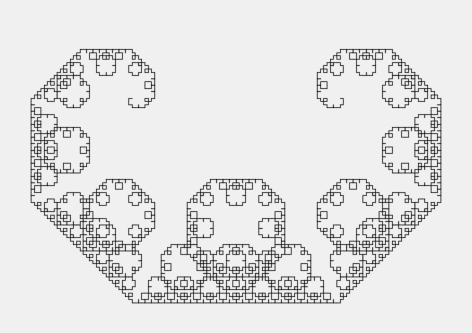


図 25: C 曲線

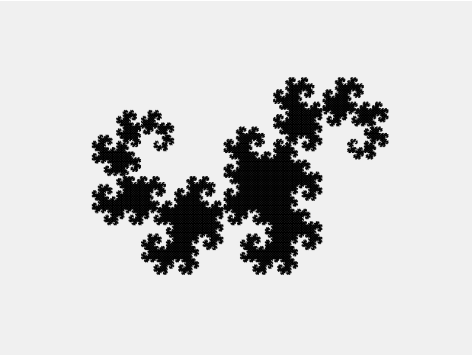


図 26: ドラゴン曲線

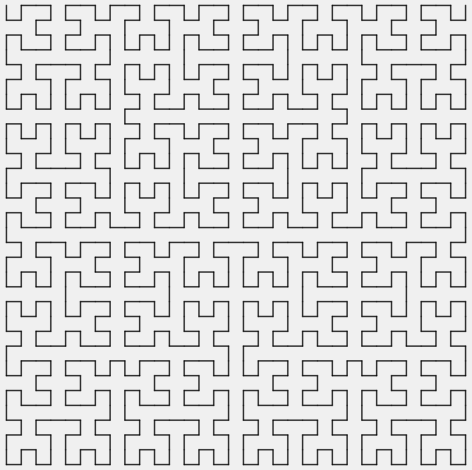


図 27: Hilbert 曲線

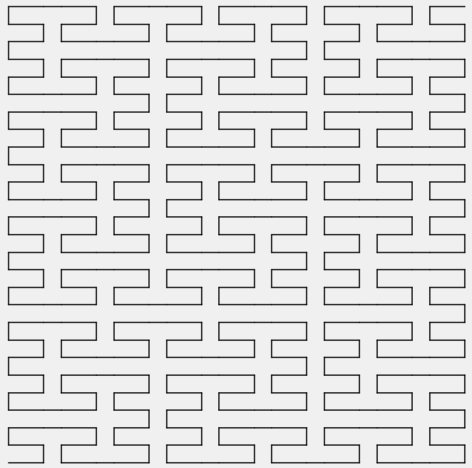


図 28: Peano 曲線

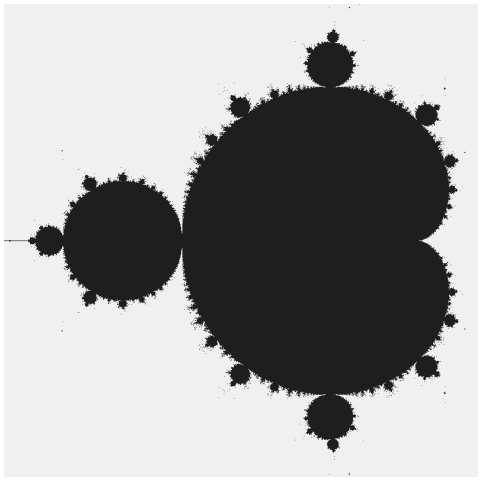


図 29: Mandelbrot 集合

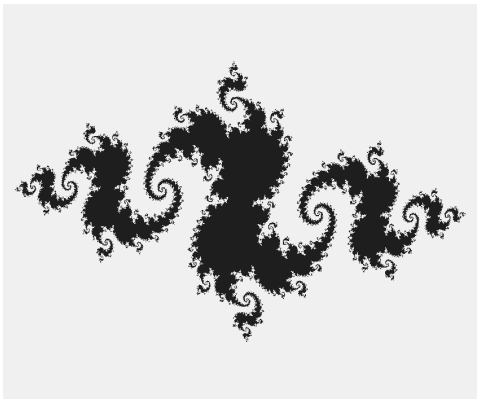


図 30: Julia 集合 (1)

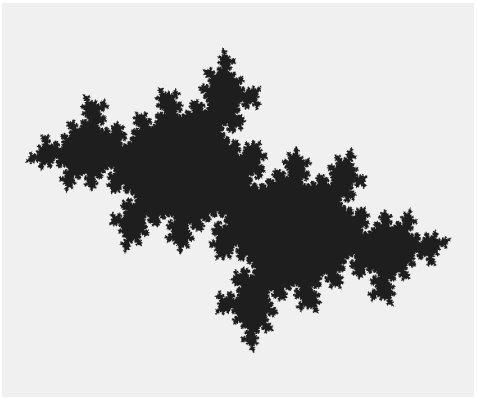


図 31: Julia 集合 (2)

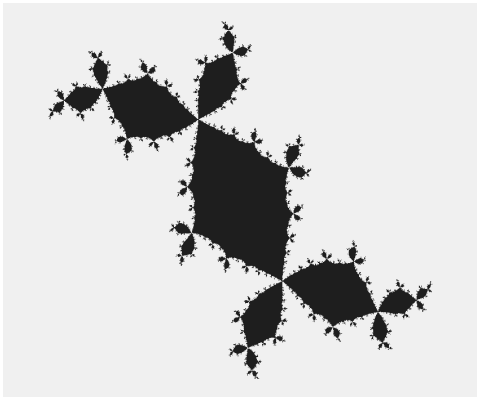


図 32: Julia 集合 (3)

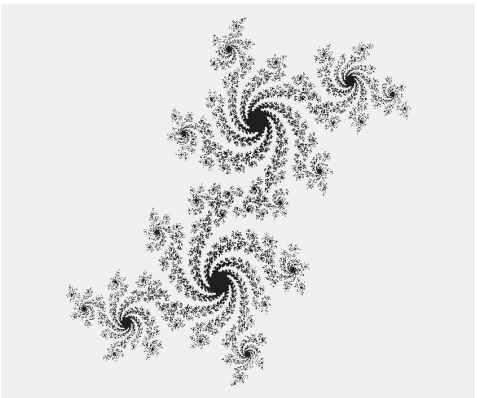


図 33: Julia 集合 (4)

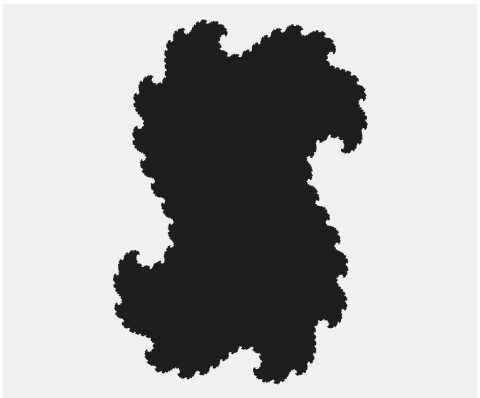


図 34: Julia 集合 (5)

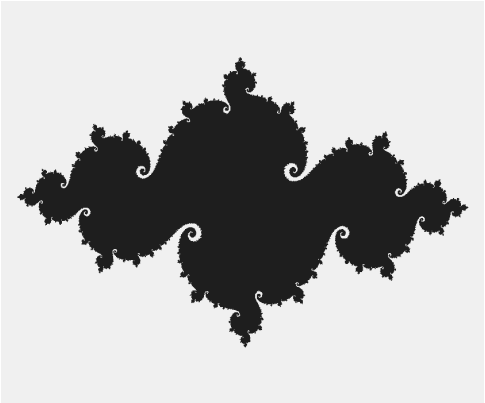


図 35: Julia 集合 (6)

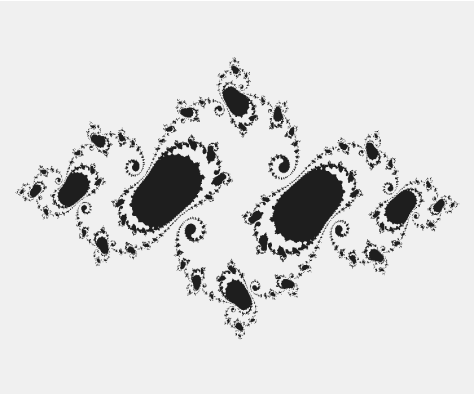


図 36: Julia 集合 (7)