鑑賞するフラクタル

吉永 塁 *

2020 年 7 月 27 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」するの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．題材としてフラクタルとセルオートマトンを取り上げます．理論的な厳密さは不十分かもしれませんが，自分が書いたコード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．ソースコードは後ろのページに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と思います．

1.1　Processing について

今回，図形を描くのに Processing 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．比較的短い記述で簡単に図形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かと思います．コード例は全て Processing で記述されていますが，他の言語で使い馴染んだものがあるならば，そちらを使っていただいても構いません．

注意事項として，Processing の座標は原点が左上で y 軸が下向き正として取られていることに気を付けてください．その他の構文や関数などについて，詳しくは [5] を参照していただければと思います．また，Processing 全般についての書籍として，[6]，[7] が参考になるかと思います．その他にも，[8] では，数学を題材として取り上げています．

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．

また，ある関数定義の中で，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．

この章では再帰関数によってフラクタル図形を描いていきます．

2.1　Sierpinski のカーペット

Sierpinski のカーペットとは，図 1 のような，正方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．

Sierpinski のカーペットをプログラムで描く方法を考えます．Sierpinski のカーペットは次の規則によって得られます．

1. 与えられた正方形を 3×3 の小正方形に分割する．
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．
3. 残りの 8 個の小正方形を 1 の正方形として同様の操作を繰り返す．
4. これらの操作を適当な回数繰り返す．

* 情報工学課程 4 年 b7122062@edu.kit.ac.jp

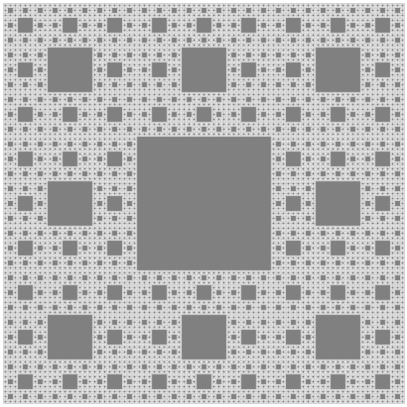

図 1　Sierpinski のカーペット

この操作を再帰関数として記述すると図 2 のようになります.

プログラム全体は 4.1.1 節に記載しました．出力は図 1 のようになります.

```
void carpet(float x, float y, float l, int 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

    return ;
}
```

図 2　Sierpinski のカーペットを描く再帰関数

また，Sierpinski のカーペットを 3 次元の立体へ拡張したものは Menger のスポンジと呼ばれます.

2.2　Sierpinski のギャスケット

Sierpinski のギャスケットとは，図 3 のような，三角形の中心を取り除く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.

Sierpinski のギャスケットは，Sierpinski のカーペットとほぼ同様にして得られます．Sierpinski のカーペットを得る手順において，正方形を正三角形に置き換えるだけです.

Sierpinski のカーペットを描く再帰関数は図 4 のようになります.

プログラム全体は 4.1.2 節に記載しました．出力は図 3 のようになります.

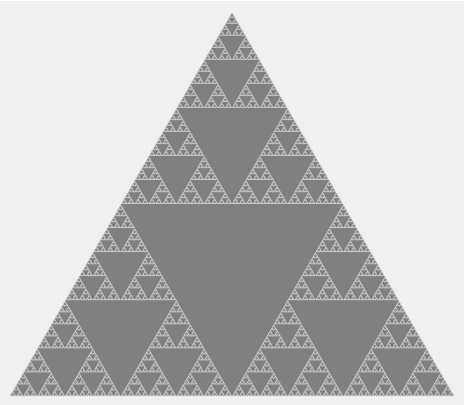

図 3　Sierpinski のギャスケット

```
void gasket(float tx, float ty, float lx, float ly, float rx, float ry, int n) {

    if ( n <= 0 ) {
        return ;
    }

    float lcx = (tx + lx) / 2;  // 上頂点と左頂点の中間点の x 座標
    float lcy = (ty + ly) / 2;  // 上頂点と左頂点の中間点の y 座標
    float rcx = (tx + rx) / 2;  // 上頂点と右頂点の中間点の x 座標
    float rcy = (ty + ry) / 2;  // 上頂点と右頂点の中間点の y 座標
    float bcx = (lx + rx) / 2;  // 左頂点と右頂点の中間点の x 座標
    float bcy = (ly + ry) / 2;  // 左頂点と右頂点の中間点の 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
    gasket(rcx, rcy, bcx, bcy, rx , ry , n-1);  // 右

    return ;
}
```

図 4　Sierpinski のギャスケットを描く再帰関数

2.3　高木曲線

高木曲線とは図 5 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．
高木曲線は次の規則によって得られます．

1. 与えれた区間の中心を頂点とするような三角波を考える．
2. 区間を中心で二分割し，左右の小区間を 1 の区間として同様の操作を繰り返す．
3. これらの操作を適当な回数繰り返す．
4. 各点について，1 から 3 の手順で得られた全ての三角波の高さを足し合わせる．

この操作を再帰関数として記述すると図 6 のようになります．
プログラム全体は 4.1.3 節に記載しました．出力は図 5 のようになります．
また，高木曲線は特徴として，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．

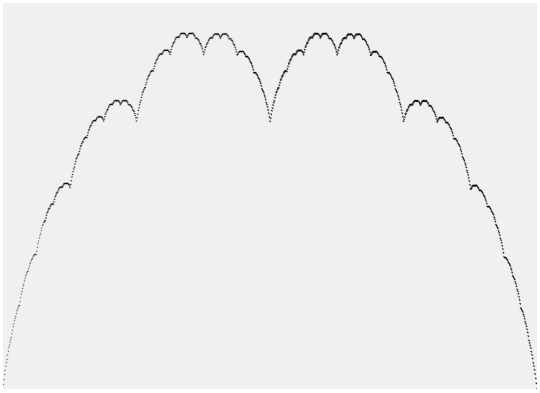

図 5　高木曲線

```
void tcurve(int lx, int rx, int n) {

    if( n <= 0 ){
        return ;
    }

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

    for ( int x = lx; x < cx; x++ ) {
        ys[x] += y++;
    }
    for ( int x = cx; x < rx; x++ ) {
        ys[x] += y--;
    }

    tcurve(lx, cx, n-1);  // 左
    tcurve(cx, rx, n-1);  // 右

    return;
}
```

図 6　高木曲線を描く再帰関数

2.4　Koch 曲線

Koch 曲線とは図 7 のような，線分を三分割し，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．

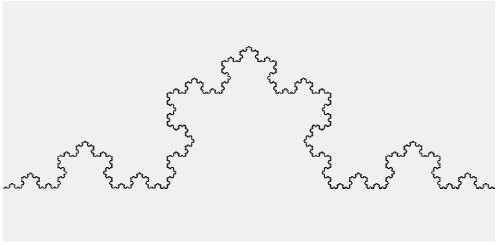

図 7　Koch 曲線

Koch 曲線は次の規則によって得られます．

1. 与えられた線分を 3 つの小線分に三等分する．
2. 中央の小線分を，小線分を一辺とするような正三角形の残りの二辺と置き換える．
3. 1，2 でできた 4 つの小線分を 1 の線分として同様の操作を繰り返す．
4. これらの操作を適当な回数繰り返す．

この操作を再帰関数として記述すると図 8 のようになります．
プログラム全体は 4.1.4 節に記載しました．出力は図 7 のようになります．

```
void kochcurve(float lx, float ly, float rx, float ry, int 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;  // (lx,ly),(rx,ry) の中心 x 座標
    float cy = (ly + ry) / 2;  // (lx,ly),(rx,ry) の中心 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

    return ;
}
```

図 8　Koch 曲線を描く再帰関数

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 33 の様に，Koch 曲線を 3 つ用いて星形のように繋げた図形を Koch 雪片といいます．

問 2.1. Vicsek フラクタル (図 23，図 24) を描くプログラムを作成してください．

問 2.2. Cantor 集合 (図 30) を描くプログラムを作成してください．

問 2.3. Hexaflake(図 32) を描くプログラムを作成してください．正六角形の書き方は，図 9 を参考にしてください．

```
void hexagon(float x, float y, float l) {

    /* (x,y) を中心とする頂点までの長さが l の正六角形 */
    beginShape();
    for ( float t = 0; t < TWO_PI; t += TWO_PI/6 ) {
        vertex(x+l*cos(t), y+l*sin(t));
    }
    endShape(CLOSE);

    return ;
}
```

図 9　正六角形を描く関数

問 2.4. Koch 曲線を描く関数 (図 8) を参考に，Koch 雪片 (図 33) を描くプログラムを作成してください．

2.5　プログラミングにおける再帰

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．今回は，フラクタル図形をを描くために再帰関数を用いましたが，一般のプログラミングにおいても再帰関数は用いられます．例えば，マージソートやクイックソートなどは再帰呼出しを使う有名なアルゴリズムです．これらのアルゴリズムにおける再帰は分割統治[*1]のために用いられます．他にも，再帰関数を使って解く有名な問題として，ハノイの塔などがあります．

また，プログラミングにおける再帰について，例えば Haskell [*2] というプログラミング言語では，命令型言語では一般的な `for` 文や `while` 文のようなループ構造が存在せず，再帰関数が多用されます．

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 が考案したライフゲームが有名です．

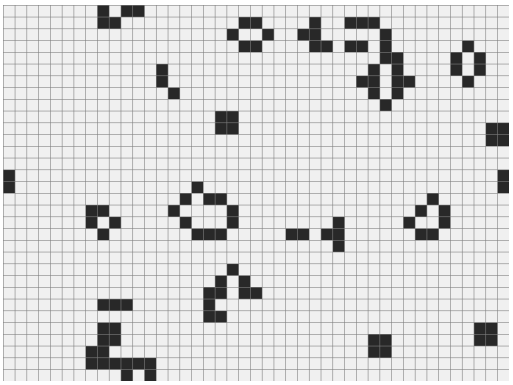

図 10　ライフゲーム
黒いマスが生きたセルを表し，それぞれのセルは周囲のセルの状態によって生存・死滅が決定する．周期的に振動するものや移動するものなど様々なパターンが現れる．

3.1　基本セルオートマトン

セルの状態が $\{0, 1\}$ のいずれかをとり，次の状態が，自身と隣接する二つのセルの状態によって決定されるような一次元セルオートマトンを基本セルオートマトンと呼びます．[*3]またここで，一次元セルオートマトンとは，セルが直線状（一次元）に配置されているようなものを指します．[*4]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．

3.2　状態遷移と Wolfram コード

基本セルオートマトンでは，あるセルの次の状態は，そのセルと隣接する二つ（左隣・右隣）のセルの現在の状態によって決定されます．ここで，この三つのセルの現在の状態より，次の状態を決定する遷移の規則が必要となります．

いま，基本セルオートマトンを考えているので，遷移前の三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．この 8 通りに対して，それぞれ遷移後の状態が 2 通りずつ存在するので，遷移の規則は全部で 2^8 通りだけ存在することになります．

[*1] 問題を解きやすい小さな形へと分割し，小さい問題を解いてそれらを統合することで全体の問題を解くという方法．

[*2] 余談ですが，Haskell は圏論 (category theory) をベースにした，数学的背景を持つプログラミング言語です．C や Java のようなプログラミング言語とは異なり，一風変わった特徴を持っています．興味のある方は触ってみると楽しいかもしれません．

[*3] 特徴を列挙して，一次元二状態三近傍セルオートマトンと呼ぶこともあります．

[*4] セルが平面上に配置されるライフゲームは二次元セルオートマトンです．

この 256 通りの遷移規則について，それぞれに番号を割り当てることを考えます．遷移規則は遷移後のセル状態の組み合わせによって特徴づけられるので，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ることにします．

表 1 のような遷移規則を例として説明します．

自身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	1	0	0	1	0	1	1	0

表 1　遷移規則（ルール 150）

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 $1\times 2^7+1\times 2^4+1\times 2^2+1\times 2^1=150$ なので，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して番号を割り当てます．このようにして割り当てられた数を Wolfram コードと呼びます．[*5]

例 3.1. 表 2 のような遷移規則はルール 90 と呼ばれます．

自身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

表 2　遷移規則（ルール 90）

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．

現在のセルの状態	0	1	0	1	1	1	0	0
自身と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

表 3　ルール 150 に従う遷移計算の例

各セルの状態遷移は表 3 のようにして計算されます．よって，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．コードは図 11 のように書けます．

```
int update(int l, int t, int r) {

    /* 三つのセルの状態を引数に取り，ルール 150 に従って次の状態を返す関数 */
    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;              // 三つのセルの状態の並びを二進数として見たときの数 (0...7)

    return rule150[binnum];
}
```

図 11　ルール 150 に基づく遷移を計算する関数

基本セルオートマトンでは，状態は $\{0,1\}$ の二通りなので，遷移前の三つのセルの状態の並びを二進数として見ると，これは 0 から 7 の自然数になります．よってこれを配列の添字として利用しています．配列にはこれに対応する

[*5] 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[9] の開発でも有名です．

遷移後のセルの状態を格納します．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なります．

問 3.1. 引数によって遷移規則を変更できるように，関数 update を拡張してください．つまり，三つのセルの状態と Wolfram コードを引数として取り，次の状態を返す関数 `int update(int l, int t, int r, int wcode)` を実装してください．正しく実装ができていれば，第四引数に 150 を与えたときに図 11 で示した関数と同じ挙動を示すはずです．

3.3 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．

3.3.1 大域変数

定義する大域変数を図 12 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 int 型配列です．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．

```
/* 大域変数 */
final float SQSIZE = 8.0;  // セルの描画サイズ
int[] cells;               // セルの状態を保持する配列
```

図 12　大域変数

3.3.2 セルの初期化

セルの初期化を行う関数を図 13 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．[*6]

```
void initCells() {

    /* セルの初期化 */
    cells = new int[(int)(width/SQSIZE)];           // 表示ウィンドウに入るだけ
    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
    }

    return ;
}
```

図 13　セルの初期化関数

[*6] Processing において，変数 width は表示ウィンドウの幅を保持するシステム変数です．また，対になるシステム変数 height は表示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，表示ウィンドウ（幅）に入る最大のセル個数は (int)(width/SQSIZE) で得られます．ここで，SQSIZE が float 型で定義されているので，int 型へのキャストが必要となることに注意してください．

3.3.3 世代更新

世代更新を行う関数を図 14 に示します．最初に遷移前のセルの状態を配列 cpcells に保存しています．その後，配列 cpcells が持つ値によって各セルの状態遷移を行っています．[*7] またこのとき，両端は繋がっているとして，遷移を行っています．これは，添字の剰余計算によって実現されます．[*8]

```
void updateCells() {

    /* 遷移前のセルの状態の保存 */
    int[] cpcells = new int[cells.length];      // 遷移前のセルの状態を保持する配列
    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]);
    }

    return ;
}
```

図 14　世代更新関数

3.3.4 セルの描画

セルの描画を行う関数を図 15 に示します．この関数では，世代数 gen を引数に取り，表示ウィンドウの上から gen 行目[*9] に配列 cells が持つセルの状態を描画しています．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）白でセルを描いています．

```
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
    }

    return ;
}
```

図 15　セル描画関数

3.3.5 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．draw 関数を図 16 に示します．まず始めに初期化を行い，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行っています．またここ

[*7] 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．例えば，i = 1 の場合で，cells[1] = update(cells[0], cells[1], cells[2]); とすると，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，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．

[*8] 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．

[*9] このプログラムでは，最終的に，セルの状態を世代順に並べたものを描くので，y 座標はそれぞれの世代数に対応した位置としています．

で，最大の世代は表示ウィンドウを基準に設定しています．[*10]

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

図 16　draw 関数

プログラム全体は 4.1.5 節に掲載しました．実際にはここまでのコードに加え，setup 関数が必要になりますが，セルオートマトンと直接的な関係はないため，説明は省略します．

このプログラムを実行すると図 17 のような出力が得られます．

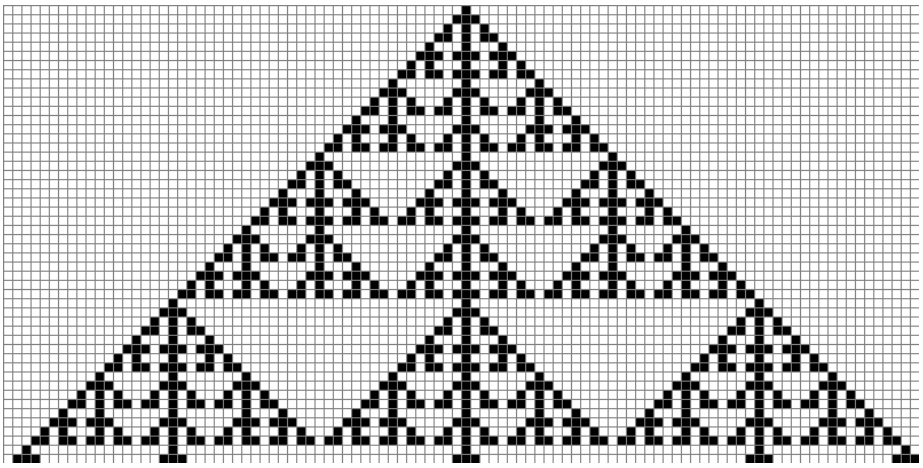

図 17　基本セルオートマトン（ルール 150）

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 update 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．この初期状態の割り当てについて，initCells 関数の for 文の内容を書き換えて，0/1 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．

- 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（左上・上・右上・左・右・左下・下・右下）の状態によって決定する．
- 死んでいるセルに，生きているセルがちょうど 3 つだけ隣接する場合，次の状態を生とする．それ以外の場合には，次の状態を死とする．

[*10] 表示ウィンドウ関連については*6 を参照してください．

- 生きているセルに，生きているセルが 2 つまたは 3 つ隣接する場合，次の状態を生とする．それ以外の場合には，次の状態を死とする．

3.4 余談: Wolfram Alpha

最後に余談として，Wolfram Alpha[9] 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．質問に対して回答を返してくれる質問応答システムです．面倒な計算[*11] 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，問題で与えられたの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て，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．

参考文献

[1] 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

[2] 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

[3] 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

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

[5] Processing.org (`https://processing.org/`)

[6] 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

[7] 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

[8] 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

[9] Wolfram|Alpha (`https://www.wolframalpha.com/`)

[*11] 積分計算や代数計算まで計算してくれます．すごい．

4　付録

4.1　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．紙面の都合上，インデント形式の変更・コメントの省略など，本文中に掲載したものと体裁が異なる場合がありますが，内容面での変更はありません．

4.1.1　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) {

    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

    return ;
}
```

図 18　Sierpinski のカーペット

4.1.2 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;  // 三角形の中心の x 座標
    float cy = 500;        // 三角形の中心の y 座標
    float l = 480;         // 三角形の中心から頂点までの長さ
    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) {

    if ( n <= 0 ) {
        return ;
    }

    float lcx = (tx + lx) / 2;  // 上頂点と左頂点の中間点の x 座標
    float lcy = (ty + ly) / 2;  // 上頂点と左頂点の中間点の y 座標
    float rcx = (tx + rx) / 2;  // 上頂点と右頂点の中間点の x 座標
    float rcy = (ty + ry) / 2;  // 上頂点と右頂点の中間点の y 座標
    float bcx = (lx + rx) / 2;  // 左頂点と右頂点の中間点の x 座標
    float bcy = (ly + ry) / 2;  // 左頂点と右頂点の中間点の 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
    gasket(rcx, rcy, bcx, bcy, rx , ry , n-1);  // 右

    return ;
}
```

図 19　Sierpinski のギャスケット

4.1.3 高木曲線

```
/* Takagi curve */
int[] ys;  // 各点の三角波の高さの和を格納する配列

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
    tcurve(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);
    return ;
}

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
    }
    return;
}

void tcurve(int lx, int rx, int n) {

    if( n <= 0 ){ return ; }

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

    for ( int x = lx; x < cx; x++ ) { ys[x] += y++; }
    for ( int x = cx; x < rx; x++ ) { ys[x] += y--; }

    tcurve(lx, cx, n-1);  // 左
    tcurve(cx, rx, n-1);  // 右

    return;
}

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

図 20　高木曲線

4.1.4 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);

    return ;
}

void kochcurve(float lx, float ly, float rx, float ry, int 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;  // (lx,ly),(rx,ry) の中心 x 座標
    float cy = (ly + ry) / 2;  // (lx,ly),(rx,ry) の中心 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

    return ;
}
```

図 21　Koch 曲線

4.1.5 基本セルオートマトン

```
/* Elementary Cellular Automaton */

final float SQSIZE = 8.0;  // セルの描画サイズ
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)];           // 表示ウィンドウに入るだけ
    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
    }
    return ;
}

int update(int l, int t, int r) {
    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;              // 三つのセルの状態の並びを二進数として見たときの数（0...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
    for ( int i = 0; i < cells.length; i++ ) {
        cpcells[i] = cells[i];
    }
    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]);
    }
    return ;
}

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
    }
    return ;
}
```

図 22　基本セルオートマトン（ルール 150）

4.2　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 4 つ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 36) は有名なので見たことのある方も多いかと思います．

ここでは，その一部を，解説は省いて図のみの形となりますが，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．

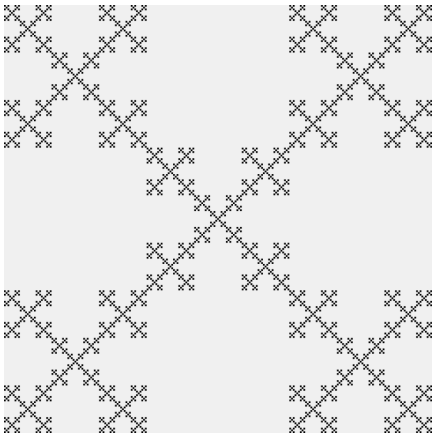

図 23　Vicsek フラクタル (X 字)

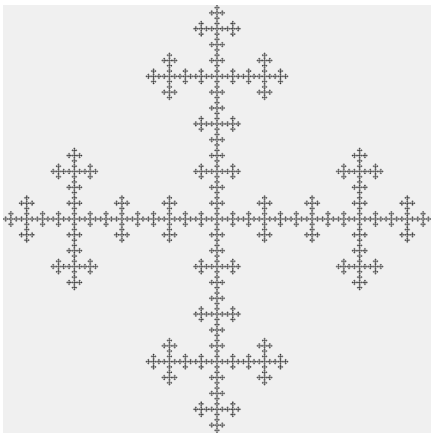

図 24　Vicsek フラクタル (十字)

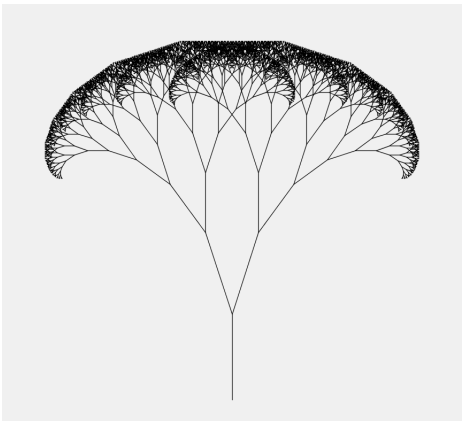

図 25　樹木曲線 (対称)

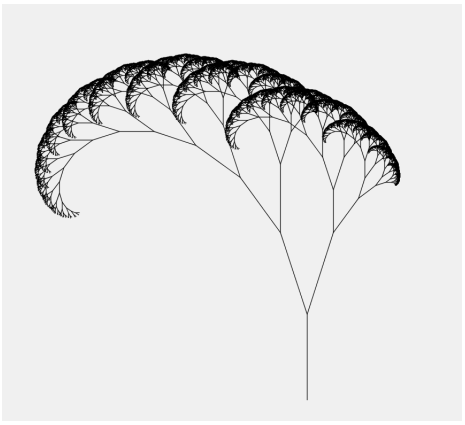

図 26　樹木曲線 (非対称)

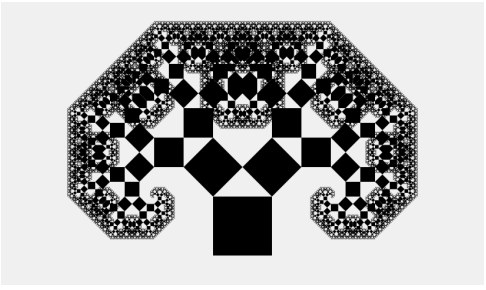

図 27　Pythagoras の木 (対称)

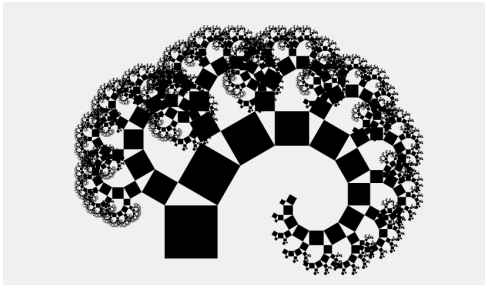

図 28　Pythagoras の木 (非対称)

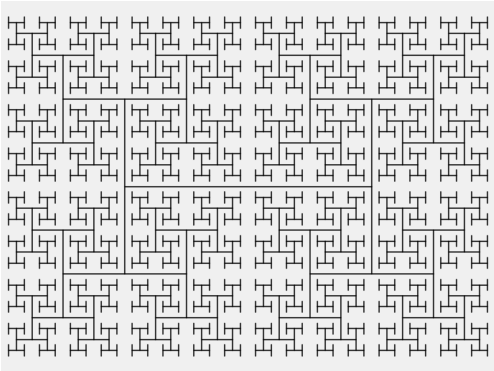

図 29　H 木

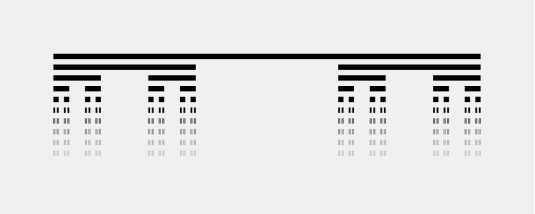

図 30　Cantor 集合

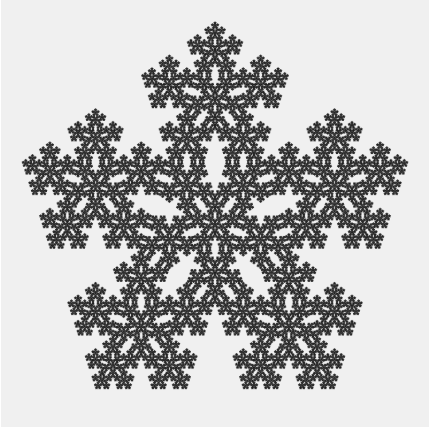

図 31　Pentaflake

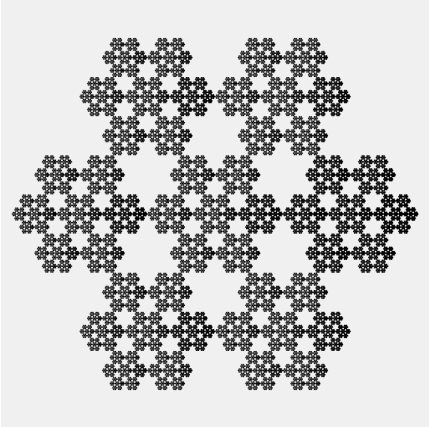

図 32　Hexaflake

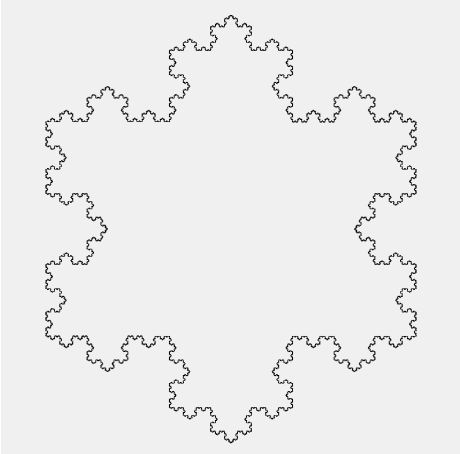

図 33　Koch 雪片

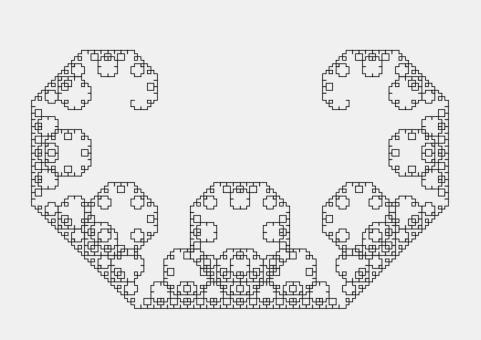

図 34　C 曲線

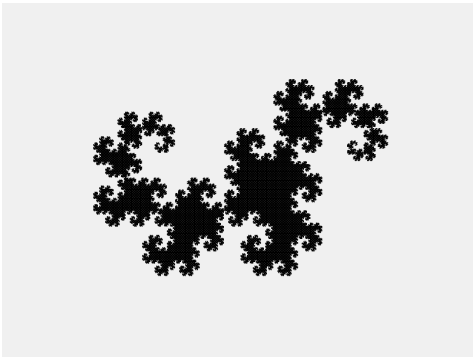

図 35　ドラゴン曲線

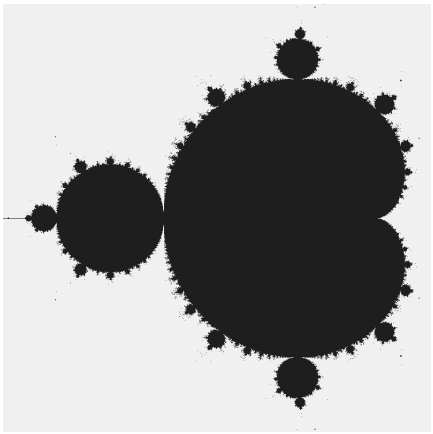

図 36　Mandelbrot 集合

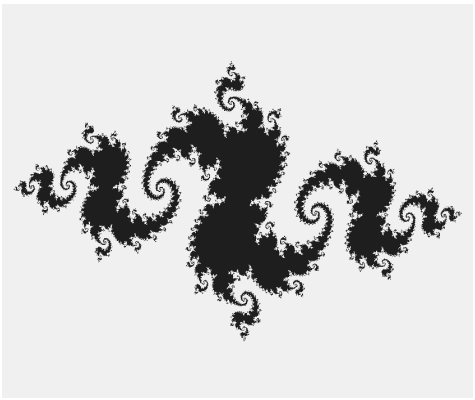

図 37　Julia 集合 (1)

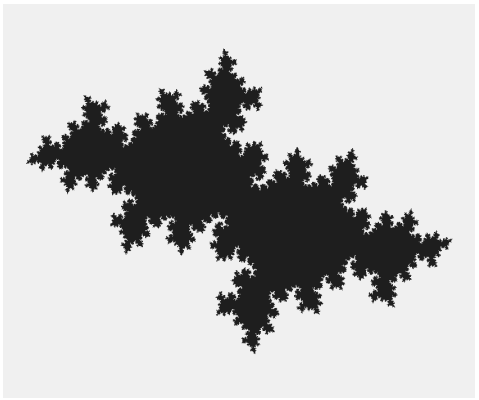

図 38　Julia 集合 (2)

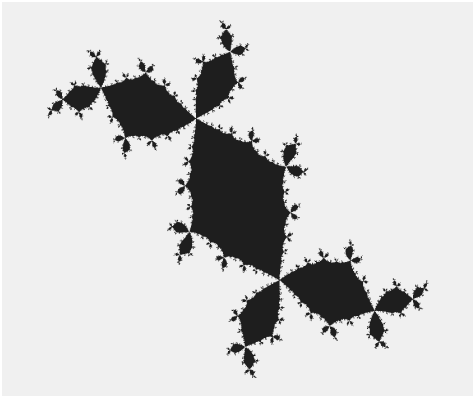

図 39　Julia 集合 (3)

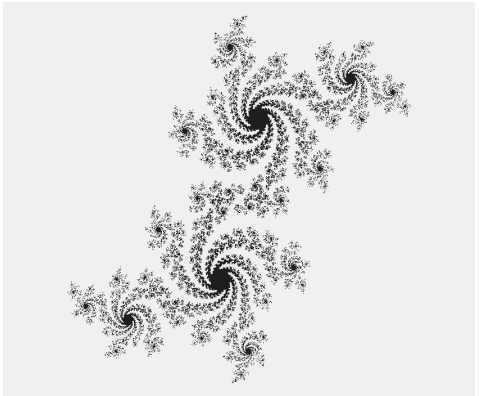

図 40　Julia 集合 (4)

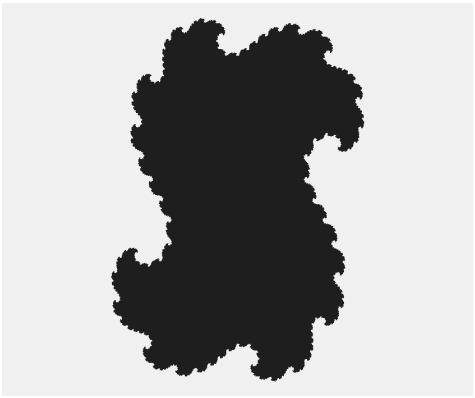

図 41　Julia 集合 (5)

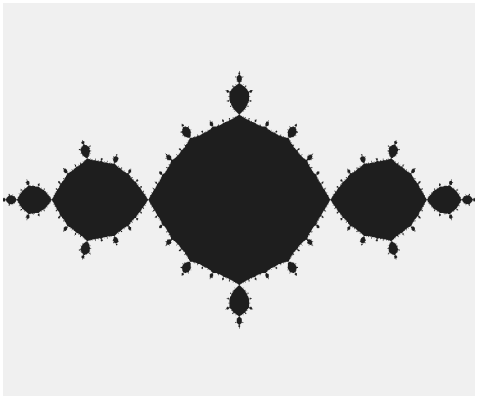

図 42　Julia 集合 (6)

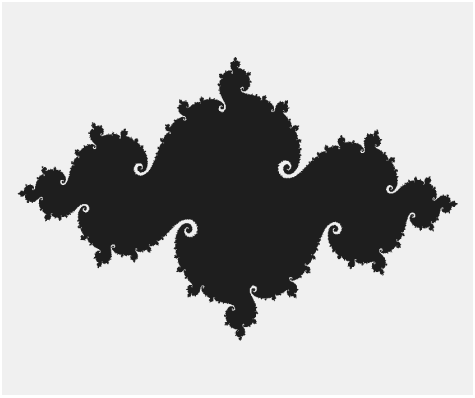

図 43　Julia 集合 (7)

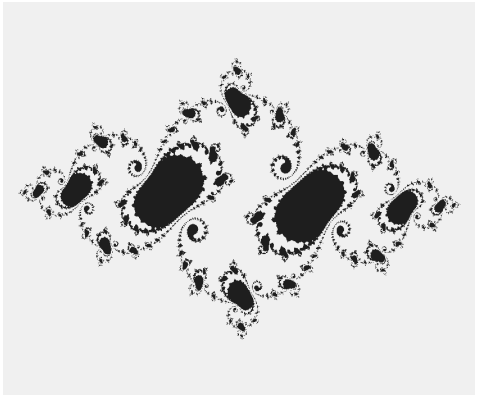

図 44　Julia 集合 (8)

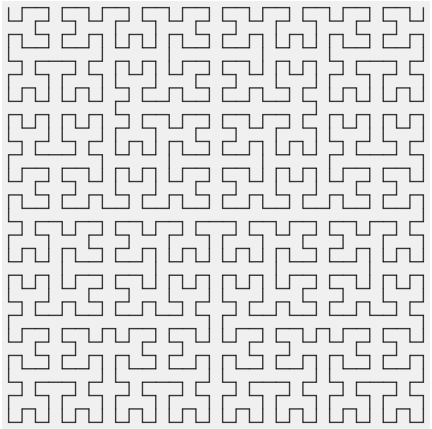

図 45　Hilbert 曲線

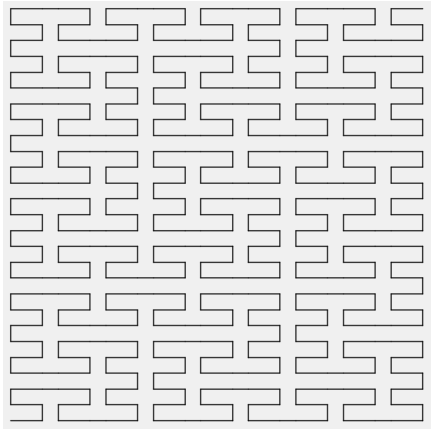

図 46　Peano 曲線