再帰曲線

柳原 宏

山口大工
hiroshi@yamaguchi-u.ac.jp

目次

1 Hilbert 曲線と再帰的アルゴリズム 1

2 Knopp 曲線と Koch 曲線 6

概要

この小論は再帰曲線について説明を行うが, 数学的な厳密性は追求することなくアルゴリズムの説明に主眼をおいて解説が行われていることに注意してほしい.

区間 $[0,1]$ から位相空間への連続な写像のことを曲線と呼ぶ. この定義のもとでは, 直観的に意外なものも曲線に含まれる. Hahn-Mazurkiewicz の定理によれば連結かつ局所連結なコンパクト距離空間は区間 $[0,1]$ からの連続写像による像となっている. 従って例えば区間 $[0,1]$ から正方形への全射となっている曲線が存在する. 歴史的にはこのような曲線の発見の方が先行し, Hahn-Mazurkiewicz の定理につながるわけである. 1890 年 G. Peano は区間 $[0,1]$ から正方形 $[0,1]\times[0,1](\subset\mathbb{R}^2)$ への連続な全射 (Peano 曲線と呼ばれる) が構成できることを発表した. 曖昧な言い方であるが, このように平面または空間のある区域の全てを塗りつぶすような曲線のことを空間充填曲線 (space filling curve) と呼ぶ. この章では Peano の 1 年後に Hilbert が発表した類似の曲線の構成の仕方を解説する. この曲線は Hilbert 曲線と呼ばれるが, その像は正方形 $[0,1]\times[0,1]$ を塗りつぶすので面白い図が描けるわけではない. しかしながら Hilbert 曲線は折れ線よりなる近似曲線列の極限として構成され, 近似曲線のほうはコンピュータグラフィックで興味深い図を描くことが出来る. 筆者の専門外であるが, 一応図の描き方の再帰的アルゴリズムと, Python を用いたコードを解説する. 他に Knopp 曲線や Koch 曲線についても簡単に触れておく.

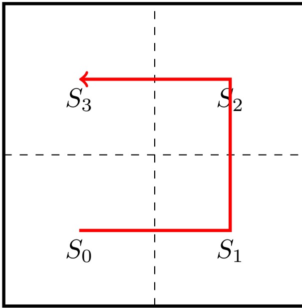
1 Hilbert 曲線と再帰的アルゴリズム

Hilbert 曲線は $n=1,2,3,\ldots$ に対応して曲線の列を構成し, その極限として定義される. 具体的な作り方を述べよう. $n=1$ の場合は区間 $I=[0,1]$ を $I_0=[0,\frac{1}{4})$, $I_1=[\frac{1}{4},\frac{2}{4})$, $I_2=[\frac{2}{4},\frac{3}{4})$, $I_3=[\frac{3}{4},\frac{4}{4})$ と 1 点よりなる集合 $\{1\}$ に分割する. そして今度は正方形 $S=[0,1]\times[0,1]$ の各辺を 2 等分し下の図のように 4 つの正方形に分解する. これら 4 つの正方形のことを第 1 世代の正方形と呼び, 図のように S_i, $i=0,1,2,3$ と番号をつけておく. そして写像 $\varphi_1:[0,1]\to[0,1]\times[0,1]$ を区間 I_i の像 $\varphi_1(I_i)$ が S_i の中心の 1 点のみよりなる集合となるようなものとする. 最後に $1\in[0,1]$ には $[0,1]\times[0,1]$ の左上の点が対応するものとすれば φ_1 が定義されたことになる.

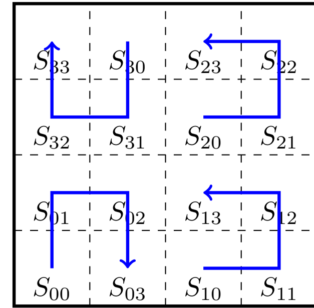
次に I_i, $i=0,1,2,3$ も同様に 4 等分し

$$I_{ij}=\left[\frac{4i+j}{4^2},\frac{4i+j+1}{4^2}\right),\quad j=0,1,2,3$$

と置く．そして第 1 世代の正方形 S_i の各々をさらに 4 等分し $S_{ij},\ j=0,1,2,3$ を作り，下の図のように番号をつける．番号のつけ方は S_0 に含まれる第 2 世代の 4 つの正方形の中で左下のものを S_{00} とし最後の S_{03} の辺が S_1 の辺に含まれるように番号をつける．このとき番号の付け方は一意であることに注意しよう．次に S_1 に含まれる第 2 世代の 4 つの正方形の中で S_{03} と辺を共有するものを S_{10} とし，最後の S_{13} の辺が S_2 の辺に含まれるように番号をつける．S_2, S_3 に含まれる第 2 世代の正方形についても，このように 4 進数で番号をつけていけば，最後の正方形 S_{33} は一番左上に位置する．写像 $\varphi_2 : [0,1] \to [0,1]\times[0,1]$ は $\varphi(I_{ij})$ が S_{ij} の S_k の中心の 1 点のみよりなる集合となり，$1 \in [0,1]$ には $[0,1]\times[0,1]$ の左上の点が対応するものと定義する．



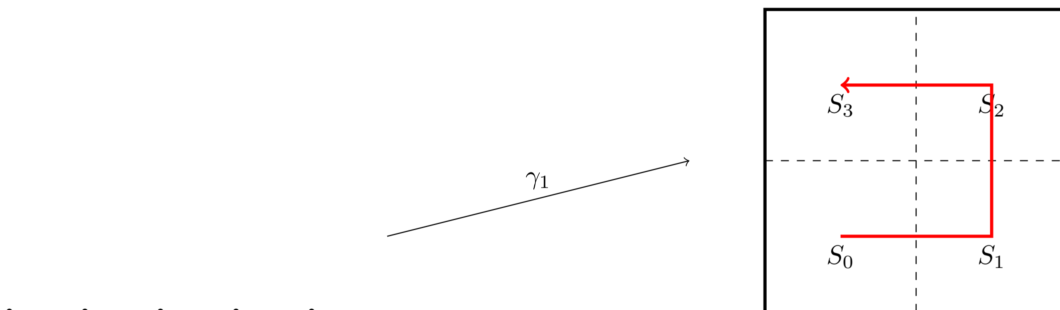
$n=1$



$n=2$

このようにして次々に $I=[0,1]$ と $S=[0,1]\times[0,1]$ を 4 等分しながら φ_k を構成していく．この構成法は $t \in [0,1)$ の 4 進小数展開 $t = 0.n_1n_2\cdots$ と密接な関係がある．実際，n_1 とは $t \in S_i$ を満たす i のことであり，n_2 とは $t \in S_{ij}$ を満たす j のことである．逆に $t = 0.n_1n_2\cdots n_k\cdots$ のとき $\varphi_k(t)$ とは $S_{n_1n_2\cdots n_k}$ の中心のことであり，$\ell > k$ について $\varphi_\ell(t) \in S_{n_1n_2\cdots n_k}$ が成り立つ．これらの事実と $\varphi_k(1)$ が k に関し一定で S の左上の頂点であることより，$\varphi(t) = \lim_{k\to\infty}\varphi_k(t),\ t\in[0,1]$ が存在すること及び極限 φ が連続であることが従う．この $\varphi : I \to S$ を Hilbert 曲線と呼ぶ．$\varphi(I)$ は全ての小正方形 $S_{n_1n_2\cdots n_k}$ について，その中心点を含むので，$\varphi(I)$ は S において稠密である．よって任意の $\mathrm{P}\in S$ について $\varphi(t_n)\to \mathrm{P}$ を満たす I 内の点列 $\{t_n\}_{n=1}^{\infty}$ が存在するが，I はコンパクトであるから収束する部分列 $\{t_{n_k}\}_{k=1}^{\infty}$ を $\varphi(t_{n_k})\to\mathrm{P},\ k\to\infty$ を満たすように取れる．このとき $t_0 = \lim_{k\to\infty} t_{n_k}$ と置けば $\varphi(t_0) = \lim_{k\to\infty}\varphi(t_{n_k}) = \mathrm{P}$ であるから $\mathrm{P}\in\varphi(I)$ である．よって $S \subset \varphi(I)$ となり，$\varphi : I \to S$ は全射である．

Hilbert 曲線を以下に述べる連続曲線の列 $\{\gamma_n\}_{n=1}^{\infty}$ で近似することも出来る．まず γ_1 の定義は $\gamma(0)$ を S_0 の中心点とし，各 $\overline{I_i},\ j=0,1,2$ においては S_i と S_{i+1} の中心を結ぶ線分へのアファイン写像で，各端点において連続になっているものとする．また I_3 の各点と端点 $1\in I$ は S_3 の中心に写像されるものとする．

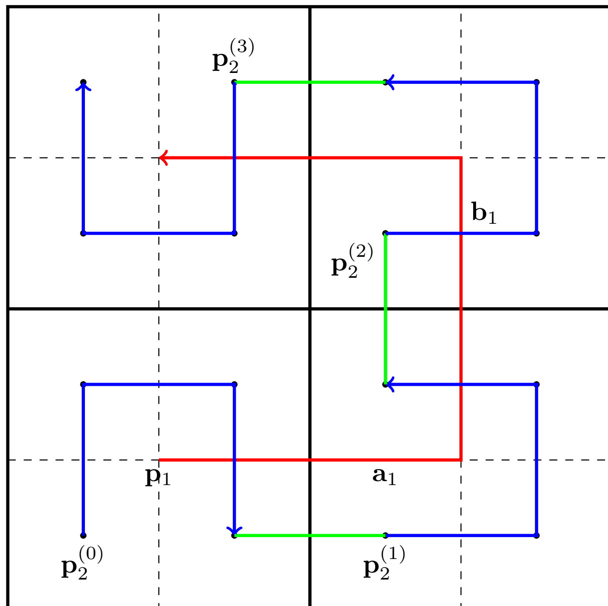


$n=1$

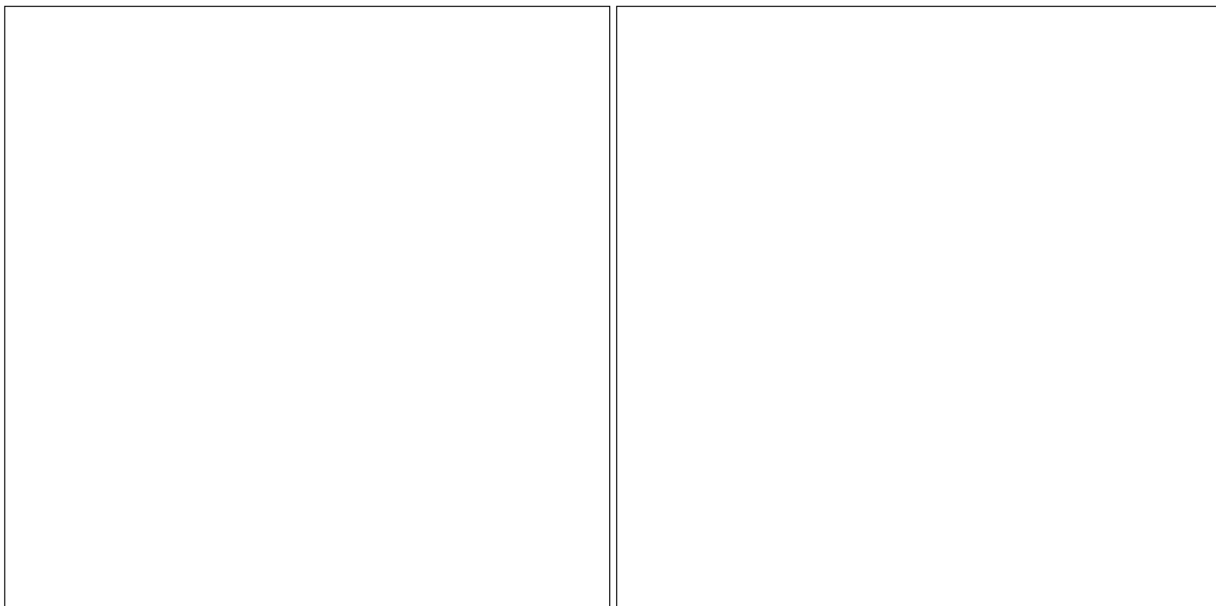
$n=2$ の場合も同様であり γ_2 の定義は $\gamma_2(0)$ を S_{00} の中心点とし，各 $\overline{I_{ij}},\ i,j=0,1,2,3\ (i,j)\neq(3,3)$ に

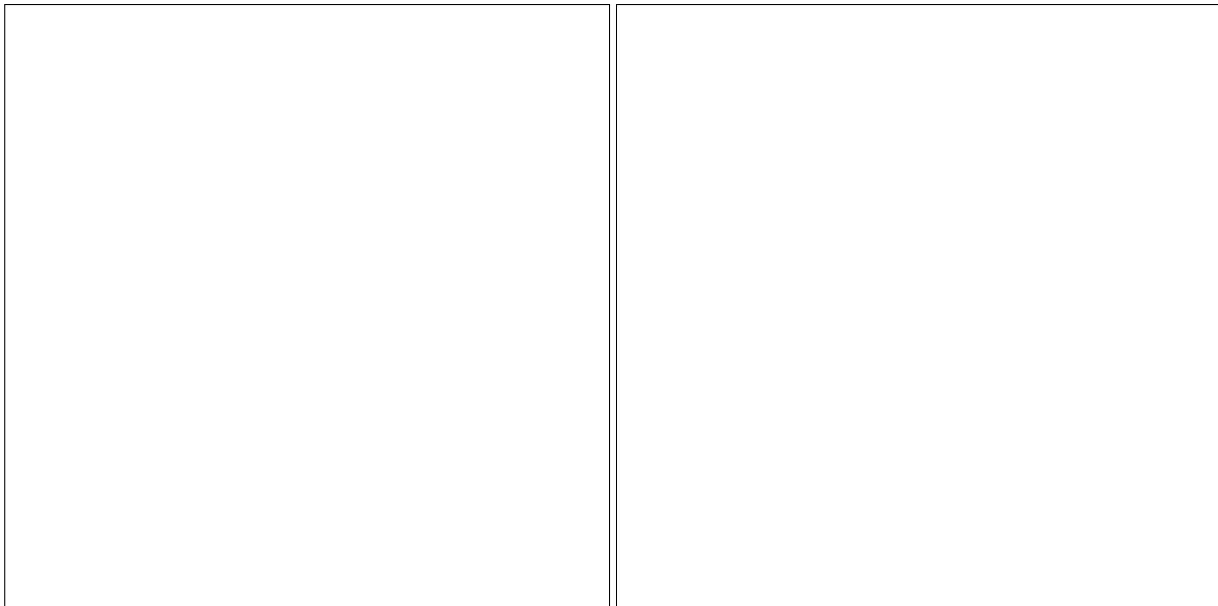
においては S_{ij} と S_{ij+1} (ただし ij を 4 進数とみなし和 $i\pm 1$ を取ること) の中心を結ぶ線分へのアフィン写像で, 各端点において連続になっているものとする. また I_{33} の各点と端点 $1\in I$ は S_{33} の中心に写像されるものとする.

下図では $n=1$ の場合の曲線の像を赤線で表し, $n=2$ の場合の像を青線と緑線で表してある.



$n=3,4,5,6$ については下図のようになる.

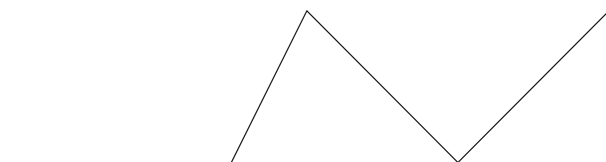




それではこのような図を描くアルゴリズムと Python によるコードを説明しよう．描画には, この本を作るのに使用している LaTeX の描画用パッケージである TikZ/PGF を利用する．例えば折れ線を描くには次のような pgfpicture 環境と`\pgfmoveto` や `\pgflineto` などのコマンドを利用する．

```
\begin{pgfpicture}
\pgfmoveto{\pgfxy(0,0)} % 初期位置を与え
\pgflineto{\pgfxy(3,0)} % 初期位置から与えた座標まで線分を描画
\pgflineto{\pgfxy(4,2)} %1 つ前の座標から与えた座標まで線分を描画
\pgflineto{\pgfxy(6,0)} %1 つ前の座標から与えた座標まで線分を描画
\pgflineto{\pgfxy(8,2)} %1 つ前の座標から与えた座標まで線分を描画
\pgfstroke % 実際に描画
\end{pgfpicture}
```

上記を原稿のファイルに入力し LaTeX で処理を行えば, 以下のように原稿の中に



と図形が出力される．今の場合, 第 n 世代の正方形の中心の座標を次々に求めることが出来れば`\pgflineto` で線分をつないだ折れ線を描画できることになる．そこで 1 つ前の第 $n-1$ 世代の座標から第 n 世代の座標を計算する再帰的な関数によるプログラミングを行うことにする．

再帰的な関数の引数としては, 何を与えればよいであろうか? 上の $n=1,2$ の場合の図中の赤線で示されている第 1 世代の “コ” の字型の 3 頂点を与えて, 第 2 世代の” コ” の字型や, それの左右上下を入れ替えたような図形の 3 頂点を計算するのも一つの方法であるが, 途中の計算の公式が少し面倒になる．そこで “コ” の字型の出発点である第 1 の頂点の位置を位置ベクトル $\mathbf{p}=(x,y)$ を成分を第 1, 2 の引数とし, 次に第 2 の頂点から第 1 の頂点を引いた差分ベクトルの $\mathbf{a}=(a_i,a_j)$ を第 3, 4 の引数とする．そして 第 3 の頂点から第 2 の頂点を引いた差分ベクトル $\mathbf{b}=(b_i,b_j)$ を第 5, 6 の引数とする．最後に再帰の段階を示す引数を n として (x,y,a_i,a_j,b_i,b_j,n) の 7 つを引数として採用しよう．

$n=1,2$ の図では第 1 世代の 4 つの正方形の中心の中で出発点である S_0 の中心を $\mathbf{p}_1$ とし, 第 2 世代については S_{i0} の中心を $\mathbf{p}_2^i$ と表してある.

それでは第 1 世代の $\mathbf{p}_1$, $\mathbf{a}_1$, $\mathbf{b}_1$ から第 2 世代の" コ" の字型の 4 つの図形についてそれぞれの出発点の頂点と進行方向を表すベクトルを求めると

$$\begin{cases}\mathbf{p}_2^{(0)} = & \mathbf{p}_1 - \frac{\mathbf{a}_1+\mathbf{b}_1}{4}, \\ \mathbf{a}_2^{(0)} = & \frac{\mathbf{b}_1}{2}, \\ \mathbf{b}_2^{(0)} = & \frac{\mathbf{a}_1}{2}, \end{cases} \quad \begin{cases}\mathbf{p}_2^{(1)} = & \mathbf{p}_1 + \frac{3\mathbf{a}_1-\mathbf{b}_1}{4}, \\ \mathbf{a}_2^{(1)} = & \frac{\mathbf{a}_1}{2}, \\ \mathbf{b}_2^{(1)} = & \frac{\mathbf{b}_1}{2}, \end{cases} \quad \begin{cases}\mathbf{p}_2^{(2)} = & \mathbf{p}_1 + \frac{3\mathbf{a}_1+3\mathbf{b}_1}{4}, \\ \mathbf{a}_2^{(2)} = & \frac{\mathbf{a}_1}{2}, \\ \mathbf{b}_2^{(2)} = & \frac{\mathbf{b}_1}{2}, \end{cases} \quad \begin{cases}\mathbf{p}_2^{(3)} = & \mathbf{p}_1 + \frac{\mathbf{a}_1+5\mathbf{b}_1}{4}, \\ \mathbf{a}_2^{(3)} = & -\frac{\mathbf{b}_1}{2}, \\ \mathbf{b}_2^{(3)} = & -\frac{\mathbf{a}_1}{2}, \end{cases}$$

となる. また n が所定の世代まで進んだときに" コ" の字型を描画する部分も必要である. 従って実際のコードを Python を用いて書けば

```
\begin{pycode}
import sys, math
def hilbert(x, y, ai, aj, bi, bj, n):
    if n <= 1: % この段階で"コ"の字型を描画
        print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x,y))
        print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x+ai,y+aj))
        print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x+ai+bi,y+aj+bj))
        print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x+bi,y+bj))
    else:
        hilbert(x-(ai+bi)/4,y-(aj+bj)/4,bi/2,bj/2,ai/2,aj/2,n-1)
        hilbert(x+(3*ai-bi)/4,y+(3*aj-bj)/4,ai/2,aj/2,bi/2,bj/2,n-1)
        hilbert(x+(3*ai+3*bi)/4,y+(3*aj+3*bj)/4,ai/2,aj/2,bi/2,bj/2,n-1)
        hilbert(x+(ai+5*bi)/4,y+(aj+5*bj)/4,-bi/2,-bj/2,-ai/2,-aj/2,n-1)
\end{pycode}
```

となる.

但し上のコードは LaTeX の中で Python を用いて関数を定義するために pythontex というパッケージを使用した. このパッケージは LaTeX の中でプログラミングが行えるようになるという便利なものであるが, その使い方についてはネットで調べて頂きたい. この原稿は FreeBSD11.1 上で TeXLive2016 と Python 2.7 を使用し書いているが pythontex を利用するには Python の Pygments パッケージなどのインストールが必要であり, 詳しい説明を行うのは明らかに筆者の力量を越えている.

さてこのように `hilbert(x, y, ai, aj, bi, bj, n)` を定義すれば図を描くのは容易である. 例えば $n=3$ の場合は

```
\begin{pgfpicture}
\hilbert_frame
\py{hilbert_initial_point(2,2,3)}
\py{hilbert(2,2,4,0,0,4,3)}
\pgfstroke
\end{pgfpicture}
```

を入力して LaTeX で処理すればよい. 但し `hilbert_initial_point(2,2,3)` の部分は始点を計算する部分で

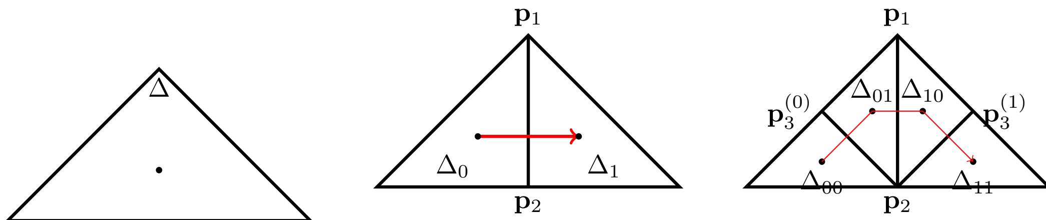
```
\begin{pycode}
import sys, math
def hilbert_initial_point(x,y,n):
        print('\\pgfmoveto{{\\pgfxy({0},{1})}}'.format(x/2**(n-1),y/2**(n-1))
\end{pycode}
```

で定義される．また\hilbert_frame は枠線を引く部分で TEX の \def 文を以下のように用いた．

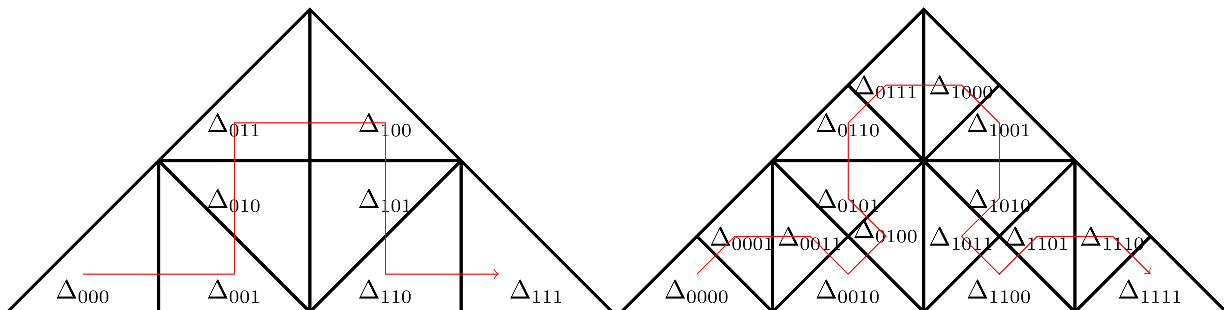
```
\def\hilbert_frame{%8cmx8cm の正方形の枠線
\pgfmoveto{\pgfxy(0,0)}
\pgflineto{\pgfxy(8,0)}
\pgflineto{\pgfxy(8,8)}
\pgflineto{\pgfxy(0,8)}
\pgflineto{\pgfxy(0,0)}
\pgfstroke
}
```

2　Knopp 曲線と Koch 曲線

Knopp 曲線とは次のような近似曲線列の極限として定義される曲線である．但し議論を簡単にするために，以下では像のみを考える．まず Δ を $\mathbb{R}^2$ における 2 等辺三角形で，座標 $(-1,0)$, $(1,0)$, $(1,1)$ を持つ点を頂点とする Δ を斜辺の中点と直角の頂点を結んだ線分で 2 つの直角 2 等辺三角形に分け，左側を Δ_0，右側を Δ_1 と名づけ，それぞれの重心を結んだ線分を描く．次に Δ_0, Δ_0 を同じように 2 等分し Δ_{00},Δ_{01},Δ_{10},Δ_{11}, 作り，この順にそれぞれの重心を結んだ折れ線を描く．



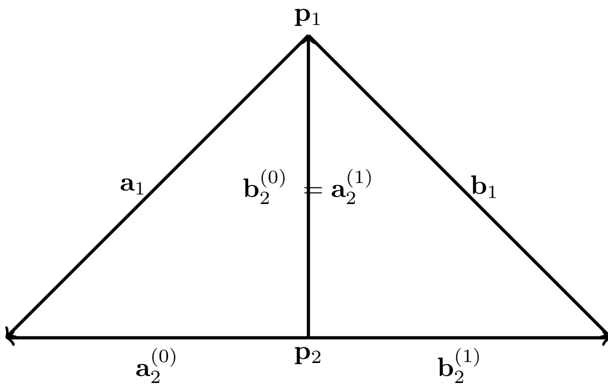
次の段階に進むと細かくなり見難いので拡大した図を描くことにすると



このように直角 2 等辺三角形を，2 等分して細分していき，重心を結んで出来る曲線の列を考えると，その極限は Hilbert 曲線と同様に三角形を塗りつぶす，これを Knopp 曲線という．

それではプログラミングについて考えてみよう．第 1 世代の三角形 Δ の直角の頂点の位置ベクトルを $\mathbf{p}_1$

とし, そこから伸びる 残りの 2 頂点に向かうベクトルを $\mathbf{a}_1$, $\mathbf{b}_1$ とする. 但し, 第 2 世代の三角形 Δ_0 の辺となっている方を $\mathbf{a}_1$ とし, Δ_1 の辺となっている方を $\mathbf{b}_1$ とする. このとき対応する Δ_0, Δ_1 に関する直角の頂点の位置ベクトルと, そこから伸びる 2 本のベクトルは

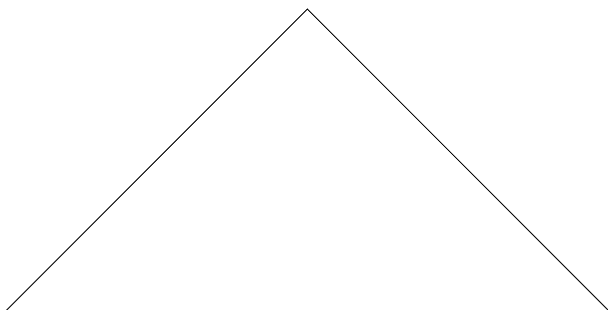


$$\begin{cases} \mathbf{p}_2 = & \mathbf{p}_1 + \frac{\mathbf{a}_1+\mathbf{b}_1}{2}, \\ \mathbf{a}_2^{(0)} = & \frac{\mathbf{a}_1-\mathbf{b}_1}{2}, \\ \mathbf{b}_2^{(0)} = & -\frac{\mathbf{a}_1+\mathbf{b}_1}{2}, \end{cases} \qquad \begin{cases} \mathbf{p}_2 = & \mathbf{p}_1 + \frac{\mathbf{a}_1+\mathbf{b}_1}{2}, \\ \mathbf{a}_2^{(0)} = & -\frac{\mathbf{a}_1+\mathbf{b}_1}{2}, \\ \mathbf{b}_2^{(0)} = & \frac{\mathbf{b}_1-\mathbf{a}_1}{2}, \end{cases}$$

となる. 従ってプログラムとしては, 上の帰納的な式に従い三角形の直角の頂点と 2 本のベクトルの計算を繰り返し, 最終的にに得られた沢山の三角形の重心の座標を出力するものを書き下ろせばよい. 拙いコードを晒すのは気が引ける (筆者はプログラミングに関してはド素人である) が Python を用いたコードを示しておこう.

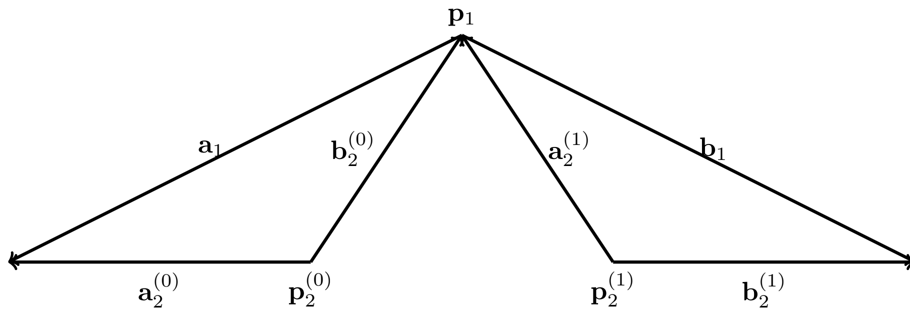
```
\begin{pycode}
import sys, math
def knopp(x, y, ai, aj, bi, bj, n):
  if n <= 0:
    print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x+(ai+bi)/3,y+(aj+bj)/3))
  else:
    knopp(x+(ai+bi)/2,y+(aj+bj)/2,(ai-bi)/2,(aj-bj)/2,-(ai+bi)/2,-(aj+bj)/2,n-1)
    knopp(x+(ai+bi)/2,y+(aj+bj)/2,-(ai+bi)/2,-(aj+bj)/2,(bi-ai)/2,(bj-aj)/2,n-1)
\end{pycode}
```

このコードを用いて $n = 10$ の場合を描画させると次のようになる.



Knopp 曲線は直角三角形から出発したが, これを 120° の角の頂点を持つ 2 等辺三角形に変更する. そして 120° の頂角を 3 等分し, 3 つの三角形に分割し, 真ん中の三角形を取り除く. こうすると, もとの三角形と相

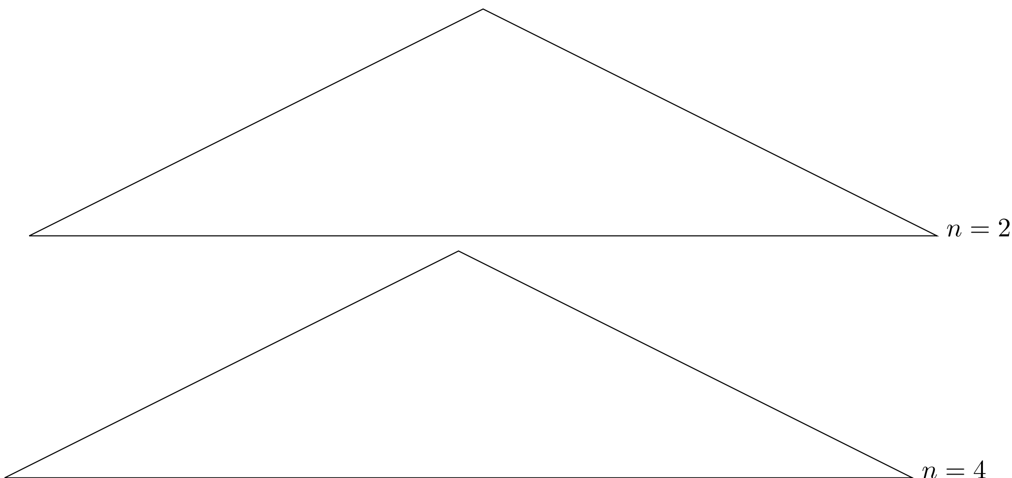
似な 2 つの三角形が得られる．この操作を何度か行い, 得られた三角形の重心を結ぶことにより, 折れ線を作る．この折れ線の極限は Koch 曲線と呼ばれ, 至るところ微分不可能であり, 自分自身と交わらない単純曲線であることが知られている．Knopp 曲線の近似曲線のコードを少し変更すれば Koch 曲線の近似曲線列を描くことも容易である．

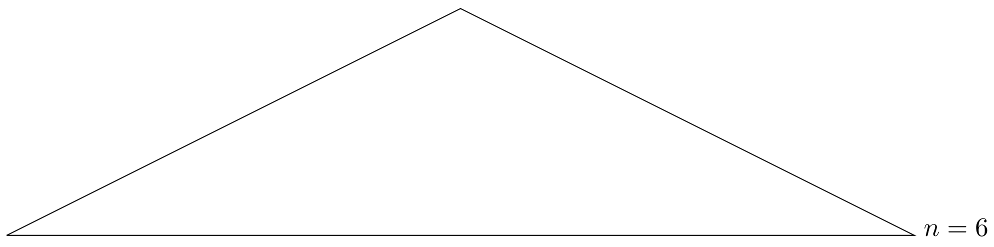


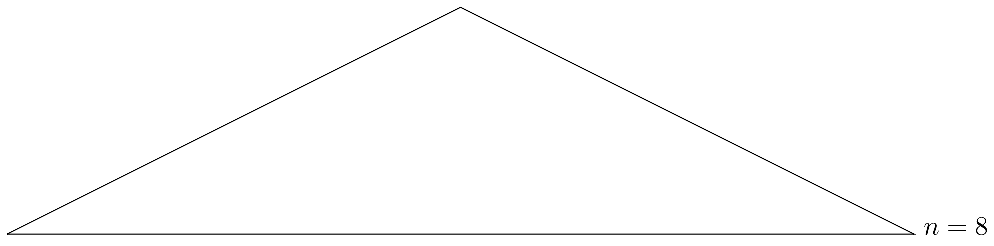
$$\begin{cases} \mathbf{p}_2 = & \mathbf{p}_1 + \frac{2\mathbf{a}_1+\mathbf{b}_1}{3}, \\ \mathbf{a}_2^{(0)} = & \frac{\mathbf{a}_1-\mathbf{b}_1}{3}, \\ \mathbf{b}_2^{(0)} = & -\frac{2\mathbf{a}_1+\mathbf{b}_1}{3}, \end{cases} \qquad \begin{cases} \mathbf{p}_2^{(1)} = & \mathbf{p}_1 + \frac{\mathbf{a}_1+2\mathbf{b}_1}{3}, \\ \mathbf{a}_2^{(0)} = & -\frac{\mathbf{a}_1+2\mathbf{b}_1}{3}, \\ \mathbf{b}_2^{(0)} = & \frac{\mathbf{b}_1-\mathbf{a}_1}{3}, \end{cases}$$

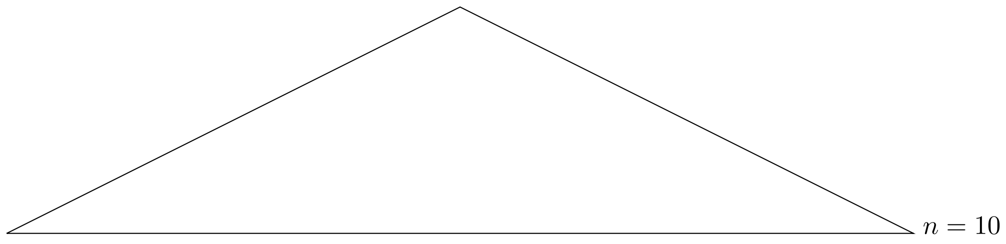
となる．

```
\begin{pycode}
import sys, math
def koch(x, y, ai, aj, bi, bj, n):
  if n <= 0:
    print('\\pgflineto{{\\pgfxy({0},{1})}}'.format(x+(ai+bi)/3,y+(aj+bj)/3))
  else:
    koch(x+(2*ai+bi)/3,y+(2*aj+bj)/3,(ai-bi)/3,(aj-bj)/3,-(2*ai+bi)/3,-(2*aj+bj)/3,n-
1)
    koch(x+(ai+2*bi)/3,y+(aj+2*bj)/3,-(ai+2*bi)/3,-(aj+2*bj)/3,(bi-ai)/3,(bj-aj)/3,n-
1)
\end{pycode}
```









参考文献