Chapter 6

フラクタル図形

6.1 フラクタル

自然界には海岸線や岩山などの複雑な形状が見られるが，**フラクタル** (fractal) と呼ばれる形状に似通った性質を持つと言われている．フラクタル形状はすべての場所において微分不可能な形状で，数学者のマンデルブロ (Mandelbrot) によって提唱された．

フラクタル形状に見られる特徴に**自己相似性** (self-similarity) がある．自己相似性とは，異なるスケールにおいて自らに相似な部分が見られる形状の性質を言う．より具体的には，適当な縮小写像を施すと，もとの形状に相似した形状が複数個見られる．言い方を換えると，フラクタル図形は特定の規則を満たした縮小写像の集合とみなすこともできる．

たとえば，フラクタル図形の代表として知られるものに，マンデルブロ集合とジュリア集合がある．これは，複素数列 $Z_{n+1} = Z_n^2 + C$ を考えるとき，$Z_0 = 0$ として複素数列が発散しない複素数 C の集合がマンデルブロ集合であり，$C =$一定 として複素数列が発散しない初期複素数 Z_0 の集合がジュリア集合となる．これらは，反復適用して発散しない縮小写像を与える点の集合と考えられる．

フラクタル図形を生成する手法に，**反復関数系** (Iterated Function System: IFS) がある．これはアフィン変換などで表せる縮小写像の反復システムであり，図 6.1 に示すコッホ曲線やシェルピンスキー三角形，ドラゴン曲線などのフラクタル図形を生成できる．

フラクタルの大きな特徴として，位相次元とは異なるハウスドルフ次元を持つという性質が挙げられる．通常の位相次元とは，点集合の自由度である．すなわち，集合内の点の近傍が持つ自由度であり，当然のことながら必ず整数値となる．たとえば，単独の点は自由度を持たないので 0 次元，曲線上の点は曲線方向の 1 自由度のみを持つので 1 次元，曲面上の点の近傍は平面と位相的に等しいので 2 自由度を持ち 2 次元となる．

これに対して，**ハウスドルフ次元**とは，形状の量を測るためのハウスドルフ測度の次元を表す．たとえば，曲線 (1 次元) であれば長さ，曲面 (2 次元) であれば面積，立体 (3 次元) であれば体積が量を表す．ここで，スケールが s となる状況 (1 辺の長さが s 倍になる状況) を考えると，それぞれの量は，長さは s 倍，面積は s^2 倍，体積は s^3 倍というように，形状の次元数 n に対して s^n 倍となる．そこで逆にスケール s に対する量の次数 n をハウスドルフ次元とする．フラクタル図形を位相次元の量で測ろうとすると，無限大 (ないしはゼロ) になってしまうが，ハウスドルフ次元を導入することによって量を測ることが可能になる．詳細は後に述べるが，コッホ曲線の場合には，ハウスドルフ次元は $\log_3 4 = 1.26186$ となる．

6.2 フラクタル図形のプログラム

例１： フラクタル図形 `Fractal` — `fractal.py`

フラクタル図形を描くための (抽象) クラスであり，このクラス単独では利用できない．`self.base` は基本図形の点列を表し，`numpy.array` の行列 `self.mats` とベクトル `self.vecs` は，縮小写像の行列と平行移動ベクトルを表している．`Fractal` の初期化メソッド`__init__`ではフラクタルオブジェクトの初期化を行う．すなわち，フラクタル図形を描画する際の再帰の回数を表す `self.times` を設定するとともに，描画先の `self.canvas` と，基本図形の点列である `self.base`，縮小写像の行列群と平行移動ベクトル群である `self.mats`，`self.vecs` を設定している．

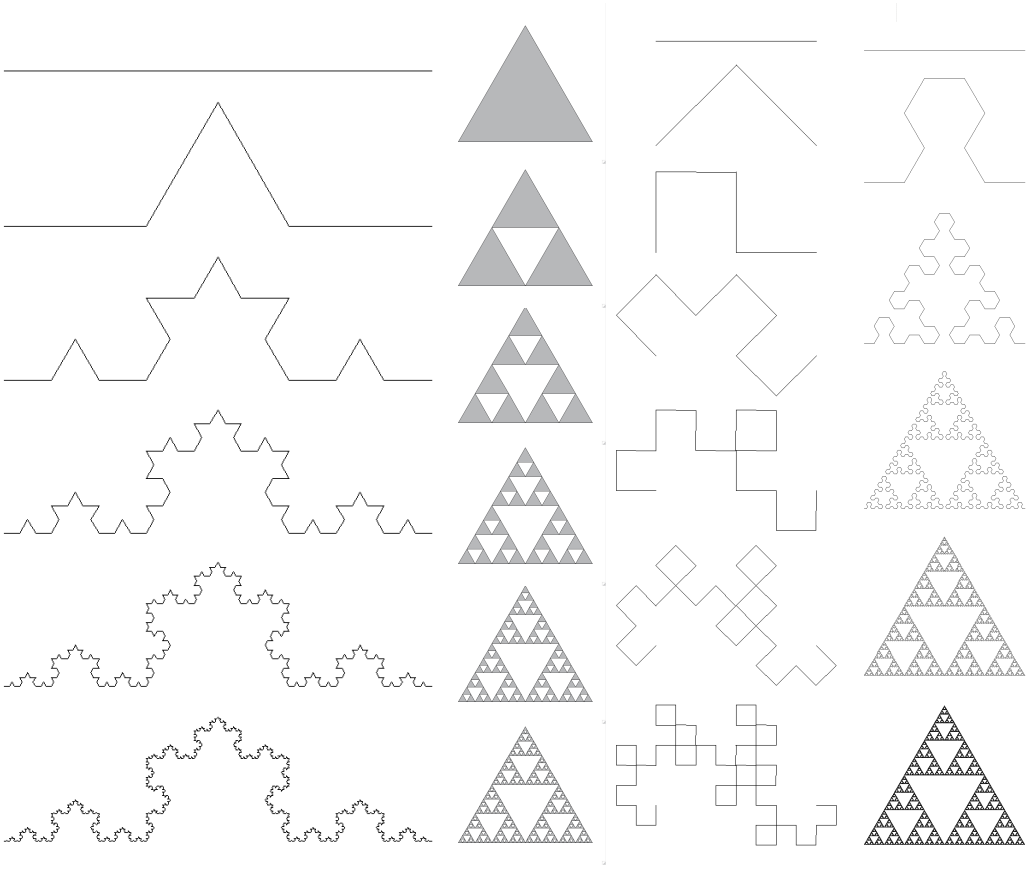

Figure 6.1: コッホ曲線，シェルピンスキー三角形，ドラゴン曲線，シェルピンスキー三角形 (曲線)

フラクタル図形描画の中心となるメソッドは，3引数からなる drawFractal メソッドで，最初の引数 l は残りの再帰回数を表している．最初の引数 l が省略されるか負の場合には，再帰の回数 l を self.times として描画を開始する．2 番目と 3 番目の引数 mat，vec は，描画しようとしている図形の縮小写像（これまで累積された縮小写像）を表すアフィン変換である．言い換えれば，描画の際に適用すべきアフィン変換の行列と平行移動ベクトルを表している．これらが省略された場合には，それぞれ恒等行列とゼロベクトル，すなわち恒等写像で描画を開始する．l が 0 よりも大きい場合には，引数で与えられたアフィン変換 mat，vec に，縮小写像の 1 つ self.mats[i]，self.vecs[i] を適用した上で自分自身（self.drawFractal）を再帰的に呼び出している．一方，l が 0 であった場合には，再帰はせずに実際の描画を行なう．その際には引数で与えられたアフィン変換 mat，vec を基本図形の点データに適用してから，self.drawObject を呼び出して基本図形を描画する．このクラスを利用するためには，初期化メソッド __init__ と実際の描画を行なう drawObject メソッドを実装しなくてはならない．特に，__init__メソッドでは，描画対象領域となるワールド座標系の指定，フラクタルの基本図形の点群である self.base ならびに縮小写像群を与える self.mats と self.vecs を定義する必要がある．

```
import sys                                  # sys モジュールの import
import numpy as np                          # numpy モジュールの import (np で)
from myCanvas import MyCanvas               # myCanvas モジュールの import

class Fractal(object):                      # Fractal クラスの定義
  def __init__(self, canvas, base, mats, vecs): # 初期化メソッド
    '''
    canvas - 描画する canvas
    base - 基本図形の点データ
    mats - 縮小写像（アフィン変換）の行列群
    vecs - 縮小写像（アフィン変換）の平行移動ベクトル群
```

```
    Fractal オブジェクトを初期化する
    '''
    if len(sys.argv) > 1:                # シェル引数がある場合
      num = sys.argv[1]                  # 第 1 引数を頂点数の文字列
    else:                                # シェル引数がない場合
      num = input('# of iterations -> ') # 再帰回数の文字列を入力
    self.times = int(num)                # 再帰回数の設定
    self.canvas = canvas                 # canvas の設定
    self.base, self.mats, self.vecs = (base, mats, vecs) # 基本図形と縮小写像の設定

  def drawFractal(self, l=-1, mat=np.array(((1,0), (0,1))), vec=np.array((0,0))):
                                         # Fractal オブジェクトの描画メソッド
    '''
    l        - 再帰のレベル，省略時 -1
    mat, vec - 累積された縮小写像，省略時 恒等行列，零ベクトル
    l < 0 (= -1) ならば再帰回数 (self.times) を設定して描画を始める
    l > 0 ならば縮小写像の 1 つをさらに適用して l-1 レベルの描画を行う
    l = 0 ならば累積された縮小写像を用いて基本図形を描画 (drawObject) する
    '''
    if l < 0:                            # 引数なしの実行 (最初の実行)
      l = self.times                     # 再帰回数を設定し，描画を始める
    if l > 0:                            # 再帰の最終段階に達していない
      for i in range(len(self.mats)):    # 縮小写像の数だけ反復
        self.drawFractal(l-1, mat.dot(self.mats[i]), mat.dot(self.vecs[i])+vec)
                                       # i 番目の縮小写像を適用して l-1 レベルの描画を行う
    else:                                # これ以上は再帰せず，基本図形を描画する
      points = []                        # 基本図形の点データの初期化
      for i in range(len(self.base)):    # 点データの個数分だけ反復
        points.append(mat.dot(self.base[i])+vec) # 点データに累積された縮小写像を施す
      self.drawObject(points)            # 基本図形の描画
```

例２： コッホ曲線 — koch.py

コッホ曲線は反復関数系で生成可能なフラクタル図形である．コッホ曲線生成の 1 ステップは，図 6.1 左のように，1 本の線分を 3 等分し，中央の 3 分の 1 を除去して，正三角形の 2 辺を加える操作である．言い方を変えると，基本図形である線分が与えられると，1/3 の縮小と回転，平行移動からなる縮小写像が施された 4 つの線分になる．この操作を反復すると，4 回目には，図 6.2 のような図形が得られる．また，1 回の操作では 1/3 の長さの線分が 4 本になるので，コッホ曲線の長さは毎回 4/3 倍になる．これを無限に繰り返すと，コッホ曲線の全長は無限に長くなる．そこでハウスドルフ次元を考える．スケールが 3 倍となるときに全長が 4 倍となるので，ハウスドルフ次元を n とすると，$3^n = 4$ で， $n = \log_3 4 = 1.26186$ となる．

```
import math                              # math モジュールの import
import numpy as np                       # numpy モジュールの import (np で)
from vectorMatrix import rotMatrix, scaleMatrix
                       # vectorMatrix モジュールの rotMatrix, scaleMatrix 関数の import
from myCanvas import MyCanvas            # myCanvas モジュールの import
from fractal import Fractal              # fractal モジュールの import

class Koch(Fractal):                     # Koch クラス (Fractal) の定義
  def __init__(self, canvas):            # 初期化メソッド
    '''
    canvas - 描画する canvas
    Koch オブジェクトを初期化する
    '''
    base = [np.array((0, 0)), np.array((1, 0))] # 基本図形 (線分) の両端点
    mats = [scaleMatrix(1/3)]            # 縮小写像の行列 - 0
    mats.append(scaleMatrix(1/3).dot(rotMatrix(math.pi/3)))  # 縮小写像の行列 - 1
    mats.append(scaleMatrix(1/3).dot(rotMatrix(-math.pi/3))) # 縮小写像の行列 - 2
    mats.append(scaleMatrix(1/3))        # 縮小写像の行列 - 3
    vecs = [base[0]]                     # 縮小写像の平行移動ベクトル - 0
    vecs.append(mats[0].dot(base[1]) + vecs[0]) # 縮小写像の平行移動ベクトル - 1
    vecs.append(mats[1].dot(base[1]) + vecs[1]) # 縮小写像の平行移動ベクトル - 2
    vecs.append(mats[2].dot(base[1]) + vecs[2]) # 縮小写像の平行移動ベクトル - 3
    super().__init__(canvas, base, mats, vecs)  # Fractal オブジェクトの初期化
```

```
  def drawObject(self, pnts):              # 基本図形の描画メソッド
    '''
    pnts - 基本図形の点データ
    基本図形（線分）を描画する
    '''
    self.canvas.drawPolyline(pnts)         # 線分の描画

def main():                                # main 関数
  canvas = MyCanvas(xo = 50, r = 1.2)      # MyCanvas の作成
  Koch(canvas).drawFractal()               # Koch オブジェクトの描画
  canvas.mainloop()                        # ルートフレームの実行ループ開始

if __name__ == '__main__':                 # 起動の確認（コマンドラインからの起動）
  main()                                   # main 関数の呼出
```

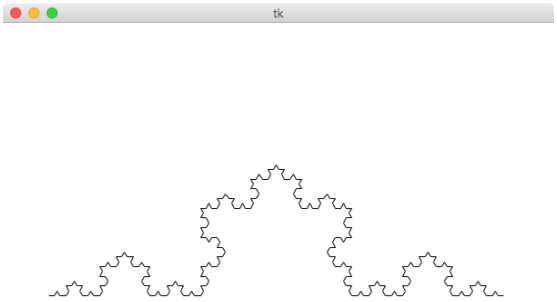

Figure 6.2: コッホ曲線

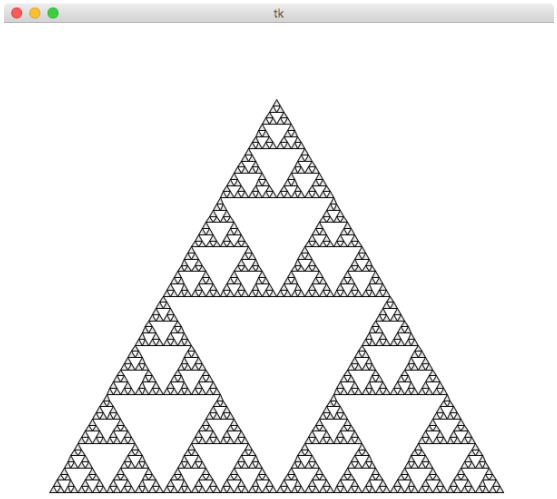

Figure 6.3: シェルピンスキー三角形

例３： シェルピンスキー三角形 — `sierpinski.py`

シェルピンスキー三角形も反復関数系で生成可能なフラクタル図形である．シェルピンスキー三角形生成の 1 ステップは，図 6.1 中央のように，三角形の各辺を 2 等分して全体を 4 つの三角形とし，そのうちの中央の三角形を取り除く．この操作を反復すると，6 回目には，図 6.3 のような図形が得られる．1 回の操作では長さに関して 1/2 の三角形が 3 つになるので，シェルピンスキー三角形の長さ (周囲長) は毎回 3/2 倍，面積は毎回 3/4 倍になる．これを無限に繰り返すと，シェルピンスキー三角形の全長は無限に長くなり，面積はゼロになる．そこで，ここでもハウスドルフ次元を考える．スケールが 2 倍となるときに全長が 3 倍となるので，ハウスドルフ次元は， $n = \log_2 3 = 1.58496$ となる．

```
import math                                # math モジュールの import
import numpy as np                         # numpy モジュールの import（np で）
from vectorMatrix import scaleMatrix
                                  # vectorMatrix モジュールの scaleMatrix 関数の import
from myCanvas import MyCanvas              # myCanvas モジュールの import
from fractal import Fractal                # fractal モジュールの import

class Sierpinski(Fractal):                 # Sierpinski クラス（Fractal）の定義
  def __init__(self, canvas):              # 初期化メソッド
    '''
    canvas - 描画する canvas
    Sierpinski オブジェクトを初期化する
```

```
    '''
    base = [np.array((0, 0)),             # 基本図形（三角形）の頂点
            np.array((-1, -3**0.5)), np.array((1, -3**0.5))]
    mats = [scaleMatrix(1/2)]             # 縮小写像の行列 - 0
    mats.append(scaleMatrix(1/2))         # 縮小写像の行列 - 1
    mats.append(scaleMatrix(1/2))         # 縮小写像の行列 - 2
    vecs = [base[0]]                      # 縮小写像の平行移動ベクトル - 0
    vecs.append(mats[0].dot(base[1]) + vecs[0]) # 縮小写像の平行移動ベクトル - 1
    vecs.append(mats[0].dot(base[2]) + vecs[0]) # 縮小写像の平行移動ベクトル - 2
    super().__init__(canvas, base, mats, vecs)  # Fractal オブジェクトの初期化

  def drawObject(self, pnts):             # 基本図形の描画メソッド
    '''
    pnts - 基本図形の点データ
    基本図形（三角形）を描画する
    '''
    self.canvas.drawPolygon(pnts)         # 多角形の描画

def main():                               # main 関数
  canvas = MyCanvas(yo = 84, r = 2.4)     # MyCanvas の作成
  Sierpinski(canvas).drawFractal()        # Sierpinski オブジェクトの描画
  canvas.mainloop()                       # ルートフレームの実行ループ開始

if __name__ == '__main__':                # 起動の確認（コマンドラインからの起動）
  main()                                  # main 関数の呼出
```

章末課題

ドラゴン曲線のプログラム

例 2「コッホ曲線」を参考に，図 6.1 のドラゴン曲線を描くプログラムを作成せよ．図 6.4 は 12 ステップ後の状態を示したものである．ドラゴン曲線は 1 ステップごとに，1 辺が縮小された 2 辺になっているが，2 辺の並び方 (それぞれの向き) に注意すること．また，ドラゴン曲線のハウスドルフ次元を計算し，そこから想像されることを述べよ．

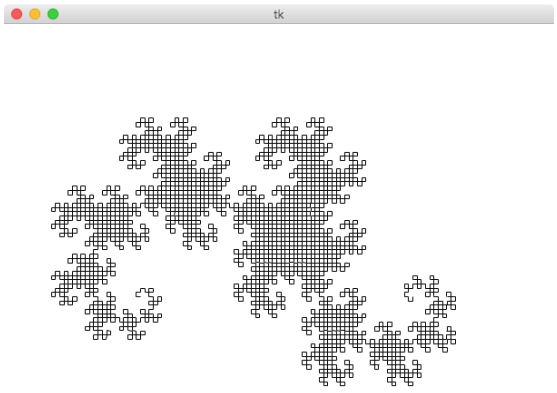

Figure 6.4: ドラゴン曲線

様々なフラクタル図形

例 2「コッホ曲線」や例 3「シェルピンスキー三角形」を参考に，図 6.1 のシェルピンスキー三角形（曲線）や図 6.5 のフラクタル図形などを描くプログラムを作成し，それぞれののハウスドルフ次元を計算してみよ．なお，各ステップごとの 1 辺の長さの変化は，パスカル三角形 (mod 3) で 1/3，パスカル三角形 (mod 4) で 1/4，シェルピンスキーカーペットで 1/3，五角形片で $1/(1+\phi) = 1/\phi^2$ （ただし，ϕ は黄金比で $\phi = (1+\sqrt{5})/2$)，六角形片で 1/3 となっている．

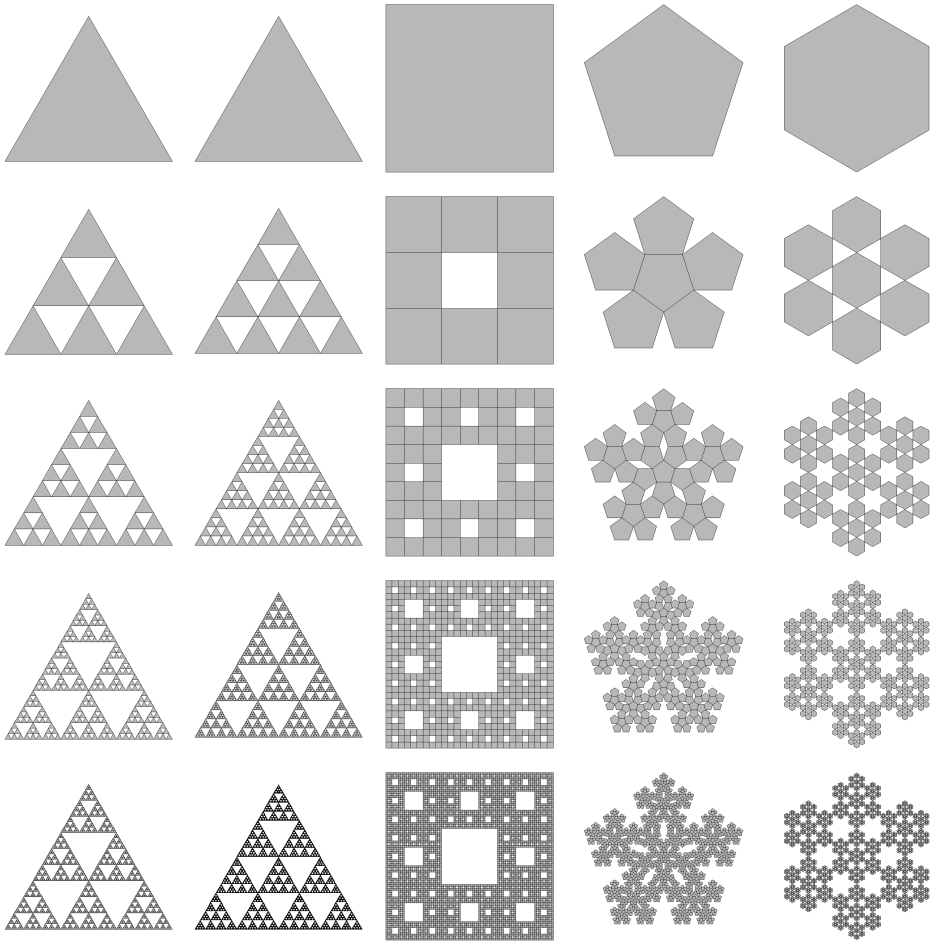

Figure 6.5: パスカル三角形 (mod 3)，パスカル三角形 (mod 4)，シェルピンスキーカーペット，五角形片，六角形片

L システムのプログラム

反復関数系のフラクタルに類似したものに植物の成長モデルなどを記述する L システムがある．図 6.6 では 1 段階ごとに 3/4 倍の縮小と $2\pi/9$ ラジアンの回転を施したセグメントが 2 つ加わることによって，木の成長を模倣している．このような木の成長を表現するプログラムを作成せよ．この場合には，各ステップごとにセグメントが加わっていく（もとのセグメントは残っている）ために，`Fractal` クラスで定義されている 3 引数の `drawFractal` メソッドに相当するものを一部書き換える必要があることに注意せよ．

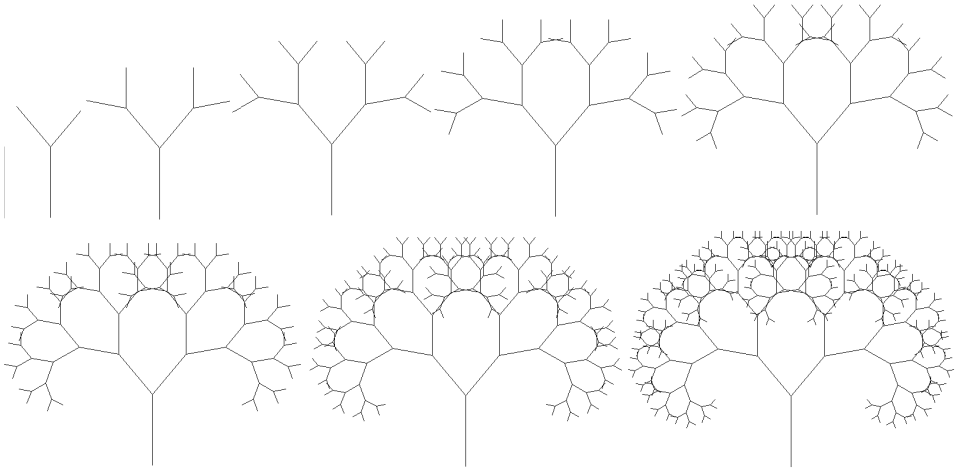

Figure 6.6: L システムによる木の成長