Pythonによる数学入門

並木　誠

本稿では，Python 言語を利用して基礎数学，特に計算を必要とする数学を，自学したり，他者に伝えるための資料を作ったり，習得した数学を実際に計算して確かめたりする場合に有効な方法を，具体的な例を挙げながら解説する．

キーワード：Python 言語，SymPy，NumPy，SciPy.linalg，VPython

1. はじめに

オペレーションズ・リサーチ (OR) という分野で必要とされる数学の多くは，定義，定理，証明が連続しがちな抽象的な数学ではなく，具体的な数値計算を必要とする数学である．OR でのさまざまな表現方法としての数理モデルを，ただ使えればよいブラックボックスとはせずに，その背後にある数学を理解することは，既存のモデルの改善や新たなモデルの構築などに大いに役立つであろう．本稿の目的は，筆者がこれまで見聞きした数学に関する Python 言語の便利な機能を紹介することによって，数学や Python 言語の理解を深めたり，興味関心を引いたりすることである．

本稿の構成は以下のとおりである．2 節では Python で扱える数と演算について述べる．分数や複素数を標準で（追加のパッケージを必要とせず）扱えることを述べる．3 節では，関数の定義とデコレータという付加機能，SymPy という数式処理用のパッケージを利用したプロットや極限を求める例を示す．4 節では，同じく SymPy を用いた微積分について述べる．5 節では，線形代数について，特に NumPy というベクトルや行列を扱うための多次元配列 (`ndarray`) を簡単に説明し，SymPy.linalg の線形代数関連の関数を，例題で取り上げる．6 節では，3D グラフィックスを比較的簡単に取り扱うためのパッケージ VPython を紹介する．

2. 数と演算

私たちが生まれて最初に出会う数学といえばおそらく数えるということだろう．数えるための数を自然数といい，N で表し，以下整数 Z，有理数 Q，実数 R，複素数 C を学んでいく．

表 1　数に関する Python の演算子

演算記号	意味	例
`+`	和	`1.0+2` → `3.0`
`-`	差	`3-5` → `-2`
`*`	積	`2.0*5` → `10.0`
`//`	整数除算	余りがある除算．`9//2` → `4`
`/`	除算	通常の除算．整数同士でも結果小数．`9/2` → `4.5`
`%`	余り	整数除算による余り．`9%2` → `1`
`**`	べき乗	`2**10` → `1024`

Python で扱える数は次のとおりである[1]．分数や複素数も，特別な追加のパッケージを導入することなく，標準で利用できるところは特筆すべきであろう．

Python で扱える数の種類

- 整数 (`int`) 例：`1234`, `-10000` など
- 浮動小数点数 (`float`) 例：`3.1415`, `1.0e-10` (1.0×10^{-10}) など
- 分数 (`Fraction`) 例：`Fraction(1,3)` ($= \frac{1}{3}$) など
- 複素数 (`complex`) 例：`2+3j` ($= 2 + 3i$), `1j` ($= i$) など

これらの数に対して，表 1 のような基本演算が可能である．

数と演算について，いくつかの注意が必要である．まずはじめに，整数についていったいどのくらい大きな（あるいは小さな）数を扱えるのか．時間とメモリーの許す限りである．次のサンプルコードを実行してみよう．計算に数分かかるので注意が必要である．

なみき　まこと
東邦大学理学部情報科学科
〒 274–8510　千葉県船橋市三山 2–2–1
namiki@is.sci.toho-u.ac.jp

[1] バージョン 3.6 以降を想定している．


```
a=3**100000000
```

3 の 1 億乗を変数 a に代入する．ちなみにこれは 47712126 桁の整数である．このように大きい数字も時間に余裕があれば扱える．

次のコードは，分数を扱う例である．

```
from fractions import Fraction
a = Fraction(4,7); print(a)
a = float(a); print(a)
b = Fraction(a); print(b)
b = b.limit_denominator(1000); print(b)
```

分数を扱うには Fraction オブジェクトが定義されているモジュールを最初に読み込む．Fraction の引数に小数を渡すと分数の近似値を返す．上のコードを実行すると次の出力が得られる．

```
4/7
0.5714285714285714
2573485501354569/4503599627370496
4/7
```

limit_denominator は，指定した値より小さな分母となる分数に近似するインスタンスメソッドである．上の例の場合，b=2573485501354569/4503599627370496 を分母が 1000 より小さな値になるような分数で近似している．

次のサンプルコードは複素数に関するものである．虚数単位 i は 1j で表す[2]．

```
import math
print(math.e**(1j * math.pi))
print(1j**1j)
print(math.e**(-math.pi/2))
```

定数 math.pi (π) と math.e（ネイピア数 e）を使うために，最初に math モジュールを読み込んだ．このコードの出力は

```
(-1+1.2246467991473532e-16j)
(0.20787957635076193+0j)
0.20787957635076193
```

であり，オイラーの公式 $e^{i\pi} = -1$ と $i^i = e^{-\pi/2}$ が確かめられた[3]．

3. 関数，プロット，極限

Python で関数を定義する方法は 2 通りある．一つは lambda 式を用いる方法で，もう一つは def 文を使う方法である．ここで lambda 式とは，無名関数を定義するときに使われる式である．def 文は文字どおり関数を定義する文で，return 文で値を返し，定義ブロックでより複雑な処理が可能である．

次のサンプルコードは，よく知られた Fibonacci 数を，自然数 n を引数とする関数として lambda 式と def 文を使ってそれぞれ定義したものである．

```
fib = lambda n: 1 if n==0 or n==1 else fib
    (n-1) + fib(n-2)
def fib2(n):
    return 1 if n==0 or n==1  else fib2(n
        -1) + fib2(n-2)
```

いずれも自分自身を呼び出している再帰的な関数である．マシンにもよるだろうが，どちらの場合も $n = 35$ を超えたあたりで計算時間が増えてきて，瞬時に結果を出力とはいかなくなる．関数の呼び出し回数が指数関数的に増えるからである．

呼び出される関数は，大部分で重複しているというところに目をつけて，一度呼び出された関数値を一時的に保存しておき，二度目以降に呼び出されたときは，新たに計算するのではなく保存した値を参照し，計算の効率化を図ろうとする方法がある．これを**メモ化**という．ナップサック問題の解法の一つとして知られている**動的計画法**に対してもメモ化は有効である．

次のサンプルコードは，Fibonacci 関数でのメモ化を，Python の**デコレータ**と呼ばれる機能で実現した例である．ここでデコレータとは，関数やクラス定義の中身を変更することなく働きを追加したり変更したりすることができる機能のことである．

```
from functools import lru_cache
@lru_cache()
def fib3(n):
    return 1 if n==0 or n==1 else fib3(n
        -1) + fib3(n-2)
```

備え付けのキャッシュの機能を果たすデコレータ lru_cache を使うことにより，関数定義のところは変更することなく，たった 2 行でメモ化機能を付加できるというところがメリットである．

実行時間がどのくらい改善されるか，以下のコードを実行してみよう．

```
%time fib(35)
%time fib2(35)
%time fib3(300)
```

%time は，その行の以下の Python コマンドの実行時間を計測するための IPython の**マジックコマンド**の

[2] 2*1j のように係数が数のとき 2j と省略できる．
[3] 正確には $i^i = e^{-\pi/2+2n\pi}$（n は整数）である．

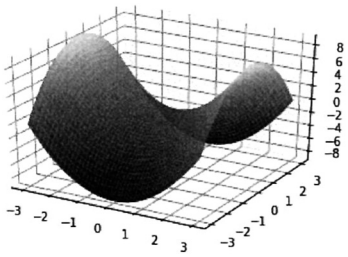

図 1　2 変数関数のプロットの例

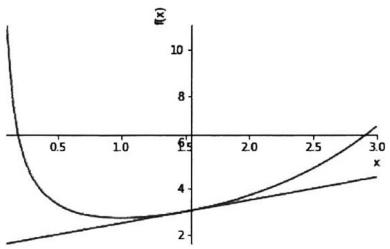

図 2　関数とその接線を同時にプロット

一つである．出力結果の記述は省略する．各自確かめられたい．

定義した関数（1 変数関数）は，座標平面上にプロットすることによって大体の挙動を把握することができる．特に関数が，数学の代表的な関数の組み合わせで定義されているような場合は，**SymPy** パッケージの `plot` が面倒でなくてよい．ここで SymPy とは，Python で数式処理を可能にする追加パッケージである [1]．次のサンプルコードが関数定義とプロットの例である．

```
from sympy import *
x = symbols('x')
f = lambda x: exp(x)/x
plot(f(x), (x,0.1,3))
```

まず最初に SymPy パッケージを読み込み，変数 `x` に`'x'` という名前の記号を紐付けする．関数 `f` を定義し，SymPy の `plot` 関数でプロットする．上のコードの出力は，図 2 の関数のプロットと同じなので省略する．

関数の極限を求めることで，軸付近での関数の振る舞いを正確にみることができる場合がある．次のコードは関数 `f` の極限を求めるためのコードである．

```
print(limit(f(x),x,0))
print(limit(f(x),x,oo))
```

これを実行すると，ともに `oo` を出力する．`oo` は無限大という意味である．関数 `f` のプロットの右端の挙動は，どこかの値に収束するわけではなく，$+\infty$ に発散することがわかった．

2 変数関数のプロットは次のサンプルコードのように `plotting.plot3d` を利用する．これを実行した場合の出力は図 1 となる．

```
x, y = symbols('x y')
plotting.plot3d(x**2-y**2, (x,-3,3), (y
    ,-3,3))
```

4. 微積分

SymPy による数式処理で，微分・積分も楽々可能となる．次のコードは微分の例である．前節での $f(x) = \frac{e^x}{x}$ の導関数を求め，それを利用して接線の方程式を求め，同時にプロットするコードである．

```
x = symbols('x')
f = lambda x: exp(x)/x
def g(x):
    y = symbols('y')
    return diff(f(y), y).subs({y:x})
h = lambda x,a: g(a)*(x-a)+f(a) # 接線
plot(f(x), h(x,3/2), (x,0.1,3.0))
```

このサンプルコードを実行すると図 2 のようなプロットが得られる．

関数 `f(x)` を微分する場合，`diff(f(x),x)` とすればよい．よって `g = lambda x: diff(f(x),x)` としたいところだが，このように定義してしまうと，`g(1)` のように具体的な `x` の値についての微分係数を求めようとしたとき，数値で関数を微分することになり，エラーになるので注意が必要だ．サンプルコードのように，インスタンスメソッドの `subs` で代入するというステップを介さねばならない．

関数 `f(x)` のプロットをみると，極小値をもちそうである．方程式を解いてそのことを確かめる．導関数が 0 となる `x` を求めればよいので次のようなコードになる．

```
solve(g(x),x)
```

出力は `[1]` であり，つまり `x=1` で極小値をとる．

続いて微分の応用として Taylor 展開を考える．$f(x)$ の，x_0 のまわりでの **Taylor 展開**とは，$x = x_0$ の近傍で誤差が小さくなるように $f(x)$ を多項式で表すことである．つまり次の右辺のような多項式を求める．

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots \\ \cdots + a_n(x - x_0)^n + \cdots$$

係数の一般項 a_n は両辺を n 階微分し，$x = x_0$ を代

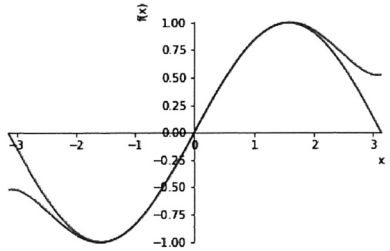

図 3　サイン関数とその原点のまわりでの Taylor 展開

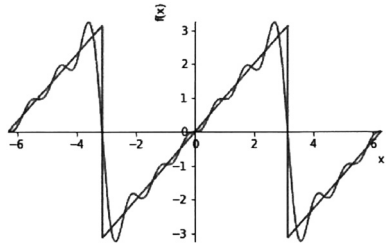

図 4　$f(x) = x$ とその Fourier 級数展開のプロット

入することによって得られる．それらを上の式に代入して得られる Taylor 展開の公式は次のとおりである．

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

これを $f(x)$ の $x = x_0$ のまわりでの Taylor 展開という．

次のサンプルコードは `f(x) = sin(x)` の `x=0` での Taylor 展開を求めるコードである．

```
from sympy import *
x = symbols('x')
f = lambda x: sin(x)
series(f(x),n=6)
```

これを実行すると，

```
x - x**3/6 + x**5/120 + O(x**6)
```

が得られる．`O(x**6)` は剰余項であり，この場合，`x` が 0 に近ければ近いほど，剰余項の部分は前の項よりも早く 0 に近づくという意味である．図 3 は，サイン関数とその Taylor 展開（6 次まで）を同時にプロットしたときのものである．

続いて SymPy を用いた積分について述べる．定積分，不定積分ともに `integrate` 関数を使う．次のサンプルコードは定積分と不定積分の例である．

```
from sympy import *
x = symbols('x')
print(integrate(sin(x)**5, x))
print(integrate(sin(3*x)*sin(3*x), (x,-pi,
    pi)))
```

第一引数に積分する関数を指定する．第二引数として変数のみを指定すれば不定積分，（変数，`a`, `b`）という形式にすれば，区間 `[a,b]` での定積分となる．出力結果は次のとおりである．

```
-cos(x)**5/5 + 2*cos(x)**3/3 - cos(x)
pi
```

積分の応用として，Fourier 級数展開を考える．周期関数 $f(x)$ の **Fourier 級数展開**とは，$f(x)$ を，基本的で扱いやすい周期関数 $\cos nx$, $\sin nx$ $(n = 0, 1, 2, \ldots)$ の線形結合として表すことである．つまり，

$$\begin{aligned} f(x) &= a_0 + a_1 \cos x + a_2 \cos 2x + \cdots \\ &\quad + b_1 \sin x + b_2 \sin 2x + \cdots \\ &= a_0 + \textstyle\sum_{n=1}^{\infty} a_n \cos nx + \sum_{n=1}^{\infty} b_n \sin nx \end{aligned}$$

で表す．各係数 a_0, a_n, b_n $(n = 1, 2, \ldots)$ は，$\cos nx$, $\sin nx$ $(n = 0, 1, 2, \ldots)$ をそれぞれ両辺にかけて周期 $-\pi \leq x \leq \pi$ の範囲で積分することによって

$$\begin{aligned} a_0 &= \tfrac{1}{2\pi} \textstyle\int_{-\pi}^{\pi} f(x)\mathrm{d}x \\ a_n &= \tfrac{1}{\pi} \textstyle\int_{-\pi}^{\pi} \cos nx \cdot f(x)\mathrm{d}x \\ b_n &= \tfrac{1}{\pi} \textstyle\int_{-\pi}^{\pi} \sin nx \cdot f(x)\mathrm{d}x \end{aligned}$$

と計算される．ただし関数 $\cos nx$ や $\sin nx$ $(n = 0, 1, 2, \ldots)$ は，自分自身以外と積をとって区間 `[-pi,pi]` で積分すると 0 となり，自分自身と積をとって区間 `[-pi,pi]` で積分すると π となることを利用している．

次のサンプルコードは，周期 2π の関数 $f(x) = x$ $(-\pi \leq x \leq \pi)$ を Fourier 級数展開して（$n = 6$ まで），もとの関数と同時にプロットしたものである．

```
from sympy import *
x = symbols('x')
f = x - 2*pi*floor((x - pi)/(2*pi)) - 2*pi
g = fourier_series(x, (x,-pi,pi)).truncate
    (n=6)
plot(f, g, (x,-2*pi,2*pi));
```

これを実行すると図 4 の出力が得られる．

SymPy には微分・積分以外にもその他さまざまな機能が備えられている．詳しくはドキュメント [1] を参照されたい．

5. 線形代数

線形代数，特に計算を必要とする線形代数で重要な役割を果たす数学のオブジェクトと言えば，ベクトルと行列である．Python でベクトルや行列を扱うには，

表 2　多次元配列 ndarray の主な属性

属性	内容
ndarray.ndim	配列の次元数
ndarray.shape	配列の型（整数のタプル）
ndarray.T	配列の転置
ndarray.dtype	配列の要素のデータタイプ

NumPy の多次元配列 ndarray を用いる．

多次元配列オブジェクトを生成する方法は，リストやタプルを array() メソッドの引数として渡したり，乱数を使ったり，定形のメソッドを使ったりする方法がある．生成した配列のインスタンスには重要な属性が附随している．表 2 は多次元配列の重要な属性と意味を示している．

入れ子構造が一重の配列の次元は 1 であり，ベクトルを表現するのに用いられる．入れ子構造が二重の配列の次元は 2 であり，行列を表すのに用いられる．

たとえば，次のようにすると x にはベクトル，A には行列が代入される．

```
x = np.array([0,1,2,3]);
print('x =', x); print('x.T =', x.T);
A = np.array([[1,2,3,4],[5,6,7,8]]);
print('A =', A); print('A.T =', A.T);
```

それぞれの属性は，x.ndim = 1, x.shape = (4,), A.ndim = 2, A.shape = (2,4) である．上のコードを実行すると次の出力が得られる．

```
x = [0 1 2 3]
x.T = [0 1 2 3]
A = [[1 2 3 4]
 [5 6 7 8]]
A.T = [[1 5]
 [2 6]
 [3 7]
 [4 8]]
```

ベクトルの転置は，ベクトルのままであることに注意しよう．

a はスカラーとする．ベクトル x の a 倍は a * x によって，行列 A の a 倍は a * A で計算される．型が等しいベクトル同士，行列同士の加減乗除は，成分ごとに行われるので，ベクトル x, y の線形結合は，a*x + b*y　のようにすればよい．ベクトル同士，ベクトルと行列，行列同士の積は，np.dot で求められる．x と y がベクトルならば，np.dot(x,y) は，x と y の内積である．ベクトル x と行列 A の積は，x が m 次ベクトルで A が (m,k) 型のときのみ np.dot(x,A) で計算され，結果は k 次ベクトルとなる．行列 A とベクトル x の積は，A が (m,n) 型で x が n 次ベクトルのときのみ np.dot(A,x) で計算され，結果は m 次ベクトルとなる．行列 A と行列 B の積は，A が (m,n) 型で B が (n,k) 型のときのみ np.dot(A,B) で計算され，結果は (m,k) 型の行列となる．

行列やベクトルに関する演算は，線形代数 (linear algebra) のパッケージとして SciPy の一部に収められている．ここで SciPy とは，最適化，計算幾何学，確率・統計など科学技術計算全般をカバーする優れたパッケージである（文献 [2, 3] 参照）．NumPy の一部として numpy.linalg にも同様のものがあるが，これを理由にどちらを使ったらいいかの混乱があるようだ．SciPy と NumPy を管理している Web ページ [4] では，numpy.linalg には scipy.linalg 以上のものはなく，NumPy パッケージのみ使いたいという場合以外は scipy.linalg を使うよう推奨されている．

ベクトルや行列のみならず任意の次元の ndarray の機能として忘れてはならないのが，**ブロードキャスト**，**ユニバーサル関数**，**ブールインデックス配列**である．同じ型同士の ndarray の演算は成分ごとに行われる．ブロードキャストとは，簡単にいうと ndarray 同士の演算において，型の一致が不完全な場合は，情報を補完して演算を行う機能である．ユニバーサル関数とは，ブロードキャストの機能を，2 項演算に留めず，任意の個数の関数に適用したものである．ブールインデックス配列とは，成分が True または False である配列のことで，その配列によりもともとの配列の部分を切り出すことができる．

これらの NumPy の ndarray に対する機能は，データ処理全般をサポートする Pandas の DataFrame というオブジェクトでも有効であり，Python でデータを扱うには知っていて損はない．

線形代数の応用問題は数多く考えられるが，まず擬似逆行列を使った最小自乗近似曲線を求める例を示そう．ここで，**擬似逆行列** (pseudo-inverse matrix) とは $m \times n$ 行列 A に対して

$$AA^+A = A, \qquad A^+AA^+ = A^+$$
$$(AA^+)^* = AA^+, \quad (A^+A)^* = A^+A$$

を満たす $n \times m$ 行列 A^+ のことである．$b \in R^m$ に対して，$Ax = b$ の解が存在しないとき，A^+b は，$||Ax - b||$ が最小となるベクトル x となることが知られている．Python では A^+ は，scipy.linalg.pinv 関数で求められる．

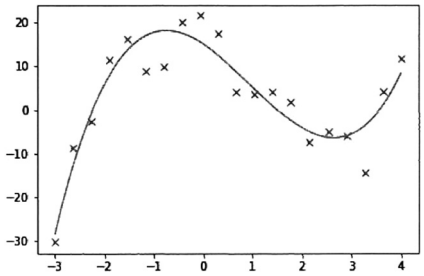

図 5　3 次の最小 2 乗近似曲線

次のサンプルコードは，擬似逆行列を用いてデータの 3 次近似曲線を求める例である．

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as linalg

a3, a2, a1, a0 = 1.0, -3.0, -6.0, 2.0
N = 20
np.random.seed(1234)
xi = np.linspace(-3, 4, N)
yi = a3*xi**3 + a2*xi**2 + a1*xi + a0+N*np
     .random.rand(N)

A = np.vander(xi, 4)
c = np.dot(linalg.pinv(A), yi)

xi2 = np.linspace(-3, 4, 100)
yi2 = c[0]*xi2**3 + c[1]*xi2**2 + c[2]*xi2
      + c[3]
plt.plot(xi, yi, 'x', xi2, yi2)
plt.show()
```

コードは四つの部分からなる．最初の 3 行で必要なパッケージを読み込む．次の 5 行で架空のデータの生成している．Python では

```
a3,a2,a1,a0 = 1.0,-3.0,-6.0,2.0
```

のようにカンマで区切ることによって複数変数への代入が可能である．`np.linspace(-3,4,N)` は，区間 `[-3,4]` を `N` 等間隔に分けた数列を生成する．その次の一文では，`a3*xi**3 + a2*xi**2 + a1*xi + a0` の値に誤差を加えて `yi` に代入している．このように `ndarray` の演算は成分ごとに行われる．

その次の 2 行が近似曲線（3 次）の係数を求める部分である．`np.vander` は，各行が等比数列となっている行列を生成する．3 次の近似曲線の係数は，`c = np.dot(linalg.pinv(A),yi)` で求められる．

最後の 4 行が出力部分である．上のサンプルコードを実行すると，図 5 のようなプロットされたデータの近似多項式（3 次）が得られる．

線形代数の二つ目の応用例として，Fibonacci 数列の一般項を求めてみよう．なお一般項を求めるのに数式処理をするため，使うパッケージは SciPy.linalg ではなく SymPy の中で定義されている関数を使う．

$f(n) = f(n-1) + f(n-2)$ より，連続した二つの Fibonacci 数は次の式を満足する．

$$\begin{aligned}\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix} \\ &\vdots \\ &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}\end{aligned}$$

よって $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ として，A^{n-1} を計算すればよい．A^{n-1} を効率よく計算するには，**固有値，固有ベクトル**を次のように利用する．

A の異なる二つの固有値と固有ベクトル[4]を (λ_1, x^1), (λ_2, x^2) とし，2 次正方行列 P, D をこれらを使って

$$P = \left[x^1, x^2\right], \quad D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

と定義する．P は固有ベクトルを D に対応する順番で並べた正方行列，D はそれぞれの固有値を対角成分にもつ行列である．すると $AP = PD$（$Ax^i = \lambda_i x^i$ を並べたもの）を満たすので，$A = PDP^{-1}$ である[5]．よって $A^{n-1} = (PDP^{-1})^{n-1} = PD^{n-1}P^{-1}$ を得る．D は対角行列なので，

$$D^{n-1} = \begin{bmatrix} \lambda_1^{n-1} & 0 \\ 0 & \lambda_2^{n-1} \end{bmatrix}$$

であり，非常に計算しやすい．

以上のことを Python と SymPy の数式処理を使って忠実に実行するためのコードが次のコードである．SymPy での行列は `Matrix` オブジェクトであること，行列の積は演算子 `*` が使えることに注意したい．

```
from sympy import *
n = symbols('n')

A = Matrix([[1,1],[1,0]])
((ev1,m1,b1),(ev2,m2,b2)) = A.eigenvects()
P = Matrix.hstack(b1[0],b2[0])
D = diag(ev1**(n-1),ev2**(n-1))
B = simplify(P*D*Matrix.inv(P))
```

[4] 常に存在するとは限らない．

[5] 異なる固有値に対する固有ベクトルは線形独立なので，P は正則となる．

```
f=simplify((B * ones(2,1))[0,0])
print(f)
print(simplify(f.subs({n:10})))
```

さらに上のコードの実行結果は次のとおりである．一般項の出力部分は長いので途中省略するが，第 10 項が正しく計算されているところを見ると正解であろう．

```
2**(-n)*sqrt(5)*((1 + sqrt(5))**n + ...
89
```

6. おわりに

本稿の最後に，シミュレーションの定番とも言えるモンテカルロ法による π の推定をやってみよう．(x, y) 平面の $0 \leq x \leq 1, 0 \leq y \leq 1$ の領域にランダムに点を発生させたとき，発生させた点のうち原点からの距離が 1 以下の点の比率は，発生させた点が増えれば増えるほど四分の一円の面積つまり，$\frac{\pi}{4}$ に近くなるだろうということを利用して π を推定する．

これをそのままやっても面白味に欠けるので，3 次元の場合をやってみる．まず π の推定部分の Python コードは次のとおりである．2 次元の場合とほぼ変わらない．近づく値は，球の体積の八分の一つまり $\frac{\pi}{6}$ であることに注意する．

```
import numpy as np

N =3000
np.random.seed(1)
p = np.random.rand(N,3)
pi = np.sum(np.sqrt(p[:,0]**2 + p[:,1]**2+
    p[:,2]**2) <= 1)/N*6
print(pi)
```

π の推定は，`pi = np.sum` ··· の一行のみである．ブロードキャスト，ブールインデックス配列の威力である．出力は `3.158` であった．`N=3000` ではまだまだ不十分か．

シミュレーション結果を可視化してみよう．Python で 3D のオブジェクトを比較的自由に描画したり，アニメーションを作ったりするためのパッケージ VPython を利用する．どのようなことができるかの詳細は Web ページ vpython.org や文献 [5] を参照されたい．

```
from vpython import *
inner = p[np.sqrt(p[:,0]**2 + p[:,1]**2+p
    [:,2]**2) <= 1]
# vpython 初期設定
scene = canvas(width = 800,height = 600)
vmin = -0.1
length = 1.4
scene.up = vector(0,0,1)
scene.forward = vector(-1,-1,-1)
scene.center = vector(0,0,0)
scene.range = 0.9*length
scene.background = color.white
```

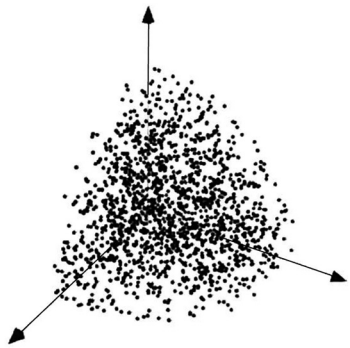

図 6 π の推定

```
cb = color.black
arrow(pos=vector(vmin,0,0),axis=vector(
    length,0,0),
      shaftwidth=0.002,headwidth=0.05,
          color=cb)
arrow(pos=vector(0,vmin,0),axis=vector(0,
    length,0),
      shaftwidth=0.002,headwidth=0.05,
          color=cb)
arrow(pos=vector(0,0,vmin),axis=vector
    (0,0,length),
      shaftwidth=0.002,headwidth=0.05,
          color=cb)

balls = [sphere(pos=vector(*v), radius
    =0.01,color=color.black) for v in
    inner]
```

先のコードに引き続き上のコードを実行すると，図 6 のような球を八等分した第一象限部分に小さな点が集まった図形が得られる．

この図自体は面白みに欠けるが，実際の出力はマウスでドラッグすると視点が簡単に変えられたり，ズームできたりする仕組みになっている．3D アニメーションも簡単に作成できるので，興味がある読者はぜひ試していただきたい．

以上で本稿を終わりにする．少しでも多くの読者が Python を通して数学を楽しんだり，数学を通して Python に興味をもったりするきっかけとなれば幸いである．

参考文献

[1] 2016 SymPy Development Team, “SymPy 1.1.1 documentation,” http://docs.sympy.org/latest/index.html （2018 年 7 月閲覧）

[2] 久保幹雄（監修），並木誠，『Python による数理最適化入門』，朝倉書店，2018.

[3] 久保幹雄，小林和博，斉藤努，並木誠，橋本英樹，『Python 言語によるビジネスアナリティクス—実務家のための最適化・統計解析・機械学習—』，近代科学社，2016.

[4] 2018 SciPy developers, “Numpy and Scipy Documentation,” https://docs.scipy.org/doc/ （2018 年 7 月閲覧）

[5] 上坂吉則，『VPython プログラミング入門』，牧野書店，2011.

