プログラミング基礎演習 1
Taylor 展開による数値解析入門

東京都立大学情報科学科
柴田祐樹

2021 年

1 はじめに

プログラミングを学ぶためには，題材が必要である．文法を覚えるのは機械を扱う上での説明書を読むことに対応し，機械の扱いになれるには実際に使う必要があり，そのとき何かしらの作業内容が必要である．私は脳科学や認知科学の専門家ではないが，題材が必要な理由はこれとほぼ同じことだと思う．プログラミングで初学者がつまずく点と言えば繰り返し処理の書き方であるが，具体的な作業内容よりも級数の計算を行うことが初頭数学の理解のためにも有益で簡単だと考えるため，ここでは特に初頭的で重要である Taylor 級数による無理数の計算方法をプログラムで書く方法を述べる．Taylor 展開について詳しくないもののためにも多少の導入を用意したため，不安であるならば飛ばさず順に読み進めてもらえれば良いだろう．

2 Taylor 級数の課題

計算機で計算可能な演算は四則演算に限られるから，exp, sin などを計算するには，$ax^2 + bx + c$ のような多項式とは違い，工夫を要する．多項式は見ての通り四則演算だけで値が計算可能であるが，exp などは見た限りで全くそうではない．これら特殊な関数に対し，Taylor 級数展開は四則演算により近似値を計算する方法を与えるから，数学により得られた定理を実用可能なものとするために大変重要で必要な理論なのである．

2.1 基本形

ある関数 $f(x)$ を以下の級数で表現する，あるいは近似することを考える．

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots = \sum_{i=0}^{\infty} a_i(x - x_0)^i$$

ただし，$a_1, a_2, ...$ は実数の係数である．単に x の冪乗でなく，起点となる点 x_0 からの差を考えることで，後に分かるとおり点 x_0 の周り置いて高精度に近似が可能となるため，この形式で書き表しておくのが良い．だがこの状態では見通しが悪いため，$h = x - x_0$ を代入した，

$$f(x_0 + h) = a_0 + a_1 h + a_2 h^2 + \cdots = \sum_{i=0}^{\infty} a_i h^i \tag{1}$$

を考えることにする．係数 a は両辺を h で n 回微分した場合を考えると直ぐに求まるので，まずはこの n 階微分を求める．具体的には以下，

$$f^{(n)}(x_0 + h) = n! a_n + (n+1)! a_{n+1} h + \frac{(n+2)!}{2!} a_{n+2} h^2 + \cdots = \sum_{i=n}^{\infty} i! a_i h^{i-n}$$

のようになり，この式に $h = 0$ を代入すれば直ちに，

$$f^{(n)}(x_0) = n! a_n \Rightarrow a_n = \frac{f^{(n)}(x_0)}{n!}$$

と係数が求まる．これを式 (1) へ代入して，よく知られた，

$$f(x_0 + h) = f(x_0) + f'(x_0) h + \frac{f''(x_0)}{2} h^2 + \cdots = \sum_{i=0}^{\infty} \frac{f^{(i)}(x_0)}{i!} h^i \tag{2}$$

で書かれる Taylor 級数展開が得られる．数値計算（プログラムで計算）を行うには有限の次数で級数を打ち切る必要が有るため，このときの誤差の議論には余剰項が重要になってくる．

2.2 級数の実例と収束半径

ここでは式 (2) を直接用い，無理数を近似する方法を幾らか紹介する．

まず，簡単な例は指数関数であり，$f(x) = \mathrm{e}^x$ のとき，任意の $n = 1, 2, ...$ について $f^{(n)} = \mathrm{e}^x$ となり，以下の通り簡単に級数が求まる．

$$\mathrm{e}^{x_0 + h} = \mathrm{e}^{x_0} \left(1 + h + \frac{1}{2} h^2 + \frac{1}{3} h^3 + \cdots \right)$$

ここで，プログラムで実装する場合，結局のところ繰り返しの添字で式を表現しなければならないため，添字と総和で表記するのが望ましい．例えば上記例では，

$$\mathrm{e}^{x_0 + h} = \mathrm{e}^{x_0} \sum_{i=0}^{N} \frac{1}{i!} h^i + R(x_0) \tag{3}$$

となる．これより有限の項数で議論し，誤差についても考えていくため，∞ ではなく N までの和を考える．$R(x_0)$ は有限の項数で打ち切ったことによる近似で生じた誤差を表す[*1]．添

[*1] この誤差のより精確な表現方法として，Lagrange の剰余項などがある．

```
N = 16
e = 1
n = 1
for k in range(N):
    e = e + 1/n
    n *= k
```

図 1　Python3 で式 (4) を解いて Napier 数を求めるコード（コード A0）

字と総和記号で表す一般性の高いこの後者の記法を常に頭で意識できるようになければならない．まずこれができなければ数式を繰り返し文で表現することは無理である．

ところで，この式は，e^{x_0} の値がわかっていることを前提としているが，実際にそのようなことはないため，結局の所自明な値は $e^0 = 1$ だけであることから，$x_0 = 0$ を代入して，

$$e^h = \sum_{i=0}^{N} \frac{1}{i!} h^i + R(0) \tag{4}$$

を用い，e^x のそれぞれの値を求めていくことになる．式 (4) を $h = 1$ として，つまり e の値を実際に求めるコードの例を図 1 に示す．

ここで，途中まで，たとえば e^1 まで求めて，そこから式 (3) を使い e^2 を求めれば，式 (4) から直接 e^2 を求めるよりも実のところ誤差を減らすことができる．もっと極端なことを言えば，$1, e^{0.1}, e^{0.2}, ..., e^2$ のようにより細かく刻んで繰り返し式 (3) を用いれば同じ N を用いてもより精度を高めることができるのだが，その一般化された手法は次の節で差分法として紹介する．

指数関数を近似する際に，式 (3) を繰り返し用いる方法と式 (4) から直接算出する方法を紹介したが，これは指数関数の Taylor 展開の収束半径が ∞[*2]であるために，どちらの方法でも精度はともかくとりあえず求まるのでこれらのちがいについて有用性がわかりづらいだろう．しかし，普通の Taylor 展開では収束半径が ∞ になることは無い．例えば，$f(x) = \ln x$ の場合，

$$\ln(h + x_0) = \ln x_0 + \frac{1}{x_0} h - \frac{1}{2! x_0^2} h^2 + \frac{2}{3! x_0^3} h^3 + \cdots = \ln x_0 + \sum_{i=1}^{N} \frac{(-1)^{i-1}(i-1)!}{i!} h^i + R(x_0) \tag{5}$$

となるが，分母の階乗が打ち消されるため，h の級数のみで収束性を確保しなければならず，等比級数の収束条件と同様に，$|h| < 1$ でなければならない[*3]．なので，$\ln x$ を全て求めるために，$x_0 = 1$ から求めようとしたのではせいぜい $\ln 2$ までしか求められないため，すべての値を求めたい場合は繰り返し式 (5) を適用する方策を取る[*4]．他にも，$\sin x, \cos x, x^{-1}$ など

[*2] 収束半径とは，たとえば式 (4) を適応可能な $|h|$ の範囲である．

[*3] 精確には $-1 < h \leq 1$ である．

[*4] $\ln \frac{1}{x} = -\ln x$ の関係を使えば，直接的な方法だけで求めることも不可能ではないが，ここではこれについて省略する．

の関数が収束条件 1 であり，これらについても繰り返し法が有用となる．

2.3 差分法

前節で収束半径が ∞ でない関数の値を求めるためには，Taylor 展開を繰り返し適用すれば良いと述べたが，この手続きをプログラムで記述する方法と，精度を保つために必要な項数と繰り返し回数について紹介する．まず，それぞれの関数について述べるよりも一般形式 (2) に立ち返り議論を行ったほうが効率的であるため，この式 (2) について考えることとする．いま，我々は x_0 が $f(x_0)$ について自明な値を与えるという前提のもと，$x = x_1, x_2, ..., x_M$ について $f(x)$ の値を順に全て求めることを考える．それぞれの x の値は等間隔にしておいたほうが実装と考察が楽であるため，とりあえず $x_k = kh + x_0, k = 1, 2, ..., M$ とおく．この列に対して，$f(x_k)$ の値がわかっている時に，次の値 $f(x_{k+1})$ すなわち $f(x_k + h)$ の値が知りたいため，これに関する Taylor 展開の定理を用いれば，

$$f(x_{k+1}) = f(x_k + h) = f(x_k) + f'(x_k)h + \frac{f''(x_k)}{2}h^2 + \cdots = f(x_k) + \sum_{i=1}^{N} \frac{f^{(i)}(x_k)}{i!}h^i + R(x_k)$$

の漸化式が得られる．ここで，余剰項 $R(x_k)$ は小さいものと無視して，$N = 1$ くらいで近似した，

$$f(x_{k+1}) = f(x_k) + f'(x_k)h \tag{6}$$

の項を用いても，h が十分に小さければ，$f(x)$ を精度良く求めることができる．計算負荷の観点から言えば，区間 a から b について $f(x)$ の値を求めたいならば，h で均等に分割した時，$M = (b - a)/h$ の点が必要になる（割り切れるとして）ため，精度を高めるためには漸化式を解く回数（繰り返し回数）M を増やす，つまり計算量を増やせば良い．同じ繰り返し回数であっても，精度がもう少し欲しければ，$N = 2$ まで残して，

$$f(x_{k+1}) = f(x_k) + f'(x_k)h + \frac{f'(x_k)}{2}h^2 \tag{7}$$

を用いても良い．ここで，表記の簡素化のために $f(x_k)$ でなく以降は単に f_k と表すこととする．

さて，実際に式 (6) に指数関数を求めるために $f(x) = \mathrm{e}^x$ を代入してみると，$f'(x) = f(x)$ であるので，

$$f_{k+1} = f_k + f_k h \tag{8}$$

が得られる．まずはこの式を Python3 で実装したものが図 2 である．このコードでは $M = 100$, $x_0 = 0$, $x_M = 1$ となるよう設定している．得られた結果は 2.716923932235896 であり，実際の値 2.7182818284590452 に対し 3 桁等しい値が得られている．繰り返し回数（分割精度）が計算に与える影響を見るために，$M = 2, 4, 8, 16, 32, 64, 128$ について計算した結果を表 1 に示す．比較のために式 (6) を用いた場合を 1 次，式 (7) を用いた場合を 2 次として併記している．M を増やせば，また 2 次の式を用いれば精度が向上することが分かる．

```
f = 1
x = 0
M = 100
h = 1/M
for k in range(M):
    f = f + f*h
    x = x + h
print(f) # 2.704813829421526
```

図 2　Python3 で式 (8) を解いて Napier 数を求めるコード（コード A1）

表 1　さまざまな M の値における結果．実際の値は 2.7182818284590452（コード A2E）．

M	1 次	2 次
2	2.2500000000000000	2.6406250000000000
4	2.4414062500000000	2.6948556900024414
8	2.5657845139503479	2.7118412385519850
16	2.6379284973665995	2.7165935224747670
32	2.6769901293781833	2.7178496739802589
64	2.6973449525650999	2.7181725115638300
128	2.7077390196880193	2.7182543383212767
256	2.7129916242534331	2.7182749357407485

```
x = 0
M = 1000
f = [0]*(M+1)
f[0] = 1
h = 1/M
for k in range(M):
    f[k+1] = f[k] + f[k]*h
print(f[M]) # 2.716923932235896
```

図 3　Python3 で式 (8) を解いて Napier 数を求めるコードの List を用いた版（コード A2）

このコードで，もとの式 (6) などには f_{k+1}, f_k なる k の添字が付いていたが，このコードにはそういったものは見当たらない．この理由は次のとおりである．式 (6) のとおりに書くならば，図 3 であるが，f_k の値は，f_{k+1} の計算が終われば二度と使われないため，わざわざ保持しておく必要はなく，一つの値だけ保持するようにして毎回上書きしてしまうように図 2 のように実装したためである．

2.4 差分法の実用例

前節で差分法を紹介し，指数関数の求め方について述べたが，もともと指数関数は収束半径が ∞ なため，差分法を用いる，つまりプログラムを用いる有用性が余り感じられないだろうと思う．そこで，ここでは幾らか実用的で収束性の悪い級数の例を挙げる．

まず，$f(x) = x^{-1}$ について，

$$\begin{aligned} f(x+h) &= x^{-1} - x^{-2}h + x^{-3}h^2 - x^{-4}h^3 + \cdots \\ &= f(x) - (f(x))^2 h + (f(x))^3 h^2 - (f(x))^4 h^3 + \cdots \end{aligned}$$

であるため，この関数を求めるための漸化式は，二次の場合，

$$r_{k+1} = r_k - r_k^2 h + r_k^3 h^2 \tag{9}$$

となる．ここでは後述する他の漸化式と見分けるために関数記号を r にしてある．次に，$\ln x$ であるが，同様の手順により，

$$l_{k+1} = l_k + \frac{1}{x_k}h - \frac{1}{2x_k^2}h^2$$

とできるが，先程 x^{-1} の列については定義したため，これを用いて，

$$l_{k+1} = l_k + r_k h - r_k^2 h^2 \tag{10}$$

と書ける．逆数くらい割り算で求めれば良いと思われるかも知れないが，割り算を電子回路で実装するのは実は結構効率が悪い．これにくらべ，式 (9) は逆数の計算を乗算と加減算のみで表現できるため，効率が高い．割り算は 0 除算による例外が発生することも有るため可能な限り避けたほうが良いと私は思う．式 (9) と式 (10) を実装すると，図 4 のように書くことができる．$x = 1$ から 2 までを 1024 分割して繰り返しを処理したが，得られた値は $1/2 = 0.50000012, \ln 2 = 0.693147181$ であり，真の値 $0.5, 0.693147181$ に近いと言える．

次に，$\sin, \cos$ の例を述べよう．この 2 つの関数は，それぞれ，

$$\begin{aligned} \sin(x+h) &= \sin(x) + \cos(x)h - \frac{\sin(x)}{2}h^2 - \frac{\cos(x)}{6}h^3 + \cdots \\ \cos(x+h) &= \cos(x) - \sin(x)h - \frac{\cos(x)}{2}h^2 + \frac{\sin(x)}{6}h^3 + \cdots \end{aligned}$$

であるから，互いに独立には求められないことがまずわかり，それぞれ s, c として，

$$s_{k+1} = s_k + c_k h - \frac{s_k}{2}h^2 \tag{11}$$

$$c_{k+1} = c_k - s_k h - \frac{c_k}{2}h^2 \tag{12}$$

を漸化式として採用する．この漸化式を用いて実装した例を図 5 に示す．この式は依存関係が s, c の間に有るため，コード中では s2, c2 へそれぞれ値を入れ，古い値が上書きされないようにした後，漸化式の計算をどちらについても完了した後にもとの s, c へ代入してい

```
l = 0
r = 1
x = 1
M = 1024
h = 1/M

for k in range(M):
    l = l + r*h - r*r/2*h*h
    r = r - r*r*h + r*r*r*h*h
    x += h
print(r, l) # 0.5000001193841119 0.6931471805501844; 1/2 = 0.5, ln 2 =
    0.69314718056
```

図 4　Python3 で対数と逆数を求めるコード（コード A3）

```
c = 1
s = 0
x = 1
M = 1024
h = 3.14159265*(2+0.25)/M

for k in range(M):
    s2 = s + c*h - s/2*h*h
    c2 = c - s*h - c/2*h*h
    s = s2
    c = c2
    x += h
print(s, c) # 0.7071466738731728 0.7070672972845574
```

図 5　Python3 で sin, cos を求めるコード（コード A4）

る．$x = 0$ からはじめて，$x = 2\pi + \frac{1}{4}\pi$ まで求めたため，s, c ともに $1/\sqrt{2} = 0.707106781$ となるはずであるが，結果は $s = 0.707146673, c = 0.707067297$ であった．今までの例より誤差が大きいのは区間が長いためだと思われるが，定性的には妥当な値が求められていると言える．

実はこの sin, cos の計算には Leapfrog 法と呼ばれる特殊な方法が存在する．図 6 に示すように，図 5 で用意していた，s2, c2 という一時変数を用意しないで，s (sin) の方を計算したら即座にそれを c (cos) の計算に用いるというものである．この方法は高い安定性を持つ

```
c = 1
s = 0
x = 1
M = 1024
h = 3.14159265*(2+0.25)/M

for k in range(M):
    s = s + c*h
    c = c - s*h
    x += h
print(s, c)
```

図 6　Python3 で sin, cos を求めるコード Leapfrog

ことが知られる．長期間計算し続けるような場合にはこちらを用いると良い．

2.5　積分

前節まで紹介した差分法は積分にも使える．たとえば Gauss 積分,

$$f(x) = \int_0^x \mathrm{e}^{-\frac{t^2}{2}}\mathrm{d}t$$

は解析的に求まらないことで有名であるが，これを Taylor 展開で書けば,

$$f(x+h) = f(x) + \mathrm{e}^{-\frac{x^2}{2}}h - \frac{x}{2}\mathrm{e}^{-\frac{x^2}{2}}h^2 + \frac{x^2-1}{6}\mathrm{e}^{-\frac{x^2}{2}}h^3 + \cdots$$

であるため，さらに $g(x) = \mathrm{e}^{-\frac{x^2}{2}}$ とするならば,

$$f(x+h) = f(x) + g(x)h - \frac{x}{2}g(x)h^2 + \frac{x^2-1}{6}g(x)h^3 + \cdots$$
$$g(x+h) = g(x) - xg(x)h + \frac{x^2-1}{2}g(x)h^2 + \frac{3x-x^3}{6}g(x)h^3 + \cdots$$

となるから，同様に適当な項で和を打ち切って漸化式を立てればプログラムにより上記積分を実行できるはずである．$x = 0$ から初めて $x = 16$ などの十分に大きい x まで漸化式を解いて求めれば $\sqrt{\pi/2}$ に近づくことが確かめられる．ある程度の精度が見たいならば $x \in [0, 10]$ の区間を 1000 分割はしたほうが良い．

2.6　誤差について

Lagrange の剰余項を見ればここまで導出してきた手法で生じる誤差を見積もれる．Lagrange の剰余項によれば，N 次の項まで考慮した場合，

$$R(x) = \frac{f^{(N+1)}(x+\theta h)}{(N+1)!}h^{N+1}$$

と書ける [1]．ただし，$\theta \in (0,1)$ である．これは，分割幅 h に対して，h^{N+1} で誤差が小さくなることを意味している．実際には漸化式を解くたびにこの誤差は蓄積され，繰り返し回数は大体 $\frac{1}{h}$ であるため，h^N の程度の誤差が最終的に残る．これはつまり，$N=2$ のとき，h を半分にすれば 4 倍の精度，$N=3$ のとき，h を半分にすれば 8 倍の精度となることを意味する．

3　まとめ

本書ではプログラミング初学者にむけて Taylor 展開を用いた数値計算の方法を解説した．数値解析入門としてはまったく不十分であるが，繰り返し処理の最初の訓練題材としては参考になるのではないかと思う．誤植があった場合申し訳ないが連絡をいただければありたがい．

参考文献

[1] 第 10 階数学演習，広島大学．https://home.hiroshima-u.ac.jp/kyoshida/MathExercise/2015(1stSemester)/exercise1(no10).pdf (Accessed 2020/10/1)