アルゴリズム入門　＃5

地引　昌弘

2021.11.04

はじめに

前回では、コンピュータを用いた代表的な解析手法の一つである逐次細分最適化法[1] (“少しだけ or 一つだけ実行する”を何度も繰り返し、その結果を集計する) の例として、数値積分および微分方程式の数値的解法を取り上げました。今回は、逐次細分最適化法を用いた数値的計算の精度について考えてみます。

1　前回の演習問題の解説

1.1　演習 4-1a〜4-1c — 数値積分

長方形の高さとして、区間の左端の $f(x)$ を使うと増加関数では値が小さくなり、右端の $f(x)$ を使うと大きくなるので、まずは「左端と右端の平均」を取ってみるという課題でした (減少関数だと逆に大きく/小さくなります)。これは考えてみると、面積を計算するのにその区間の関数を直線で補間した「台形」を考え、その面積を求めているのと同等です。このため、これを数値積分の**台形公式** (Trapezoid Rule) と呼びます。台形公式の計算内容は次のようになります (区間の幅を d で表す):

$$s = \sum \frac{1}{2}\{f(x) + f(x+d)\}d$$

これを計算する Python プログラムを示します:

```
def integ3(a, b, n):
  dx = (b - a)/n; s = 0.0
  for i in range(n):
    x = a + i*(b - a)/n
    y0 = x**2
    y1 = (x + dx)**2
    s = s + 0.5*(y0 + y1)*dx
  return(s)
```

台形公式は直線による補間なので、曲線が上に凸だと値は小さく、下に凸だと値は大きくなります。一方、これも演習にありましたが、区間の中央の x を使った長方形で計算すると (これを**中点公式**と言います)、逆に上に凸だと大きく、下に凸だと小さくなります (図 1)。よって、両者をほど良く混ぜてみよう、というのが演習 4-1c の趣旨でした。

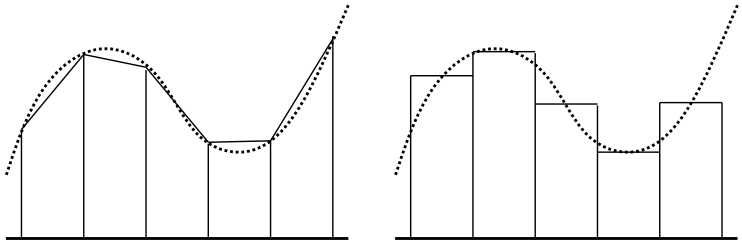

図 1: 台形公式と中点公式

[1] これは一般的な名称ではありません/本講義内だけの名称です。

実は、“左端値による長方形”, “中央値による長方形”, “右端値による長方形” の各面積を 1:4:1 で混ぜると、(言い換えれば、“台形公式による面積”, “中点公式による面積” を 1:2 で混ぜると)、より良い結果が得られます。プログラムも示しておきます:

```
def integ4(a, b, n):
  dx = (b - a)/n; s = 0.0
  for i in range(n):
    x = a + i*(b - a)/n
    y0 = x**2
    y1 = (x + 0.5*dx)**2
    y2 = (x + dx)**2
    s = s + (y0 + 4*y1 + y2)*dx/6.0
  return(s)
```

実際に計算させてみましょう (「正解」は 333 だったことに注意):

```
>>> integ2(1.0, 10.0, 100)
328.5571499999999              ← (前回の) 左端の f(x) を用いて計算した値はこれ
>>> integ3(1.0, 10.0, 100)
333.01215                      ← 台形公式による精度向上は、このくらい
>>> integ4(1.0, 10.0, 100)
332.99999999999994             ← 混合比 1:4:1 は、確かにすごく良い
>>> integ4(1.0, 10.0, 10)      ← 試しに分割数を減らしてみると…
333.0                          ← えっ、さらにぴったり??
```

この計算式は**シンプソンの公式** (Sympson Rule) と言われ、数値積分では標準的な方法です[2]:

$$s = \sum \frac{1}{3}\{f(x) + 4f(x+d) + f(x+2d)\}d$$

何故こちらの方が良いかというと、シンプソンの公式では、当該区間を 2 次曲線で補間することになるからです。よって、積分しようとする関数が 2 次以下の多項式だと「ぴったり」になり、そのため上の例では、区間数が少ないほど誤差が出ないので良くなったわけです (この例では分割数 1 でもぴったりなので、もはや数値積分とは言えないですが…)。

参考までに、2 次式の補間になる理由を示しておきます。当該区間の曲線を以下の 2 次式

$$y = ax^2 + bx + c$$

で表せるものとします。また、区間の幅を $2d$, 左端を x_0, 中央を $x_1 = x_0 + d$, 右端を $x_0 + 2d$, これらに対応する関数値を y_0, y_1, y_2 とします。上の 2 次式の不定積分は、$\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx$ なので、面積 (定積分) は次のようになります:

$$s' = \left[\frac{1}{3}ax^3 + \frac{1}{2}bx^2 + cx\right]_{x_0}^{x_0+2d}$$

これを整理すると次のようになります:

$$3s' = \{a({x_0}^2 + 12x_0d + 8d^2) + b(6x_0 + 6d) + 6c\}d$$

ここで、

$$y_0 = a{x_0}^2 + bx_0 + c$$

$$y_1 = a(x_0 + d)^2 + b(x_0 + d) + c$$

$$y_2 = a(x_0 + 2d)^2 + b(x_0 + 2d) + c$$

[2]この式は、以下の式変形を見やすくするため区間の半分を d とし、プログラムでは 6 で割る代わりに 3 で割っています

を代入すると、次の式が成り立ちます:

$$3s' = (y_0 + 4y_1 + y_2)d$$

$$s' = \frac{1}{3}(y_0 + 4y_1 + y_2)d$$

この式は (つまり s' は)、シンプソンの公式の 1 区間分ですね。

参考: シンプソンの公式の趣旨は、対象が曲線であることから、長方形 · 台形といった直線による近似ではなく、上辺をより曲線に近付けた近似を試みることでした。そこで、扱いやすい曲線として 2 次関数を取り上げ、左端 · 中央 · 右端を通過する 2 次関数を利用したわけです。このような 2 次関数を簡単に求める方法としては、ラグランジュの補間式が存在します。$n+1$ 個の点 $(x_0, y_0), \ldots, (x_i, y_i), \ldots, (x_n, y_n)$ に対するラグランジュの補間式 (つまり、これら $n+1$ 個の点を通る n 次関数) の一般形は下記の式で表されます (是非、付録も読んでみて下さい):

$$L(x) = \sum_{i=0}^{n} f(x_i) \cdot \prod_{0 \le j \le n} \frac{x - x_j}{x_i - x_j} \quad (\text{但し } i \neq j)$$

ここで取り上げたシンプソンの公式を発展させ、さらに数値積分の精度を上げるには、より高次の関数を使うことが考えられます。例えば、3 次の補間式を用いて近似するシンプソンの $\frac{3}{8}$ 公式などが知られています。

ところで、未知の関数 $y = f(x)$ 上にある $n+1$ 個以上の点が与えられた時、これらの点群から $y = f(x)$ を近似的に求めることは、何を意味するのでしょうか。$f(x)$ が n 次式であると分かっていれば、$n+1$ 個の点を用いたラグランジュの補間式より近似関数は求められます。よって以下では、素性が全く分からない $f(x)$ に従うある現象を測定したデータから (ここには当然、誤差が含まれています)、その現象を近似する関数を求める場合を考えます。ここで何の工夫もなければ、データを多く集めることは、かえって誤差のバラつきに振り回された極値が増えることに繋がり、近似関数の凹凸が増えてしまいます。例えば、集めた全てのデータを使ってラグランジュの補間式を計算すると、近似関数は $f(x)$ に比べて超高次になってしまう恐れがあるわけです (特にこのような場合、測定区間外では、$f(x)$ と近似関数との乖離が大きくなります)。データを多く集めて近似精度を上げることが、逆に真実から離れてしまうとは、注意を要する問題ですね。

ところで、他にも数値積分の精度を高める方法はないでしょうか。例えば、台形公式でも次のような方法があります。まずは、ある d で積分を計算しておき、次に d をより小さくして (例えば半分とか) 計算した値と比較します。これを繰り返し、値の変化がなくなったら、これ以上細かく分割しても (d を小さくしても) 意味がないと判断して止めるという方法です。このような計算方法を**漸近法**と言います。

1.2 演習 4-2a〜4-2c — 繰り返し

この辺は簡単なので、プログラムだけ示します (べき乗を計算するだけなら `2**n` でよいのですが、繰り返しの制御構造に慣れることが目的なのでループを使います):

```
def pow2(n):
  result = 1
  for i in range(n):
    result = result*2
  return(result)

def fact(n):
  result = 1
  for i in range(n):
    result = result*(i + 1)
  return(result)
```

階乗については「1 × 2 ×⋯× N」を計算したいわけですが、range 関数が渡して来るカウント値は「0, 1, ⋯, N-1」なので、全て 1 を足してから掛けています。組合せの数を整数で計算するには、「小さい側から」乗算・除算・乗算・除算を繰り返すと都合が良いです。例えば、${}_7C_4$ を $\frac{4\times5\times6\times7}{1\times2\times3\times4}$ のように並べ替え、左から 1 列ずつ乗算・除算の順で計算するわけです。この順序でやれば、除算が必ず割り切れるため[3]、誤差なしで計算できます:

```
def comb(n, r):
  result = 1
  for i in range(r):
    result = result*((i + 1) + (n - r))/(i + 1)
  return(result)
```

2 差分法

前回取り上げた微分方程式の数値的解法ですが、今回はこれをもう少し別の角度から考えてみましょう。

2.1 差分方程式

まずは、元の (or 真の) 関数を $y = F(x)$ とします。この関数に対する微分の定義も含めた常微分方程式は、次のようになります:

$$\frac{dy}{dx} \;=\; \lim_{\Delta x\to 0}\frac{F(x+\Delta x)-F(x)}{\Delta x} \;=\; f(x,y)$$

ここで、これまで見て来た数値的解法の鍵である “少しだけ動かすことで近似する” という考え方に沿って、$\lim_{\Delta x\to 0}$ の代わりに、コンピュータで扱える程度の微小な Δx を想定すると、次のような式が得られます (“≈” は、近似的に等しいの意):

$$\frac{F(x+\Delta x)-F(x)}{\Delta x} \;\approx\; f(x,y)$$

この方程式は、引き算を用いているので、**差分方程式** (Difference Equation) と呼ばれます[4]。差分法は計算がシンプルなので、数値的に解く場合は大変よく使われています。

さて、この差分方程式ですが、まずは次のように式変形してみましょう:

$$F(x+\Delta x) \;=\; F(x)+\Delta x\cdot f(x,y)$$

ここで、$x \to x_i,\ x+\Delta x \to x_{i+1},\ F(x_i) \to y_i$ とします。これらを上の漸化式に代入し、$F(x_i+\Delta x) = F(x_{i+1}) \to y_{i+1}$ とすると、最終的に以下の式が得られます:

$$x_{i+1} = x_i + \Delta x$$
$$y_{i+1} = y_i + \Delta x\cdot f(x_i,y_i)$$

あれ? この漸化式はどこかで見たことがありませんか。そう、前回取り上げたオイラー法ですね。つまり、オイラー法と (1 階) 差分法は同じものと言えます。

---

[3] ${}_{(r+a)}C_r = \frac{(r+a)\times(r+(a-1))\times(r+(a-2))\times\ldots\times(a+1)}{r\times(r-1)\times(r-2)\times\ldots\times1} = \frac{(1+a)\times(2+a)\times(3+a)\times\ldots\times(r+a)}{1\times2\times3\times\ldots\times r}$ とした場合、右辺における左から i 回目までの乗算・除算は ${}_{(i+a)}C_i$ を意味するので、必ず割り切れる理由は分かりますね。

[4] 微分方程式と差分方程式の英語名は、違いがややこしいですね。

2.2 テイラー級数

差分法は、$x + \Delta x$ と x との差分 (つまり多項式) により $F(x)$ を近似する方法です。実は、他にも $F(x)$ を x の多項式で近似できる方法があります。関数 $y = F(x)$ は、$F(x)$ 上の点 $(a,\ F(a))$ における導関数の無限和を用いることで、次のような x の多項式として表わすことができます。ここで、$\mathrm{O}((x-a)^{k+1})$ は $k+1$ 階以降の項をまとめた項です。また、$\frac{dy}{dx} = f(x,\, y)$ より ($\because$ 微分方程式がこの形をしているため)、$F^{(k)}(x) = \frac{d^k y}{dx^k} = f^{(k-1)}(x,\, y)$ です:

$$F(x) = F(a) + F^{(1)}(a)(x-a) + F^{(2)}(a)(x-a)^2 + \cdots + \frac{F^{(k)}(a)}{k!}(x-a)^k + \mathrm{O}((x-a)^{k+1})$$

これを**テイラー級数** (Taylor Series) と呼び、このような (つまり、a に関連した) テイラー級数を求めることを、"a の周りでテイラー展開する" と言います。以下では、差分方程式とテイラー級数との関係を少し調べてみましょう。

2.3 オイラー法の誤差

まずは、オイラー法について考えてみます。オイラー法の漸化式を以下に再掲します:

$$x_{i+1} = x_i + h$$
$$y_{i+1} = y_i + h\frac{dy}{dx} = y_i + hf(x_i,\, y_i)$$

2.1 節で行なった式変形を逆に辿ることで、上の漸化式は、差分方程式を次のように近似計算していることになります:

$$\frac{F(x+\Delta x) - F(x)}{\Delta x} \ = \ f(x,\, y) \tag{1}$$

この式は $F(x)$ に対する 1 階の導関数 ($f(x,\, y)$) しかないので、テイラー級数では 1 次の項までを調べることにします:

$$F(x) = F(a) + F^{(1)}(a)(x-a) + \mathrm{O}((x-a)^2)$$

ここで、$F(x)$ の代わりに $F(x+\Delta x)$ を考え (x を $x+\Delta x$ で置き換え)、これを x の周りでテイラー展開し (a を x で置き換え)、$F^{(1)}(x) = f(x,\, y)$ とすると、以下の式になります (ルンゲ–クッタ法の誤差を求める際も同じ考え方をするので、テイラー展開についてよく理解しておこう):

$$F(x+\Delta x) = F(x) + f(x,\, y)\cdot\Delta x + \mathrm{O}((\Delta x)^2)$$

これより、テイラー展開により得られた解析的な $F(x)$ の差分は、次のようになります[5]:

$$\frac{F(x+\Delta x) - F(x)}{\Delta x} \ = \ f(x,\, y) + \mathrm{O}(\Delta x) \tag{2}$$

式 (1) と式 (2) を比較すると、オイラー法により数値的に計算された差分は、テイラー展開により解析的に得られた差分に比べて、$\mathrm{O}(\Delta x)$ 程度の誤差が存在します。これより、オイラー法の計算精度は、ステップ幅 Δx に比例した精度であることが分かります。

[5] テイラー級数の定義を見ると、$\mathrm{O}((x-a)^{k+1})$ は、$(x-a)$ の $k+1$ 次以上の多項式であることが分かります。よって、$\mathrm{O}((\Delta x)^2)/\Delta x = \mathrm{O}(\Delta x)$ となります。

2.4 ルンゲ–クッタ法の誤差

次に、ルンゲ–クッタ法について考えてみます。ルンゲ–クッタ法の漸化式を以下に再掲します:

$$
\begin{aligned}
&x_{i+1} = x_i + h \\
&k1 = h\frac{dy}{dx} = hf(x_i,\, y_i) && k1:\ \text{始点における傾きを用いて } x \text{ が } h \text{ 増えた時の } y \text{ の増分} \\
&k2 = hf(x_i + h,\, y_i + k1) && (x_i + h,\, y_i + k1):\ \text{仮の終点} \\
& && k2:\ \text{仮の終点における傾きを用いて } x \text{ が } h \text{ 増えた時の } y \text{ の増分} \\
&y_{i+1} = y_i + \tfrac{1}{2}(k1 + k2)
\end{aligned}
$$

この漸化式に、$x_i \to x,\ x_{i+1} \to x + \Delta x,\ y_i \to F(x_i)$ を代入し、$y_{i+1} \to F(x_{i+1}) = F(x_i + \Delta x)$ とすると、以下の式になります:

$$F(x + \Delta x) \ = \ F(x) + \frac{1}{2}\left(\Delta x \cdot f(x,y) + \Delta x \cdot f(x + \Delta x,\ F(x) + \Delta x \cdot f(x,y))\right) \tag{3}$$

上の式では、$f(x + \Delta x,\ F(x) + \Delta x \cdot f(x,y))$ の項が複雑そうに見えますが、改めてその意味を考えてみると、これは $f(x,y)$ において x を Δx だけ増やした時、$F(x)\ (= y)$ が $f(x,y) \cdot \Delta x\ (= \Delta y)$[6] だけ増えたことを表わしています。そこで、点 $(x + \Delta x,\ y + \Delta y)$ における $f(x,y)$ の全微分 $df = f(x + \Delta x,\ y + \Delta y) - f(x,y) = f_x dx + f_y dy$ および、$\frac{dy}{dx} = f(x,y)$ より $dy = f(x,y) \cdot dx$ (この表記については、第 4 回資料の付録を参照) を考えると、$\Delta x \to 0$ とした時、この項は下記の式で表わせます (再度、$\Delta y = f(x,y) \cdot \Delta x$ に注意):

$$df = f(x + \Delta x,\ y + \Delta x \cdot f(x,y)) - f(x,y) = f_x(x,y) \cdot \Delta x + f_y(x,y) \cdot f(x,y) \cdot \Delta x$$

これより、$f(x + \Delta x,\ y + \Delta x \cdot f(x,y))$ を式 (3) に代入すると、以下のようになります:

$$F(x + \Delta x) \ = \ F(x) + \Delta x \cdot f(x,y) + \frac{1}{2}\left\{(\Delta x)^2 \cdot f_x(x,y) + (\Delta x)^2 \cdot f_y(x,y) \cdot f(x,y)\right\}$$

以上より、ルンゲ–クッタ法による差分方程式は、次の近似式を計算していることになります:

$$\frac{F(x + \Delta x) - F(x)}{\Delta x} \ = \ f(x,y) + \frac{1}{2}\,\Delta x\left\{f_x(x,y) + f_y(x,y) \cdot f(x,y)\right\} \tag{4}$$

さて、この式には $F(x)$ に対する 2 階の導関数 ($f_x(x,y)$ および $f_y(x,y)$) が存在するので、テイラー級数では 2 次の項までを調べることにします:

$$F(x) = F(a) + F^{(1)}(a)(x - a) + \frac{1}{2}F^{(2)}(a)(x - a)^2 + \mathrm{O}((x - a)^3) \tag{5}$$

以下、オイラー法の時と同様に、$F(x)$ の代わりに $F(x + \Delta x)$ を考え、x の周りでテイラー展開します。この時、$F^{(1)}(x) = f(x,y)$ より、$F^{(2)}(x)$ は以下のようになります (要は、$f(x,y)$ を全微分するわけです):

$$F^{(2)}(x) = f^{(1)}(x,y) = f_x(x,y) \cdot \Delta x + f_y(x,y) \cdot \Delta y = f_x(x,y) \cdot \Delta x + f_y(x,y) \cdot f(x,y) \cdot \Delta x$$

これを式 (5) に代入すると、下記の式となります:

$$F(x + \Delta x) \ = \ F(x) + f(x,y) \cdot \Delta x + \frac{1}{2}\left\{f_x(x,y) + f_y(x,y) \cdot f(x,y)\right\}(\Delta x)^2 + \mathrm{O}((\Delta x)^3)$$

以上より、テイラー展開により得られた解析的な $F(x)$ の差分は、次のようになります ($\mathrm{O}((\Delta x)^3)/\Delta x = \mathrm{O}((\Delta x)^2)$ に注意/前ページの脚注を参照):

$$\frac{F(x + \Delta x) - F(x)}{\Delta x} \ = \ f(x,y) + \frac{1}{2}\left\{f_x(x,y) + f_y(x,y) \cdot f(x,y)\right\}\Delta x + \mathrm{O}((\Delta x)^2) \tag{6}$$

式 (4) と式 (6) を比較すると、ルンゲ–クッタ法により数値的に計算された差分は、テイラー展開により解析的に得られた差分に比べて、$\mathrm{O}((\Delta x)^2)$ 程度の誤差が存在します。これより、ルンゲ–クッタ法の計算精度は、ステップ幅 Δx の 2 乗に比例した精度であることが分かります[7]。確かに、オイラー法より精度が良いですね。

---

[6] $\frac{dy}{dx} = \frac{dF(x)}{dx} = f(x,y)$ より、$f(x,y) \cdot \Delta x = \lim\left\{\frac{F(x+\Delta x) - F(x)}{\Delta x}\right\} \cdot \Delta x = \lim\{F(x + \Delta x) - F(x)\} = \Delta y$

[7] これが、今回取り上げたルンゲ–クッタ法が、2 次のルンゲ–クッタ法と呼ばれる理由です。

以下、参考までに、ルンゲ–クッタ法を改良して精度を向上させた、**4 次のルンゲ–クッタ法** (4th-order Runge–Kutta Method) を紹介しておきます。これは、ステップ幅 h を半分に分け、次の各点を計算します。

1. 始点での傾きを用いて計算した終点 $k1$
2. 始点での傾きを用いて計算した仮の二分点
3. 仮の二分点における傾きを用いて、再度始点から計算した二分点 $k2$
4. 二分点 $k2$ における傾きを用いて、再度始点から計算した二分点 $k3$
5. 二分点 $k3$ における傾きを用いて、再度始点から計算した終点 $k4$

そして、これら $k1 \cdot k2 \cdot k3 \cdot k4$ の四つを $1:2:2:1$ の比率で混ぜます:

$$x_{i+1} = x_i + h$$
$$k1 = hf(x_i,\ y_i)$$
$$k2 = hf(x_i + \tfrac{h}{2},\ y_i + \tfrac{k1}{2})$$
$$k3 = hf(x_i + \tfrac{h}{2},\ y_i + \tfrac{k2}{2})$$
$$k4 = hf(x_i + h,\ y_i + k3)$$
$$y_{i+1} = y_i + \tfrac{1}{6}(k1 + 2 \cdot k2 + 2 \cdot k3 + k4)$$

比率が $1:2:2:1$ である理由は、テイラー展開における 4 次の項までを当てはめるためですが、この計算は少々複雑なので、その詳細についてはこれ以上深入りしません。但し、オイラー法や 2 次のルンゲ–クッタ法での議論から予想できるように、テイラー級数の 4 次項までを考えるということは、4 次のルンゲ–クッタ法における計算精度は、ステップ幅 h の 4 乗に比例した精度であることが分かります。因みに、誤差を減らすためにステップ幅をさらに細かくすれば良いか (つまり、テイラー級数の $n \gg 4$ 次項までを当てはめるような数値計算法を適用すれば良いか) と言えば、丸め誤差や (h が小さくなることに起因する) 情報落ち誤差/桁落ち誤差も累積するため、ある程度以上は計算精度が上がらなくなります。そのため、実際には 4 次のルンゲ–クッタ法がよく使われているのです。

3 数値計算の安定性

さて今度は、次の微分方程式を数値的に解く場合を考えてみましょう:

$$\frac{dy}{dx} = -3y \quad (\text{差分方程式としては、} F(x+\Delta x) = F(x) + \Delta x \cdot (-3F(x)))$$

この微分方程式も以前と同様、両辺の積分から一般解を求めることができます。得られた一般解より、例えば点 $(0, 1)$ を通る特殊解は、$y = e^{-3x}$ となります。この特殊解のグラフ (の傾向) を表示する Python プログラムは、次のようになります:

```
!pip install ita              # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def euler2(x0, y0, xmax, count):
  h = (xmax-x0) / count
  x = x0; y = y0
  s = ita.array.make1d(count)
  for i in range(count):
    x = x + h
    y = y + h * (-3)*y        # f(x, y) = -3y
    s[i] = y
  ita.plot.plotdata(s, line=True)
  return(y)
```

では、このプログラムを用いて、ステップ数を変えながら、(0, 1) を始点として x 座標が 100 になるまで計算したグラフを見てみましょう (以下のグラフは、あくまで傾向を見るだけなので、グラフにある x 座標の値は実際の x 座標の値と合っていません)。

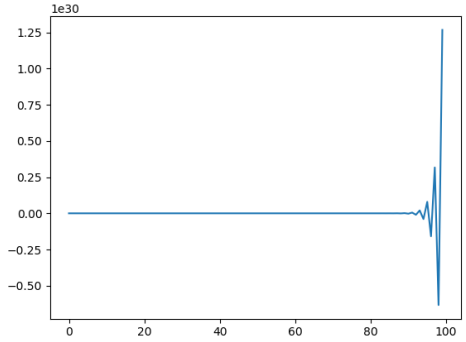

図 2: 差分法: 100 ステップのグラフ

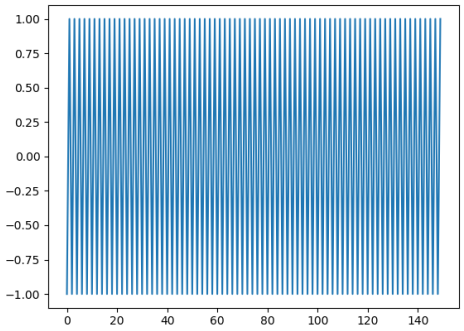

図 3: 差分法: 150 ステップのグラフ

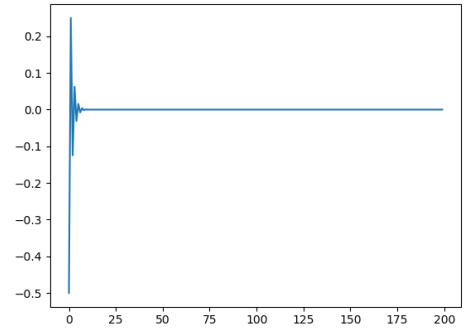

図 4: 差分法: 200 ステップのグラフ

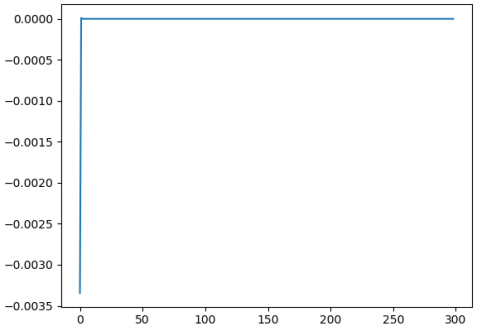

図 5: 差分法: 299 ステップのグラフ

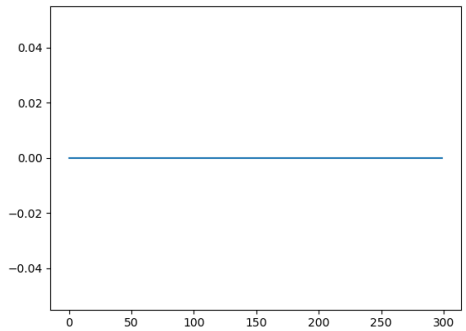

図 6: 差分法: 300 ステップのグラフ

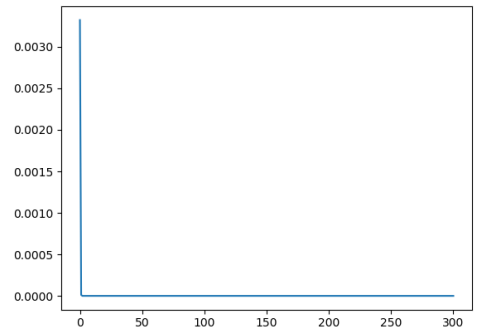

図 7: 差分法: 301 ステップのグラフ

一言で言えば、滅茶苦茶ですね。本当はどんな形をしているのでしょうか。

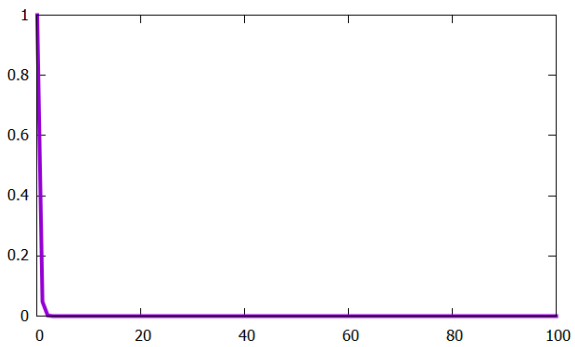

図 8: $y = e^{-3x}$ のグラフ

図 8 を見る限り、ステップ数を 301 まで増やした差分法の近似が一番近いようです。

何故、このようなことが起こるのか、少し考えてみましょう。今回対象とした差分方程式を x_i, y_i で表記すると、$F(x+\Delta x) = F(x) + \Delta x \cdot (-3F(x))$ より、$y_{i+1} = (1-3\Delta x)\, y_i$ となります。ここで、$\frac{2}{3} \leq \Delta x$ となるような Δx を選んでしまうと、$1-3\Delta x$ は -1 以下の値になります。これは、1) y_{i+1} と y_i の値は符号が毎回反転する、2) y_{i+1} の絶対値は常に y_i の絶対値より大きくなる、ことを意味しており、近似曲線が解曲線よりジグザグに離れて行ってしまいます (今回の例では、ステップ数 300 が $\frac{2}{3} \leq \Delta x$ の境界だったわけです)。数値的計算では近似値の精度を保つため、y_{i+1} と y_i の値が大きく違わないように Δx を決める必要があります (是非、付録も読んでみて下さい)。

参考 (誤差と安定性の関係):
オイラー法により得られる各 y_i は、$F(x_i) \approx y_i = y_{i-1} + f(x_{i-1}, y_{i-1}) \cdot \Delta x$ として求められますが、これは Δx に応じた誤差を含む y_{i-1} から、さらに Δx に応じた誤差を加味して y_i が算出されることを意味します。つまり、y_i には、i が増えるにつれ、誤差が蓄積されて行くわけです。この時、各回の誤差がある閾値より大きいと、y_i が $F(x_i)$ (要は真値) よりかけ離れてしまいます (非安定化)。そこで、安定化する (各回の誤差がある閾値に収まる) ように、ステップ幅 Δx を調整する必要があるというわけです。

演習 5-1 x および計算する項の数 n を与えて、次の**テイラー展開** (Taylor Expansion) を計算せよ。

$$\cos x = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

$$\sin x = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

実際に値の分かる x を用いて精度を確認してみること (例えば、$x = \pi/3$ など)。項数 n は、幾つぐらいが適切か? また、$x = \pm 10\pi$ とした場合はどうなるか?

演習 5-2 オイラー法 (差分法) を用いて、以下の数値的計算を行ないなさい。

a. 微分方程式 $dy/dx = -2y$ を対象に、点 (0, 1) を通る近似解曲線を求めるプログラムを作成しなさい。作成後は、様々な値の刻み幅で計算し、上で説明した誤差に関する性質が成り立っているかどうかを確認しなさい (特に、解が安定する刻み幅の前後で試してみること)。

b. ロジスティック方程式 $\frac{d}{dt}x(t) = ax(t) - bx(t)^2$ を対象に (参照: 教科書の練習問題 6.4)、近似解曲線を求める `logi_func(x0, y0, xmax, a, b, count)` 関数を作成しなさい[8]。作成後は、ロジスティック方程式の初期値 $x(0)$ (プログラムでは `y0` に該当) を変えながら、数値計算の結果として得られる近似解曲線の様子を観察しなさい。

演習 5-3 正の整数 N を受け取り、N 以下の素数を全て打ち出すプログラムを作成せよ[9]。次に、素数を全て打ち出すのではなく、発見した素数の個数だけを最後に表示するように改造し、かつ処理に要した時間も表示させよ[10]。

所要時間の表示について: 所要時間を表示する簡単な方法は、処理の開始時と終了時の時刻を取得し、両者の差を表示することです。Python では、下記のようなコードを追加することで、所要時間をを秒単位で取得・表示することができます。

```
import time

def t_test(…):                          ← 所要時間を計りたい処理に応じた引数を用意
  start = time.process_time()           ← 開始時刻の取得
  ........                              ← 所要時間を計りたい処理を、この間に挟む
  finish = time.process_time()          ← 終了時刻の取得
  print("%g" % (finish - start))        ← 両者の差 (所要時間) を秒単位で表示
```

[8] 引数 `x0,`;`,y0,`;`,xmax,`;`,count` の意味は、これまでに作成した euler 関数や rungekutta 関数と同じです。また、引数 `a, b` については、ロジスティック方程式の係数に対応しており、`a, b > 0` です。

[9] N が素数ということは、N を $2 \sim N-1$ のいずれで割っても余りが出るということです。剰余は演算子 "%" で計算できます。

[10] プログラムの実行において、画面に何かを表示するという処理は、計算や比較などの演算に比べて相当な時間を要します。よって、速さを計測するという観点からは、できるだけ表示を省略して内部の処理だけの時間を計った方が良いのです。

付録 1: ラグランジュの補間式について (副題: 意味を考えよう)

以下では、ラグランジュの補間式を導出する方法を題材として、扱っている問題が複雑になり過ぎた場合に、関連する様々な対象が備える意味を考えることで、新しい発想を導く事例について見てみます (もう一つの付録も、同じような観点から書いています)。

$y = f(x)$ 上にある $n+1$ 個の点 $(x_0, y_0), \ldots, (x_i, y_i), \ldots, (x_n, y_n)$ を通る n 次補間式 $L(x)$ を考えます。まずは、議論を始めるにあたり、そもそもこのような $L(x)$ が一意に定まると言えるのかどうか、調べる必要がありそうです。

略証:

同じ $n+1$ 個の点を通る異なる n 次式を $L_1(x)$ および $L_2(x)$ とする。$0 \leq i \leq n$ に対し、$L_1(x_i) = L_2(x_i)$ より $L_1(x_i) - L_2(x_i) = 0$ なので、以下の式が成り立つ (因数定理)。

$$L_1(x) - L_2(x) = a(x - x_n)(x - x_{n-1}) \ldots (x - x_0)$$

この式は、左辺が n 次式、右辺が $n+1$ 次式なので、$a = 0$ となる。よって、$L_1(x) - L_2(x) = 0$ より $L_1(x) = L_2(x)$ が成り立つ。

どうやら $L(x)$ は一意に定まるようなので、次はこれを具体的に求めてみましょう。最初は、一番素直に考え、$L(x)$ を以下の式で表すことにします。

$$L(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \tag{7}$$

式 (7) に $(x_0, y_0), \ldots, (x_i, y_i), \ldots, (x_n, y_n)$ を代入した以下の式を、$A_{n+1} = (a_0, a_1, \ldots, a_n)^t$ の連立一次方程式として考え、その係数 a_i を求めてみましょう。式 (8) は、$Y_{n+1} = (y_0, y_1, \ldots, y_n)^t$ として、この連立一次方程式を行列の式で表したものです。

$$L(x_j) = \sum_{i=0}^{n} a_i x_j^i = f(x_j) = y_j$$

$$Y_{n+1} = V_{n+1} \cdot A_{n+1} \tag{8}$$

この連立一次方程式の係数行列 V_{n+1} は、具体的には次のようになります (これを**ヴァンデルモンド行列** (Vandermonde Matrix) と呼びます)。

$$V_{n+1} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}$$

式 (8) を満たす A_{n+1} を求めるために、ヴァンデルモンド行列 V_{n+1} の行列式 $\det V_{n+1}$ を計算してみましょう。まずは、V_{n+1} の $n+1$ 列目より、n 列目の x_0 倍を引いた行列 $V_{n+1}^{(n)}$ の行列式を考えます (1 行 $\cdot\, n+1$ 列目の成分: $x_0^n - (x_0^{n-1} \cdot x_0) = 0$)。次に、$V_{n+1}^{(n)}$ の n 列目より、$n-1$ 列目の x_0 倍を引いた行列 $V_{n+1}^{(n,n-1)}$ の行列式を考えます。一般に、行列 A のある行 $\cdot$ 列に対して、別の行 $\cdot$ 列の定数倍を加減して得られる行列 A' の行列式 $\det A'$ は、$\det A$ と等しくなります[11]。これより、上の操作を $n = 2$ まで繰り返した $V_{n+1}^{(n \sim 1)}$ の行列式と V_{n+1} の行列式は等しく、下記のようになります。

$$\det V_{n+1} = \det \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} = \det \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & (x_1^2 - x_1 \cdot x_0) & \cdots & (x_1^n - x_1^{n-1} \cdot x_0) \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & (x_n - x_0) & (x_n^2 - x_n \cdot x_0) & \cdots & (x_n^n - x_n^{n-1} \cdot x_0) \end{bmatrix} \tag{9}$$

[11] これは、行列式の多重線形性や交代性を用いることで、簡単に証明できます (線形代数で学びましたよね)。

また、正方行列 A を対称に区分けして、$A = \begin{bmatrix} A_{11} & A_{12} \\ O & A_{22} \end{bmatrix}$ or $\begin{bmatrix} A_{11} & O \\ A_{21} & A_{22} \end{bmatrix}$ とできる場合 (O は全ての成分が 0 の零行列)、その行列式は $\det A_{11} \cdot \det A_{22}$ になります[12]。行列式 (9) にこれを適用すると、次のようになります。

$$\begin{aligned} \det V_{n+1} &= \det[1] \cdot \det \begin{bmatrix} (x_1 - x_0) & (x_1^2 - x_1 \cdot x_0) & \cdots & (x_1^n - x_1^{n-1} \cdot x_0) \\ (x_2 - x_0) & (x_2^2 - x_2 \cdot x_0) & \cdots & (x_2^n - x_1^{n-1} \cdot x_0) \\ \vdots & \vdots & \cdots & \vdots \\ (x_n - x_0) & (x_n^2 - x_n \cdot x_0) & \cdots & (x_n^n - x_n^{n-1} \cdot x_0) \end{bmatrix} \\ &= \det \begin{bmatrix} (x_1 - x_0) & x_1 \cdot (x_1 - x_0) & \cdots & x_1^{n-1} \cdot (x_1 - x_0) \\ (x_2 - x_0) & x_2 \cdot (x_2 - x_0) & \cdots & x_2^{n-1} \cdot (x_2 - x_0) \\ \vdots & \vdots & \cdots & \vdots \\ (x_n - x_0) & x_n \cdot (x_n - x_0) & \cdots & x_n^{n-1} \cdot (x_n - x_0) \end{bmatrix} \end{aligned} \tag{10}$$

さらに、行列 A のある行 or 列に対して、これを α 倍した行列を A' とすると、$\det A = \alpha \cdot \det A'$ が成り立ちます (これも行列式の基本変形としてよく使われます)。これを利用して、行列式 (10) の各行から共通因子を括り出して行くと、次のようになります。

$$\begin{aligned} \det V_{n+1} &= (x_1 - x_0)(x_2 - x_0) \cdots (x_n - x_0) \cdot \det \begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{bmatrix} \\ &= (x_1 - x_0)(x_2 - x_0) \cdots (x_n - x_0) \cdot \det V_n \end{aligned}$$

以上より、$\det V_{n+1}$ の漸化式が得られました。これを帰納的に解くことで、ヴァンデルモンド行列 V_{n+1} の行列式は、最終的に次のようになります。

$$\begin{aligned} \det V_{n+1} &= (x_1 - x_0)(x_2 - x_0) \cdots\cdots\cdots\cdots (x_n - x_0) \times \\ &\quad (x_2 - x_1)(x_3 - x_1) \cdots (x_n - x_1) \times \\ &\qquad \vdots \\ &\quad (x_{n-1} - x_{n-2})(x_n - x_{n-2}) \times \\ &\quad (x_n - x_{n-1}) \\ &= \prod_{0 \le i < j \le n+1} (x_j - x_i) \end{aligned}$$

行列式を計算できたので、式 (8) より、A_{n+1} を以下のように求めてみることにします。

$$A_{n+1} = V_{n+1}^{-1} \cdot Y_{n+1} = \frac{1}{\det V_{n+1}} \cdot \widehat{V}_{n+1} \cdot Y_{n+1} \tag{11}$$

$\widehat{V}_{n+1}$ は V_{n+1} の余因子行列と呼ばれ、少々複雑な行列なので、先人の知見 (これはクラメルの公式と呼ばれています) を使わせてもらい、途中の計算を省略して各成分 a_i の値を書き下してみましょう。これは、次のようになります。

$$a_i = \frac{\det \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{i-1} & y_0 (= f(x_0)) & x_0^{i+1} & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^{i-1} & y_1 (= f(x_1)) & x_1^{i+1} & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \cdots & \vdots & & & & \\ 1 & x_n & x_n^2 & \cdots & x_n^{i-1} & y_n (= f(x_n)) & x_n^{i+1} & \cdots & x_n^n \end{bmatrix}}{\det V_{n+1}} \tag{12}$$

さて、ヴァンデルモンド行列には綺麗な法則性が見られたので、その行列式もうまく変形できましたが、式 (12) はどうでしょうか。分子にある行列式は少々手強そうに見えます (例えば、$i+1$ 列目辺り)。ここまで来て残念ですが、この行列式をさらに追及するよりも、他の方法を考えた方が良さそうに見えます。

[12] この証明は少し面倒ですが、これも行列式の基本変形としてよく使われています。

式 (11) に立ち返り、複雑な余因子行列には踏み込まず (せっかく求めたヴァンデルモンド行列の行列式ですが、取り敢えずこれを脇に置いて)、今度はヴァンデルモンド行列の逆行列が持つ特徴について、少し調べてみましょう。

V_{n+1} の逆行列を V_{n+1}^{-1} とし、V_{n+1}^{-1} の各成分を $v_{i,j}^{(-1)}$ $(0 \leq i, j \leq n)$ で表すと、逆行列の定義より $V_{n+1} \cdot V_{n+1}^{-1} = E$ なので、$v^{(-1)}$ と x の関係は次のようになります。

$$\sum_{0 \leq k \leq n} v_{k,j}^{(-1)} \cdot x_i^k = \delta_{i,j} \tag{13}$$

$\delta_{i,j}$ は、$i = j$ の時に 1 となり、それ以外 ($i \neq j$ の時) には 0 となることを意味する記号です (これは**クロネッカーのデルタ** (Kronecker Delta) と呼ばれ、様々な場面でよく使われています)。ここで、式 (13) を x の式と考え、改めて $L_j(x)$ と置くことにします。

$$L_j(x) = \sum_{0 \leq k \leq n} v_{k,j}^{(-1)} \cdot x^k = \delta_{i,j}$$

$L_j(x)$ は下記の特徴を持ちます。

$$L_j(x_0) = 0, \ldots, L_j(x_{j-1}) = 0, L_j(x_j) = 1, L_j(x_{j+1}) = 0, \ldots, L_j(x_n) = 0$$

$L_j(x)$ は x の多項式 (整式) なので、x_j 以外の x_i を代入して 0 になるということは、$L_j(x)$ が $(x - x_0) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)$ を因子に持つことを意味します。$0 \leq j \leq n$ なので、これらの因子は全部で n 個、つまり $L_j(x)$ は n 次式です。

これは有望そうです。ここで、補間式 $L(x)$ が満たす条件を振り返ってみると、以下の二つでした。

a: n 次式である。
b: $y = f(x)$ 上にある $n + 1$ 個の点 (x_0, y_0), ..., (x_i, y_i), ..., (x_n, y_n) を通る。

条件 a は、上の議論により良さそうな候補が見つかりそうなので、残りは条件 b です。$L_j(x)$ を利用して、$x = x_j$ の時に y_j を返す簡単な関数は作れないでしょうか。簡単に思い付く例として、例えばこんな関数を考えることができます。

$$L'_j(x) = y_j \cdot L_j(x)$$

これより、$L'_j(x)$ をうまく組み合わせることで、上の 2 条件を満たす $L(x)$ を作れそうです。例えば、こんな感じに組み合わせるのが一番簡単そうです。

$$\begin{aligned} L(x) &= L'_0(x) + \ldots + L'_j(x) + \ldots + L'_n(x) \\ &= y_0 \cdot L_0(x) + \ldots + y_j \cdot L_j(x) + \ldots + y_n \cdot L_n(x) \end{aligned}$$

やっと、ここまで辿り着きました。これまでの議論により、$L_j(x)$ は下記の式で表せることが分かっています。

$$L_j(x) = b_j \cdot (x - x_0) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_n)$$

後は、$L_j(x_j) = 1$ となるように、b_j を決めるだけです。b_j については、

$$L_j(x_j) = b_j \cdot (x_j - x_0) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n) = 1$$

より、

$$b_j = \frac{1}{(x_j - x_0) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_n)}$$

となります。よって、$L_i(x)$ は次の式となります (添字の j を i に変えました)。

$$L_i(x) = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)} = \prod_{0 \leq j \leq n} \frac{x - x_j}{x_i - x_j} \quad \text{(但し } i \neq j\text{)}$$

以上より、最終的にラグランジュ補間式の一般形は、下記の式となることが分かりました。

$$L(x) = \sum_{i=0}^{n} y_i \cdot L_i(x) = \sum_{i=0}^{n} f(x_i) \cdot \prod_{0 \leq j \leq n} \frac{x - x_j}{x_i - x_j} \quad (\text{但し } i \neq j)$$

ここでは、ヴァンデルモンド行列の逆行列 $V_{n+1}^{-1} = \frac{1}{\det V_{n+1}} \cdot \tilde{V}_{n+1}$ を直接求めるのではなく、逆行列が持つ特徴をうまく利用することで議論が進みました。参考までに、V_{n+1}^{-1} そのものを効率的に求める手順については、先人の方々にはあまり関心がなかったようで、20 世紀も後半になってから、何とラグランジュの補間式より (ここでの議論を逆に辿って) 求める手順が示されるようになりました。

付録 2: 差分法の安定化について (教科書の補足)

以下では、教科書の第 10 章にある拡散方程式の差分法について、少し補足しておきます (この付録も、扱っている問題が複雑になり過ぎた場合に、関連する様々な対象が備える意味を考えることで新しい発想を導く、という観点から読んでみて下さい)。

拡散方程式の差分法においても近似値の精度を保つには、微小区間 Δ の大きさを制約する必要があります。但し今回は、微小変化させる変数が複数あることが、問題を難しくしています。例えば、(教科書には説明がありませんが) 拡散方程式にある係数 k は、一言で言えば拡散の速さを意味しています。よって直感的には、時間 Δt 内に拡散される範囲 Δx は、$k\Delta t \leq \Delta x$ という条件を満たしている必要がありそうですね。これを見る限り、どうも Δt に制約がありそうですが、これはあくまで直感的な条件であり、数値的解法の精度を高めるには、厳密な条件を求める必要があります。拡散方程式の差分法を安定化させる具体的な条件そのものについては、教科書に記載されていますが、以下ではあくまで参考として、この条件を解析的に求める方針について軽く紹介します。

まずは、初期値を $u(x_0, t_0)$ として近似計算を始め、得られた $u(x_j, t_n)$ について考えます。以下では表記を簡略化するため、$u(x_j, t_n)$ を $u_j^{(n)}$ と表すことにします。教科書にある漸化式より、以下の行列で表した漸化式を導くことができます。

$$\begin{pmatrix} u_1^{(n)} \\ u_2^{(n)} \\ \vdots \\ u_{j-1}^{(n)} \\ u_j^{(n)} \end{pmatrix} = \begin{pmatrix} 1-2c & c & 0 & \cdots & 0 \\ c & 1-2c & c & \cdots & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & \cdots & c & 1-2c & c \\ 0 & \cdots & 0 & c & 1-2c \end{pmatrix} \begin{pmatrix} u_1^{(n-1)} \\ u_2^{(n-1)} \\ \vdots \\ u_{j-1}^{(n-1)} \\ u_j^{(n-1)} \end{pmatrix} + c \begin{pmatrix} u_0^{(n-1)} \\ 0 \\ \vdots \\ 0 \\ u_{j+1}^{(n-1)} \end{pmatrix}$$

簡略化のため、この行列漸化式をさらに $\mathbf{u}_n = A\mathbf{u}_{n-1} + \mathbf{b}_{n-1}$ と表し、下記のように順次解くことで、$u_j^{(n)}$、即ち $u(x_j, t_n)$ について議論することはできます。但し、最終的には行列の固有値を計算する必要があり、その扱いは少々面倒です。

$$\mathbf{u}_n = A\mathbf{u}_{n-1} + \mathbf{b}_{n-1} = A(A\mathbf{u}_{n-2} + \mathbf{b}_{n-2}) + \mathbf{b}_{n-1} = \cdots$$

そこで、全く別の観点から考えることにします。行列や漸化式を用いて帰納的に一つずつ計算するのではなく、$u_j^{(n)}$ がどのような関数で表せるかを、ある意味「いきなり」考えるわけです。但し、どんな関数を仮定してもよいわけではなく、扱い易い関数 (例えば、微積分し易い関数) でなければなりません。しかし、そんな都合の良いものはあるのでしょうか? 実はあったのです。その詳細は脚注[13] を見てもらうこととして、ここでは $u_j^{(n)} = g^n e^{i\cdot k\cdot j\cdot \Delta x}$ と置くことにします[14]。

さてこの時、全ての n に対して $|g|^n \leq C$ (但し、$C \neq 0$) となる条件 (要は、t について何ステップ計算しても、u は発散しないこと) が、差分法を安定化させる条件になります。新たな $u_j^{(n)}$ を見てみると、さらに複雑になったように見えますが、$u_j^{(n)}$ と $u_j^{(n-1)}$ の関係を考えてみると、漸化式における関係、つまり Δ の増減は指数関数において掛け算に変わるので、行列の計算よりも見通しは良くなります。

---

[13] 実は、“あらゆる曲線は、様々な周期的な曲線 (例えば、三角関数) の組み合わせで表現できる” という素晴らしい定理があります。これを「発見」したのは、**ジョゼフ・フーリエ** (Joseph Fourier) です。彼は、2 変数の偏微分方程式である波動方程式が変数分離法により解けることから、(波動方程式は掛け算でしたが) 熱伝導方程式も変数分離解の線形結合で近似できるのでは、と考えました。そこでこれを、周期を変えた三角関数の和として計算してみたところ、うまく近似できたというわけです。このように、あらゆる曲線を様々な周期曲線で表すことを、“フーリエ級数展開” と言います。但し、彼はその証明にまでは至りませんでした。フーリエ級数による展開では、$f(t) = \sum_{n=0}^{\infty}\{a_n\cos(2n\pi t/T) + b_n\sin(2n\pi t/T)\}$ より、a_n および b_n が未知関数 $f(t)$ の積分で表現されます。これが多くの数学者の注目を集め、解析学に留まらず、積分とは何ぞや無限とは何ぞやといった積分論 · 集合論に繋がり、この定理の証明が最終的に完結したのは、フーリエが提唱した一世紀半後でした。

[14] g は、周期曲線の振幅に該当しますが、一般に Δx や Δt を含む関数です。ここで、左辺の n と右辺の n は意味が違います。左辺の n は t_n の添字番号を表しますが、右辺の n は単なる n 乗を表します。また、i は虚数、k は周期曲線の波数 (要は周期) です。一見すると、指数関数のように見えますが、i が虚数なので、実は三角関数の組み合わせになっています。ところで、フーリエ級数展開では、任意の曲線を複数の周期曲線の加算という形で表しますが、上で置いた式には $\sum$ がないですよね。この式は、複数ある周期曲線の一つ (ここでは、分解波と呼ぶことにします) に該当しています。以上より、この式では、n が 1 増えた時に $u_j^{(n)}$ の各分解波が g 倍される場合を考えているわけです。

では、$u_j^{(n)} = g^n e^{i\cdot k\cdot j\cdot \Delta x}$ を教科書にある漸化式に代入してみましょう。

$$g^{n+1}e^{i\cdot k\cdot j\cdot \Delta x} = (1-2c)g^n e^{i\cdot k\cdot j\cdot \Delta x} + cg^n e^{i\cdot k\cdot (j+1)\cdot \Delta x} + cg^n e^{i\cdot k\cdot (j-1)\cdot \Delta x}$$

$u(x,\,t) = u(x_j,\,t_n) = u_j^{(n)}$ より、$u(x+\Delta x,\,t+\Delta t) = u(x_{j+1},\,t_{n+1}) = u_{j+1}^{(n+1)}$ に注意して下さい (同様に、$x-\Delta x$ は x_{j-1}, $t-\Delta t$ は t_{n-1} です)。この式の両辺を $e^{i\cdot k\cdot j\cdot \Delta x}$ で割ると、

$$\begin{aligned} g^{n+1} &= (1-2c)g^n + cg^n e^{i\cdot k\cdot \Delta x} + cg^n e^{-(i\cdot k\cdot \Delta x)} \\ &= g^n\{1-2c(1-cos(k\cdot \Delta x))\} \quad (i \text{ が虚数なので、} e^{i\cdot k\cdot \Delta x} + e^{-(i\cdot k\cdot \Delta x)} = 2cos(k\cdot \Delta x) \text{ です}) \end{aligned}$$

ここで、$D = \{1-2c(1-cos(k\cdot \Delta x))\}$ とすると、上の式は $g^{n+1} = Dg^n$ となります。これは、見慣れた形になりましたね。3 章でも触れたように、少なくとも $|D| \le 1$ でなければ、g^{n+1} の絶対値は常に g^n の絶対値より大きくなってしまい、近似曲線が解曲線よりどんどん離れてしまいます。よって、$0 \le 1-cos(k\cdot \Delta x)) \le 2$ より、$|D| \le 1$ が常に成立するには $c \le \frac{1}{2}$ が必要になる、というわけです。

このように、フーリエ級数展開を利用し、差分法による近似解が安定する条件について調べる方法を、**フォン・ノイマン** (Von Neumann) の安定性解析と呼びます。但し、ノイマンの安定性解析にも制約があります。例えば、ロジスティック方程式のような非線形微分方程式や、定数でない微分項を含む微分方程式 ($g(x,\,y)\cdot dy^2/dx^2 + h(x,\,y)\cdot dy/dx = f(x,\,y)$ 等) には適用できません[15]。上では詳細を端折りましたが、厳密な意味では、実は差分項ではなく差分項の誤差について調べなくてはなりません。つまり、$u_j^{(n)}$ をフーリエ級数展開するのではなく、その誤差項をフーリエ級数展開する必要があるわけです。しかし、上で挙げたような微分方程式では、誤差項が満たす方程式も複雑になるため[16]、その後の計算は見通しが良くならないのです。

[15] この他、変数が三つ以上であったり、変数が二つであっても両変数の差分項が三つ以上ある場合 (拡散方程式では、x の差分項は三つありますが、t の差分項は二つですね) には、適用できません。しかし、実際の物理現象に基づく微分方程式の多くは、これらの制約条件から外れており、ノイマンの安定性解析を適用することができます。

[16] 例えば、差分方程式を厳密に満たす解を $u_j^{(n)}$ とし、実際の数値計算で得られた解を $U_j^{(n)}$ とすると、誤差は $\epsilon_j^{(n)} = u_j^{(n)} - U_j^{(n)}$ と表せます。これより、$u_j^{(n)}$ を差分方程式 $u_j^{(n+1)} = F(u_j^{(n)},\, u_{j\pm1}^{(n)},\, \ldots)$ に代入すると、$U_j^{(n+1)} + \epsilon_j^{(n+1)} = F(U_j^{(n)} + \epsilon_j^{(n)},\, U_{j\pm1}^{(n)} + \epsilon_{j\pm1}^{(n)},\, \ldots)$ が得られます。この時、差分方程式が線形であれば、$U_j^{(n+1)} + \epsilon_j^{(n+1)} = F(U_j^{(n)},\, U_{j\pm1}^{(n)},\, \ldots) + F(\epsilon_j^{(n)},\, \epsilon_{j\pm1}^{(n)},\, \ldots)$ と変形できます。数値計算より得られる $U_j^{(n)}$ が満たす差分方程式は、$U_j^{(n+1)} \approx F(U_j^{(n)},\, U_{j\pm1}^{(n)},\, \ldots)$ と表せるので (計算精度が上がるほど "≈" →"=" となる)、これを代入すると $\epsilon_j^{(n)}$ だけの差分方程式になります (これは $u_j^{(n)}$ の満たす差分方程式と同じ形になりますね/ここがミソなのです)。しかし、差分方程式が非線形の場合は、このような都合の良い変形ができません。