常微分方程式の初期値問題の数値解法

桂田 祐史

1994 年 6 月〜2011 年 5 月 21 日, 2022 年 6 月 19 日

(なかなか整理するための時間が取れない。我ながらひどい出来だと思うのだが…正直に白状すると、私は良く理解できていない。

この文書と紛らわしい「常微分方程式の初期値問題の数値解法入門」 [1] という題の文書も用意した。そちらは古い授業資料を元にしていて、やさしい微分方程式の問題をとにかく解いてみよう、という内容である。)

目 次

1 序 **2**

2 常微分方程式の初期値問題の復習 **3**
- 2.1 数学理論 . . . 3
- 2.2 簡単な数値解法 . . . 4
  - 2.2.1 前進 Euler 法 (forward Euler's rule) . . . 5
  - 2.2.2 後退 Euler 法 (backward Euler's rule) . . . 5
  - 2.2.3 Runge-Kutta 法 . . . 5

3 基本的な概念・用語 **5**
- 3.1 多段法, 段数 . . . 6
- 3.2 局所離散化誤差、公式の次数 . . . 6
- 3.3 前進 Euler 法の収束証明 . . . 8

4 典型的なスキーム (1) Runge-Kutta 法とその一族 **9**
- 4.1 歴史 . . . 9
- 4.2 定義と Stetter の行列表現 . . . 10
- 4.3 収束定理 . . . 11
- 4.4 特徴 . . . 11
- 4.5 次数と段数 . . . 11
- 4.6 前進型公式の考察 . . . 11
  - 4.6.1 1 段 1 次 . . . 11
  - 4.6.2 2 段 2 次 . . . 12
  - 4.6.3 3 段 3 次 . . . 12
  - 4.6.4 4 段 4 次 . . . 12
- 4.7 埋め込み型の公式, RKF45 . . . 13
  - 4.7.1 RKF45 公式 . . . 13
  - 4.7.2 RKF45 による刻み幅の自動調節 (書き直し版、工事中) . . . 14

4.7.3 実験例: 爆発する問題を RKF45 で計算 . . . 15
4.7.4 RKF45 自作プログラム開発 . . . 18

5 典型的なスキーム (2) 線形多段法 **21**
5.1 PC (予測子修正子法) . . . 21

6 その他の方法 **22**
6.1 Taylor 法 . . . 22
6.2 補外法 . . . 22

7 数値的安定性 **23**
7.1 線形安定性解析 . . . 23

8 Stiff problem (硬い問題) **25**

9 参考書 **26**

10 おまけ — 実際的な誤差の推測 **27**

A 数値計算するための情報 **27**
A.1 はじめに . . . 27
A.2 C 言語+グラフィックス・ライブラリィ . . . 28
A.3 C 言語+gnuplot . . . 28
A.4 Java . . . 28
A.5 C++ の利点 . . . 29
A.6 Ruby . . . 29

B 眠りから覚めてみたら (2022/6) **30**

1 序

この文書では、1 階正規形の常微分方程式の初期値問題に対する数値解法を扱う。すなわち

$$\frac{dx}{dt} = f(t,x) \quad (t \in I) \tag{1}$$

$$x(t_0) = x_0 \tag{2}$$

を満たす $x = x(t)$ の近似解を求めることを考える。ここで I は $t_0 \in \mathbf{R}$ を含む $\mathbf{R}$ の区間で、

$$\begin{cases} f : \mathbf{R}^{n+1} \supset \Omega(\text{開集合}) \to \mathbf{R}^n \quad \text{連続}, \\ (t_0, x_0) \in \Omega \end{cases}$$

は与えられているとする。

注意 1.1 高階の方程式も 1 階にできることが多いので、1 階正規形の方程式は十分一般的であると考えられる。■

微分方程式の解は関数であり、これは関数空間の要素としてとらえるのが (現代の数学では) 普通である。(大抵の場合、関数空間は無限次元空間で、問題を難しくしている。)

微分方程式は解析的[1]に解けないことが多い。たとえ解けても便利でないことがある[2]。
近似解法では、解の有限的な近似表現を求める。具体的には、

- 「有限次元の関数空間の要素で近似する」が基本。
- 特に連続変数を離散変数に置き換えて近似する離散変数法が有力。

残念ながら

常微分方程式の初期値問題に限っても万能の方法はない。

プロでない平均的ユーザーとしては、実際的にはとりあえず **Runge-Kutta** 法を使い[3]、不満があれば他の方法を考える、くらいで良いだろう。
この講義では、基礎概念を簡単に説明した後で、

1. 刻み幅の自動調節 (**adaptive stepsize control**)
2. 硬い方程式 (**stiff problem**)

などの話題を紹介する。詳しいことを知りたい場合は、まず三井 [2] を見るとよい。

最近 (2004 年 2 月)、面白い本が出版された。三井・小藤・齊藤 [3] である。第 2 章「ハミルトン系の解法」, 第 3 章「遅延微分方程式の解法」, 第 2 章「確率微分方程式の解法」と章の名前を見れば一目瞭然、現在盛んに研究されている分野への入門ができる (ヒットだと思う)。

(「最近」が 7 年前か…) 定番本の翻訳が出た。ハイラー・ネルセット・ヴァンナー [4] とハイラー・ヴァンナー [5] である。現時点での決定版か。

2 常微分方程式の初期値問題の復習

2.1 数学理論

1 時間くらいでさらっと説明するための説明は、桂田・佐藤 [6] に書いた。

- 局所解の存在を保証するには、f の連続性を仮定するだけで十分[4]。
- 解の一意性を保証するには f の連続性だけでは不十分。

例 2.1 (無限個の解の分岐) 次の常微分方程式の初期値問題には無限個の解が存在する。

$$\begin{cases} x' = x^{1/2} & (t > 0) \\ x(0) = 0. \end{cases}$$

実際 $\forall t_0 \geq 0$ に対して

$$x(t) \stackrel{\text{def.}}{=} \begin{cases} 0 & (t \leq t_0) \\ \dfrac{(t-t_0)^2}{4} & (t > t_0) \end{cases}$$

は解である。

[1]「解析的に解く」とは、不定積分を取る、四則演算を施す、逆関数を取る、初等関数に代入するなどで求めたり — いわゆる求積法で解いたり — 、解を級数で表現したりすることを指す。

[2]例えば無限級数で解を表したとして、その値を計算するのはそんなに簡単ではない。無限級数は有限的な表現とは言えない。

[3]実は実際に色々解いてみて、最近考え方が変ってきた。やはり最低限 RKF45 くらいは使いたい。しっかり解説を書くべきだと思う。

[4]ただしこれは有限次元の場合で、無限次元の場合は連続性だけではダメである (反例がある)。連続の場合の存在証明はいわゆるコンパクト性の議論を用いるので、空間の次元が有限次元であることは本質的な仮定となってしまうのは自然である。一方、「$f = f(t,x)$ が連続かつ x につき局所 Lipschitz 条件を満たすならば、局所解が一意的に存在する」という定理は無限次元でもそのまま成り立つ。

- f が次に示す "変数 x に関する局所 **Lipschitz** 条件" を満たせば一意性が成り立つ。$\forall(t_0, x_0) \in \Omega$, $\exists V : (t_0, x_0)$ の近傍, $\exists L_V > 0$ s.t.

$$\|f(t, x_1) - f(t, x_2)\| \le L_V \|x_1 - x_2\| \quad ((t, x_1), (t, x_2) \in V).$$

  f がこの条件を満たすための分かりやすい十分条件として、f が C^1 級であることがあげられる。すなわち

$$f \in C^1(\Omega) \Longrightarrow f : \text{局所 Lipschitz}.$$

- 大域的な存在について。$(t, x(t)) \to \partial\Omega$ または $\|x(t)\| \to \infty$ となるまで左右に延長できる (延長不能解の存在定理)。

  このうち、$(t, x(t))$ が f の定義域 Ω の境界に近付くという条件は分かりやすいが、$\|x(t)\| \to \infty$ の方は見慣れない人もいるかも知れない。これについては次の例を見るとよい。

  例 **2.2** (爆発解)

$$\begin{cases} x' &= x^2 \\ x(0) &= 1 \end{cases}$$

  の解は

$$x(t) = \frac{1}{1-t} \quad (t \in (-\infty, 1))$$

  であり、

$$\lim_{t \uparrow 1} x(t) = \infty. \blacksquare$$

- 次の一様 Lipshitz 条件を満たせば、爆発しない。

$$\exists L > 0 \quad \forall (t, x_1), (t, x_2) \in \Omega \quad \|f(t, x_1) - f(t, x_2)\| \le L \|x_2 - x_1\|.$$

2.2 簡単な数値解法

(コンピューター・プログラミングの演習で、必ずと言って良いほど出会うポピュラーな数値解法を復習しよう。)

$I = [a, b]$ とする。$N \in \mathbf{N}$ に対し、I を N 個の小区間に分ける:

$$a = t_0 < t_1 < t_2 < \cdots < t_N = b.$$

このとき、各 t_j における x の値 $x(t_j)$ の近似値 x_j を求めることを考える方法を離散変数法 (**discrete variable method**) と呼ぶ。$h_j := t_{j+1} - t_j$ $(j = 0, 1, \cdots, N-1)$ を刻み幅と呼ぶ。

区間の分割の仕方としては、例えば

$$h_j \equiv h \quad \text{i.e.} \quad h = \frac{b-a}{N}, \quad t_j = a + jh \quad (j = 0, 1, \cdots, N).$$

のように等分割することが多い。以下この小節ではそれを仮定して説明するが、可変刻み幅に一般化することは容易である。

2.2.1　前進 Euler 法 (forward Euler's rule)

微分係数 $x'(t)$ を前進差分商

$$\frac{x(t+h)-x(t)}{h}$$

で近似して作った漸化式

$$x_{j+1}=x_j+hf(t_j,x_j)\quad (j=0,1,2,\cdots)$$

で $\{x_j\}_{j=0}^N$ を計算する。

数学のどこにでも出て来る巨人 Leonhard Euler (1707–1783, スイスの Basel に生まれ、ロシアの St Petersburg にて没する) による。

2.2.2　後退 Euler 法 (backward Euler's rule)

前進差分商のかわりに後退差分商

$$\frac{x(t+h)-x(t)}{h}$$

で近似して得られる漸化式

$$x_{j+1}=x_j+hf(t_{j+1},x_{j+1})\quad (j=0,1,2,\cdots)$$

で $\{x_j\}_{j=0}^N$ を計算する。

x_{j+1} を求めるために、方程式を解く必要がある (f によっては計算が面倒になることもある)。こういう方法を一般に陰解法 (implicit method) と呼び、前進 Euler 法のように方程式を解かずに、求める値が直接計算できる方法を一般に陽解法 (explicit method) と呼ぶ。

2.2.3　Runge-Kutta 法

漸化式

$$\begin{cases} k_1=hf(t_j,x_j)\\ k_2=hf(t_j+h/2,x_j+k_1/2)\\ k_3=hf(t_j+h/2,x_j+k_2/2)\\ k_4=hf(t_j+h,x_j+k_3)\end{cases}$$

$$x_{j+1}=x_j+\frac{1}{6}\left(k_1+2k_2+2k_3+k_4\right)$$

で $\{x_j\}_{j=0}^N$ を計算する方法を (古典的、あるいは 4 次の) **Runge-Kutta 法**と呼ぶ。

3　基本的な概念・用語

なるべく一般的な解法で説明しよう。簡単のため、刻み幅は一定である ($h_j\equiv h$) とする。

3.1 多段法, 段数

k 段法とは x_{j+k} を定めるために

$$x_{j+k} = a_0 x_j + a_1 x_{j+1} + \cdots + a_{k-1} x_{j+k-1} + h\Phi(t_j, t_{j+1}, \cdots, t_{j+k}, x_j, x_{j+1}, \cdots, x_{j+k}) \equiv L(t_j, t_{j+1}, \cdots, t_{j+k}, x_j, x_{j+1}, \cdots, x_{j+k}, h) \tag{3}$$

のように x_j, x_{j+1}, $\cdots$, x_{j+k} を含んだ方程式を用いる数値解法のことである。

このような方程式のことをスキーム (**scheme**) と呼ぶ。

ここで $a_0, \cdots, a_{k-1}$ は $\displaystyle\sum_{i=0}^{k-1} a_i = 1$ を満たす定数であり、Φ は f によって定まる、微分・積分などの無限小演算を含まない写像で、$f \equiv 0$ ならば $\Phi \equiv 0$ となるものである。この $(k, \{a_i\}_{i=0}^{k-1}, \Phi)$ が一つの方法を特徴づける。

問　Euler 法、Runge-Kutta 法がこの形になっていることを確かめよ。

- 上の整数 k をスキームの段数 (**step number**) と呼ぶ。
- Euler 法、Runge-Kutta 法では、段数 $k = 1$ である。このときは

  $$L(t_j, t_{j+1}, x_j, x_{j+1}, h) = x_j + h\Phi(t_j, t_{j+1}, x_j, x_{j+1})$$

  という形、つまり

  $$x_{j+1} = x_j + h\Phi(t_j, t_{j+1}, x_j, x_{j+1})$$

  というスキームになる。
- $k \geq 2$ なるスキームを多段法 (**multistep method**) と呼ぶ。
- Φ が x_{j+k} によらないように表される時、陽解法 (**explicit method**) であると呼び、そうでない場合を陰解法 (**implicit method**) と呼ぶ。陰解法では x_{j+k} を求めるために、(一般には非線型の) 方程式を解かねばならないので、ほとんどの場合に反復法が必要になり、面倒であるが、次数が高くて安定性のよい方法が作れる。

3.2 局所離散化誤差、公式の次数

解法 (3) の、t における局所離散化誤差 (**local truncation error**) を

$$\tau(t,h) := \frac{1}{h}\left[x(t+kh) - L(t, t+h, \cdots, t+kh, x(t), x(t+h), \cdots, x(t+kh))\right]$$

で、また大域的離散化誤差 (**global discretization error**, **global truncation error**) を

$$\tau(h) := \max_{t \in [a, b-kh]} |\tau(t,h)|$$

で定義する。

これを用いて公式の次数 (**order**, 位数とも呼ぶ) を次のように定義する。

公式の次数が少なくとも m $\overset{\text{def.}}{\Leftrightarrow}$ C^m 級の一意解を持つ任意の初期値問題に適用した場合 $\tau(h) = O(h^m) \quad (h \to 0)$.

例 3.1 (ポピュラーな公式について) 前進 Euler 法, 後退 Euler 法は共に 1 段 1 次の公式であり、古典的 Runge-Kutta 法は 4 段 4 次の公式である。■

注意 3.2 f があまり滑らかでないときなど、解がなめらかでない場合、次数 m の公式を用いても $\tau(h) = O(h^m)$ は期待できない。■

あらく言って m 次の公式とは、Taylor 展開して考えたとき、m 次の項まで一致するものであって、次のような性質を持つ:

(i) (どういうわけか運良く) 第 j ステップまで誤差なく計算出来たとすると

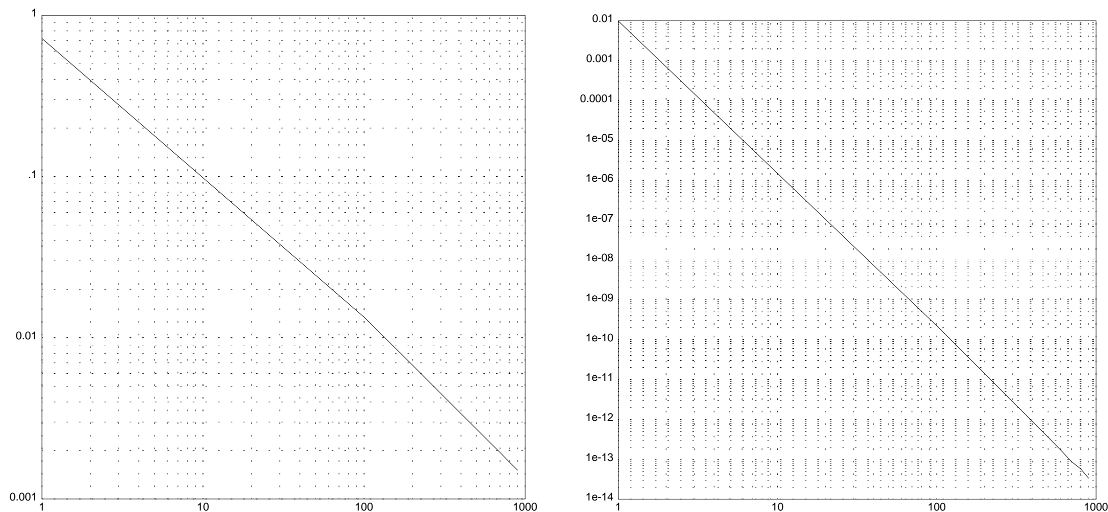
$$x(t_{j+1}) - x_{j+1} = O(h^{m+1}) \quad (h \to 0).$$

(ii) 実際は誤差が累積するので

$$x(t_N) - x_N = O(h^m) \quad (h \to 0).$$

この左辺を全離散化誤差 (**total discretization error**) と呼ぶ。

例 3.3 Euler 法は 1 次、古典的 Runge-Kutta 法は 4 次の公式である。そこで例えば $x'(t) = x$ $(t \in (0,1))$, $x(0) = 1$ という初期値問題に適用した場合の累積誤差を表示したものが次の図である (横軸は区間の分割数 N, 縦軸は累積打ち切り誤差で、いずれも対数目盛)。



(左が Euler 法によるもの、右が Runge-Kutta 法によるもので、傾きがそれぞれ -1, -4 であることが分かる。)

実はこの問題の場合、Euler 法では $x_{n+1} = (1+h)x_n$, Runge-Kutta 法では $x_{n+1} = (1+h+h^2/2+h^3/3!+h^4/4!)\,x_n$ となる。$x(t+h) = e^h x(t) = (1+h+h^2/2+h^3/3!+h^4/4!+\cdots+h^n/n!+\cdots)\,x(t)$ であるから、Euler 法は 1 次の項まで、Runge-Kutta 法は 4 次の項まであっていると言える。■

収束のための条件

以下の三条件が成り立つならば、$h \to 0$ とするとき、近似解が真の解に収束することが証明できる。

(i) 公式の次数が少なくとも 1 以上 (適合条件 (**consistency**) を満たす)。

(ii) Φ が $x_j, x_{j+1}, \cdots, x_{j+k}$ について Lipschitz 条件を満たす。

(iii) 初期値 $x_1, x_2, \cdots, x_{k-1}$ が適切に用意される。

3.3 前進 Euler 法の収束証明

もっとも簡単な前進 Euler 法の場合に近似解が厳密解に収束することを証明してみよう。

定理 3.4 (Euler 法の収束) 常微分方程式の初期値問題

$$\frac{dx}{dt} = f(t,x) \quad (t \in (a,b)), \quad x(a) = x_0$$

において、f は連続で、Lipschitz 条件

$$\|f(t,x_1) - f(t,x_2)\| \le L\|x_1 - x_2\|$$

を満たし、C^2 級の解が存在すると仮定するとき、Euler 法による近似解は微分方程式の解に収束する。

x が C^2 級であるから、$\forall h\ \exists \theta \in (0,1)$ s.t.

$$x(t+h) = x(t) + x'(t)h + \frac{1}{2}x''(t+\theta h)h^2.$$

$x'(t) = f(t,x(t))$ であることに注意して、$t = t_n$, $t + h = t_{n+1}$ とおくと、

$$x(t_{n+1}) = x(t_n) + f(t_n, x(t_n))h + \frac{1}{2}x''(t_n + \theta h)h^2.$$

Euler 法の公式より

$$x_{n+1} = x_n + f(t_n, x_n)h$$

であるから、辺々引き算して

$$x(t_{n+1}) - x_{n+1} = x(t_n) - x_n + \left(f(t_n, x(t_n)) - f(t_n, x_n)\right)h + \frac{1}{2}x''(t_n + \theta h)h^2. \tag{4}$$

これから、もしも $x(t_n) = x_n$ であったとすると、

$$x(t_{n+1}) - x_{n+1} = \frac{1}{2}x''(t_n + \theta h)h^2$$

を得る。これから Euler 法の局所離散化誤差は $O(h)$ である (ゆえに Euler 法の次数は 1 である) ことが分かる。

さて、x が C^2 級であることから

$$M := \max_{s \in [a,b]} \|x''(s)\|$$

が存在する。

(4) から

$$\begin{aligned}
\|x(t_{n+1}) - x_{n+1}\| &\le \|x(t_n) - x_n\| + \|f(t_n, x(t_n)) - f(t_n, x_n)\| \cdot |h| + \frac{1}{2}\|x''(t_n + \theta h)\| h^2 \\
&\le \|x(t_n) - x_n\| + L\|x(t_n) - x_n\| \cdot |h| + \frac{1}{2} \cdot Mh^2 \\
&= (1 + L|h|)\|x(t_n) - x_n\| + \frac{1}{2}Mh^2.
\end{aligned}$$

ここで

$$e_n := \|x(t_n) - x_n\|$$

とおくと、

$$e_{n+1} \le (1+L|h|)e_n + \frac{1}{2}Mh^2.$$

$e_0 = 0$, $n|h| \le b-a$ に注意して、以下の補題 3.5, 3.6 を用いると

$$\begin{aligned} e_n &\le \frac{(1+L|h|)^n - 1}{L|h|} \cdot \frac{1}{2}Mh^2 = \frac{M|h|}{2L}\left[(1+L|h|)^n - 1\right] \\ &\le \frac{M|h|}{2L}(e^{nL|h|} - 1) \le \frac{M|h|}{2L}(e^{L(b-a)} - 1). \end{aligned}$$

補題 3.5 数列 $\{e_n\}_{n\ge 0}$ が、実定数 A, B (ただし $A \ge 0$) に対して漸化不等式

$$e_{n+1} \le Ae_n + B \quad (n \ge 0)$$

を満たすならば

$$e_n \le A^n e_0 + (A^{n-1} + A^{n-2} + \cdots + A^2 + A + 1)B.$$

特に $A \ne 1$ ならば

$$e_n \le A^n e_0 + \frac{A^n - 1}{A - 1}B.$$

証明　順に代入していくだけである。

$$\begin{aligned} &e_1 \le Ae_0 + B, \\ &e_2 \le Ae_1 + B \le A(Ae_0 + B) + B = A^2 e_0 + (A+1)B, \\ &e_3 \le Ae_2 + B \le A(A^2 e_0 + (A+1)B) + B = A^3 e_0 + (A^2 + A + 1)B \end{aligned}$$

より明らかに

$$e_n \le A^n e_0 + (A^{n-1} + A^{n-2} + \cdots + A + 1)B. \blacksquare$$

(なんか、Gronwall の数列版か？)

補題 3.6 $x > 0$ に対して

$$(1+x)^{1/x} < e.$$

証明　$f(x) = \log(1+x)$ について、テイラーの定理より $\forall x > 0$, $\exists \theta \in (0,1)$ s.t.

$$\log(1+x) = f(x) = f(0) + f'(0)x + \frac{1}{2}f''(\theta x)x^2 = 0 + 1 \cdot x + \frac{1}{2} \cdot \frac{-1}{(1+\theta x)^2}x^2 < x.$$

ゆえに

$$\log(1+x)^{1/x} = \frac{\log(1+x)}{x} < 1.$$

これから $(1+x)^{1/x} < e$ を得る。 ■

4　典型的なスキーム (1) Runge-Kutta 法とその一族

4.1　歴史

以下の説明は、一松 [7] による。

Carl Runge (1856–1927, ドイツの Bremen に生まれ、Göttingen にて没する) が 4 次の Runge-Kutta 公式を導いた (1895)。Heun (1900) と Martin Wilhelm Kutta (1867–1944, Upper Silesia (現在のポーランド) に生まれ、ドイツの Fürstenfeldbruck にて没する) (1901) は一般的に[5]研究した。1940 年代の Gill の研究もあったが (**Runge-Kutta-Gill** の公式はかつては有名だった)、公式の本格的な再検討が始まったのは、Ceschino, Butcher, 田中正次等の 1960 年頃からの研究による。

4.2 定義と Stetter の行列表現

前節に解説したタイプの公式のうち、$k=1$ の場合、すなわち 1 段法を **Runge-Kutta** 型公式という。いくつかの (t,x) について、右辺の $f(t,x)$ を計算し、それらの重みつき平均によって、適当な次数の (=その次数までの真の解の Taylor 展開と一致するような) 公式を作っている。具体的には Runge-Kutta 型公式の一般形は

$$\begin{cases} x_{j+1} & = x_j + h\displaystyle\sum_{i=1}^{s}\mu_i k_i \\ k_i & = f\left(t_i+\alpha_i h, x_j + h\displaystyle\sum_{\ell=1}^{s}\beta_{i\ell}k_\ell\right) \quad (i=1,\cdots,s) \end{cases}$$

の形に書くことが出来る。ここで s を段数 (**number of stages**) と呼ぶ。段数とは、要するに 1 ステップ先に進めるために必要な f の計算回数である[6]。

上の式は、k_1, k_2, $\cdots$, k_s についての s 元連立 1 次方程式を解いて、その重みつき平均を x_j に足して x_{j+1} を求める、という手順で使うことになる。

段数 s と係数 $\{\alpha_i; 1\le i\le s\}$, $\{\beta_{ij}; 1\le i,j\le s\}$, $\{\mu_i; 1\le i\le s\}$ を選ぶとスキームが定まることになる。それら係数を並べた

$$\begin{array}{c|ccc} \alpha_1 & \beta_{11} & \cdots & \beta_{1s} \\ \alpha_2 & \beta_{21} & \cdots & \beta_{2s} \\ \vdots & \vdots & \cdots & \vdots \\ \alpha_s & \beta_{s1} & \cdots & \beta_{ss} \\ & \mu_1 & \cdots & \mu_s \end{array}$$

のような表を **Stetter** の行列表現 (**Stetter's notation**) と呼ぶ[7]。

公式が前進型 (陽的、**explicit**) であるとは

$$i \le j \quad \Longrightarrow \quad \beta_{ij}=0$$

が成り立つことを言う[8]。このとき、k_1, k_2, $\cdots$, k_s の順に容易に計算できる。

公式が前進型でない場合、公式が陰的 (**implicit**) であるという。陰的な場合でも

$$i < j \quad \Longrightarrow \quad \beta_{ij}=0$$

が成り立つ[9]場合は半陰的 (**semi-implicit**) であるという。

[5]例えば、4 段 4 次の公式にはどういうものがありうるかとか、5 段 5 次の公式は存在しないようだとか。

[6]1 段法なのに「段数」とは？ 英語では step, stage と区別があるのだが、日本語では同じ「段」となってしまって紛らわしい。残念ながら、この用語はもう定着してしまっている。

[7]多分 $\alpha_i=\displaystyle\sum_{\ell=1}^{s}\beta_{i\ell}$, $\displaystyle\sum_{i=1}^{s}\mu_i=1$ のような条件があるのだと思うが…

[8]行列 (β_{ij}) の対角線の上側と対角線上にある成分がすべて 0 であること。つまり (β_{ij}) が狭義の下三角行列であるという条件である。

[9](β_{ij}) の対角線の上側にある成分がすべて 0 (対角成分自身は 0 でなくてもよい) ということ。つまり (β_{ij}) が狭義下三角行列であるという条件である。

4.3 収束定理

(工事中)

定理 4.1 (Runge-Kutta 型公式の収束のための条件) Runge-Kutta 型公式が収束するためには次の二条件が成立することが必要十分。

(i) 適合性 ($\rho(1) = 0,\ \rho'(1) = \sum_{j=0}^{s} \beta_j$)

(ii) 安定

4.4 特徴

Runge-Kutta 型公式の特徴として、次のものがあげられる。

(1) 自己出発的 (self-starting) である。すなわち、多段法 ($k \geq 2$) では計算の最初に必要となる x_1, x_2, $\cdots$, x_{k-1} を準備することなく、計算が開始できる。

(2) 計算の途中で刻み幅 (stepsize) h の変更が簡単である ($\to$ adaptive stepsize control に便利)。

4.5 次数と段数

次数を大きくしようとすると、普通は f の値の計算回数 (段数, stage) が増えることになる。

公式の範囲を限定して、段数 s を与えたとき、可能な最高の次数 m を、到達可能次数 (到達可能位数) と呼ぶ (Butcher による)。

m の下からの評価は、少なくとも一つの m 次公式を作ることによって得られるが、m の上からの評価は s 元連立代数方程式の解の不存在証明なので、s が大きくなると急激に困難になる。

陽的 Runge-Kutta 型公式の場合には、$s \leq 8$ までの場合に到達可能次数が分かっているという (一松 [7][10])。

到達可能次数

段数 s	1	2	3	4	6	7	9	10
到達可能次数 m	1	2	3	4	5	6	7	8

ここで $m \geq 5$ のとき $s > m$ となることに注意しよう (この事実が 4 次の Runge-Kutta 法が人気のある理由の一つである)。

なお陰的 Runge-Kutta 型公式では、s 段で $2s$ 次を到達する公式が存在することが分かっている。

4.6 前進型公式の考察

4.6.1 1 段 1 次

$s = 1$ の場合は、公式は前進 Euler 法だけしかない。

[10] 今ではこの本も古くなったので、もっと大きな s まで分かっているかも知れない。また別の文献に書いてあることと矛盾していたような覚えがある。

4.6.2　2 段 2 次

$s=2, q=2$ の公式は **Heun** 法と総称される。

$$\beta_{11}=\beta_{12}=\beta_{22}=0,\quad \mu_1+\mu_2=1,\quad \beta_{21}\mu_2=\frac{1}{2}$$

となる。$\beta_{21}=:\beta$ とおくと、

$$\mu_1=1-\frac{1}{2\beta},\quad \mu_2=\frac{1}{2\beta}$$

となり、

$$k_1=hf(t_i,x_j),\quad k_2=f\left(t_i+\beta h,x_j+\beta k_1\right),\quad x_{j+1}=x_j+\left(1-\frac{1}{2\beta}\right)k_1+\frac{1}{2\beta}k_2.$$

特に $\beta=\frac{1}{2}$ のとき、

$$k_2=hf\left(t_i+\frac{h}{2},x_j+\frac{k_1}{2}\right),\quad x_{j+1}=x_j+k_2\qquad \text{(中点公式, 全体を改良 Euler 法).}$$

また $\beta=1$ のとき、

$$k_2=hf(t_i+h,x_j+k_1),\quad x_{j+1}=x_j+\frac{1}{2}(k_1+k_2)\qquad \text{(台形公式, 全体を修正 Euler 法).}$$

4.6.3　3 段 3 次

(省略する。一松 [7] を見よ。)

4.6.4　4 段 4 次

以下は Kutta (1901) の研究だそうである (一松 [7] からの孫引き)。
前提条件

$$\beta_{11}=\beta_{12}=\beta_{13}=\beta_{14}=\beta_{22}=\beta_{23}=\beta_{24}=\beta_{33}=\beta_{34}=\beta_{44}=0$$

のもとで、パラメーターを

$$\beta_{21}=:\alpha,\quad \beta_{32}=:\lambda,\quad \beta_{31}+\beta_{32}=\beta,\quad \beta_{42}=:\nu,\quad \beta_{43}=:\mu,\quad \beta_{41}+\beta_{42}+\beta_{43}=:\gamma$$

とおくと、次の **Kutta** の条件式をえる。

$$\begin{cases}\mu_1+\mu_2+\mu_3+\mu_4=1, & \alpha\mu_2+\beta\mu_3+\gamma\mu_4=\frac{1}{2},\\ \alpha^2\mu_2+\beta^2\mu_3+\gamma^2\mu_4=\frac{1}{3}, & \alpha^3\mu_2+\beta^3\mu_3+\gamma^3\mu_4=\frac{1}{4},\\ \alpha\lambda\mu_3+(\alpha\nu+\beta\mu)\mu_4=\frac{1}{6}, & \alpha\beta\lambda\mu_3+(\alpha\nu+\beta\mu)\gamma\mu_4=\frac{1}{8},\\ \alpha^2\lambda\mu_3+(\alpha^2\nu+\beta^2\mu)\mu_4=\frac{1}{12}, & \alpha\lambda\mu\mu_4=\frac{1}{24}.\end{cases}$$

この一般解も知られていて (自由度 2 が残る)、そのうちすべての係数が ≥ 0 で、$0<\alpha\le\beta\le\gamma\le 1$ である「単調な」公式は、次の古典的 Runge-Kutta 公式 (Runge の原公式, 本来の Runge-Kutta 公式) しかない。

本来の **Runge-Kutta** 公式

Kutta のパラメーターで、

$$\alpha = \lambda = \beta = \frac{1}{2}, \quad \nu = 0, \quad \mu = 1, \quad \gamma = 1,$$

すなわち Stetter の行列表現で

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{pmatrix},$$

スキームで

$$\begin{aligned} k_1 &= hf(t_i, x_j), \\ k_2 &= hf(t_i + h/2, x_j + k_1/2), \\ k_3 &= hf(t_i + h/2, x_j + k_2/2), \\ k_4 &= hf(t_i + h, x_j + k_3), \\ x_{j+1} &= x_j + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

4.7 埋め込み型の公式, RKF45

4.7.1 RKF45 公式

E. Fehlberg は次の公式 RKF45[11] を提案した (1970, [8])。

(5)
$$x_{j+1} = x_j + h\left(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\right),$$

ただし

$$\begin{aligned} k_1 &= f(t_j, x_j) \\ k_2 &= f\left(t_j + \frac{1}{4}h, x_j + \frac{1}{4}hk_1\right) \\ k_3 &= f\left(t_j + \frac{3}{8}h, x_j + \frac{1}{32}h(3k_1 + 9k_2)\right) \\ k_4 &= f\left(t_j + \frac{12}{13}h, x_j + \frac{1}{2197}h(1932k_1 - 7200k_2 + 7296k_3)\right) \\ k_5 &= f\left(t_j + h, x_j + h\left(\frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right)\right) \\ k_6 &= f\left(t_j + \frac{1}{2}h, x_j + h\left(-\frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right)\right). \end{aligned}$$

[11] この公式は御覧の通り大変複雑である。筆者はこの公式で誤植のドジを犯したことがある (レポート課題のプリントで間違えた — 罪重…)。この公式を用いたプログラムを書く時は複数の資料でチェックすることをお勧めする。なお、オリジナルは E.Fehlberg; “Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wrmeleitungs-probleme”, Computing, Vol.6, pp.61–71 (1970).

これは 6 段 5 次の公式であるが、それだけでなく

(6) $$x^*_{j+1} = x_j + h\left(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\right)$$

という値を作ると、x^*_{j+1} は $x(t_{j+1})$ に対して 4 次の近似値となる。この次数の差を利用してステップ幅の自動調節をする方法を以下に説明する。

このアイディアのキーは、s 段 m 次公式に、f の値を計算することなく $(m-1)$ 次公式を付随させるところにある。このような Runge-Kutta 型公式を、$(m-1)$ 次公式が m 次公式に埋め込まれているといい、埋め込み型 **Runge-Kutta** 法と呼ぶ。

4.7.2 RKF45 による刻み幅の自動調節 (書き直し版、工事中)

$t=a$ から $t=b$ まで、誤差の大きさの見積が、計算者が指定した値 ε より小さくなるように解くことを目標にする。このとき ε のことを許容誤差限界と呼ぶ。$[a,b]$ を $a=t_0<t_1<\cdots<t_N=b$ と分割して解くことにする。$t=t_j$ における値 x_j まで定まったとする。t_{j+1} における 5次公式, 4 次公式による近似値をそれぞれ x_{j+1}, x^*_{j+1} とすると、

$$x_{j+1} - x(t_{j+1}) = ch^6 + O(h^7), \quad x^*_{j+1} - x(t_{j+1}) = c'h^5 + O(h^6)$$

となる。ただし $h := t_{j+1} - t_j$ とおいた。これから

$$x_{j+1} - x^*_{j+1} = -c'h^5 + O(h^6).$$

通常は、x_{j+1} は x^*_{j+1} よりも格段に精度が良いと期待出来るので、この式の値が x^*_{j+1} の誤差の良い評価となっていると考えられる:

$$x_{j+1} - x^*_{j+1} = (x_{j+1} - x(t_{j+1})) + \left(x(t_{j+1}) - x^*_{j+1}\right) \fallingdotseq x(t_{j+1}) - x^*_{j+1}.$$

さて、

$$\delta_{j+1} := \left\|x_{j+1} - x^*_{j+1}\right\|$$

と置いておく。

目標は $[a,b]$ で解いたときの許容誤差限界を ε とすることであるから、$[t_j, t_{j+1}]$ で解いたときの許容誤差限界は

$$\varepsilon \cdot \frac{h}{H}, \quad H := b - a$$

とするのが妥当であろう。それゆえ

$$\delta_{j+1} \le \frac{\varepsilon h}{H}$$

であれば良いが、そうでない場合は、h が大きすぎて十分な精度が得られていないと考え、もっと h を小さくすることにする。h の代りに、$\overline{h} \ll h$ なる $\overline{h}$ を用いて

$$\overline{t}_{j+1} = t_j + \overline{h}$$

とおき、これに対応した $\overline{x}_{j+1}$, $\overline{x}^*_{j+1}$ を求め、$\overline{\delta}_{j+1} := \left\|\overline{x}_{j+1} - \overline{x}^*_{j+1}\right\|$ とおく。

$$\delta_{j+1} \sim \|c'\|\,|h|^5, \quad \overline{\delta}_{j+1} \sim \|c'\|\,\left|\overline{h}\right|^5$$

となると期待できるから、

$$\overline{\delta}_{j+1} \sim \left(\frac{|\overline{h}|}{|h|}\right)^5 \delta_{j+1}.$$

今度は $\overline{\delta}_{j+1} \le \varepsilon \overline{h}/H$ となって欲しいわけだが、この不等式の左辺に推定値を代入した不等式

$$\left(\frac{|\overline{h}|}{|h|}\right)^5 \delta_{j+1} \le \frac{\varepsilon \overline{h}}{H}$$

を $\overline{h}$ について解くと

$$\overline{h} \le h\left(\frac{\varepsilon h}{H\delta_{j+1}}\right)^{1/4}$$

を得る。安全のために

$$\overline{h} := 0.9h\left(\frac{\varepsilon h}{H\delta_{j+1}}\right)^{1/4}$$

で $\overline{h}$ を定めることにする。

4.7.3 実験例: 爆発する問題を RKF45 で計算

(かなりいい加減なプログラムであるが、あえてここに含める理由の一つは、公式中の係数に書き間違いがないことを確認できるようにするためである。)

```
/*
 * rkf.c --- RKF45 のサンプル (まだ未完成です)
 */

#include <math.h>

#define RKF45_STAGE 6

static double rkf45_alpha[RKF45_STAGE] = {
  0.0, 1.0/4, 3.0/8, 12.0/13, 1.0, 1.0/2};

static double rkf45_beta[RKF45_STAGE][RKF45_STAGE-1] = {
  {   0.0,         0.0,                     0.0,        0.0},
  {   1.0/4,       0.0,                     0.0,        0.0},
  {   3.0/32,      9.0/32,                  0.0,        0.0},
  {1932.0/2197, -7200.0/2197, 7296.0/2197,    0.0,        0.0},
  { 439.0/216,     -8.0,      3680.0/513,  -845.0/4104,   0.0},
  {  -8.0/27,       2.0,     -3544.0/2565, 1859.0/4104, -11.0/40}
};

static double rkf45_mu[RKF45_STAGE] = {
  16.0/135, 0.0, 6656.0/12825, 28561.0/56430, -9.0/50, 2.0/55
};

static double rkf45_mu2[RKF45_STAGE] = {
  25.0/216, 0.0, 1408.0/2565, 2197.0/4104, -1.0/5
};

double sum(int n, double *x)
{
  int i;
  double s = 0;
  for (i = 0; i < n; i++) s += x[i];
  return s;
}

/* 簡単なチェック */
void check_table()
{
```

```
  int i;
  for (i = 0; i < RKF45_STAGE; i++)
    printf("%g %g\n", rkf45_alpha[i], sum(i, rkf45_beta[i]));
  printf("rkf45_mu の和=%g\n", sum(RKF45_STAGE, rkf45_mu));
  printf("rkf45_mu2 の和=%g\n", sum(RKF45_STAGE, rkf45_mu2));
}

double dotprod(int n, double *x, double *y)
{
  int i;
  double s = 0;
  for (i = 0; i < n; i++) s += x[i] * y[i];
  return s;
}

/*
 * RKF45 で時刻 *t から、1 ステップ進む
 * 特に問題なければ *hh だけ進むが、単位長さあたりの許容誤差限界 eps_tol
 * (解説の ε /H に相当) を達成するために、必要ならば刻み幅の自動調節をする。
 */
int rkf45(double *newx,
  double *t, double x, double *hh, double hmin, double eps_tol,
  double f(double, double))
{
  int i, s = RKF45_STAGE;
  double k[RKF45_STAGE], newx1, newx2, h = *hh, error;
  do {
    /* k1,k2,k3,k4,k5,k6 を計算する */
    for (i = 0; i < s; i++)
      k[i] = h * f(*t + rkf45_alpha[i] * h, x + dotprod(i, rkf45_beta[i], k));
    /* 5 次精度, 4 次精度の値を計算 */
    newx1 = x + dotprod(s, rkf45_mu, k);
    newx2 = x + dotprod(s, rkf45_mu2, k);
    /* 見積誤差 */
    error = fabs(newx1 - newx2);
    if (error < eps_tol) {
      *newx = newx2;
      *hh = h;
      *t += h;
      return 0;
    }
    // 新 h = 0.8 h ( ε  h/ (H 誤差見積)^(1/4)
    h = 0.8 * h * pow(eps_tol * h / error, 0.25);
  }
  while (h > hmin);
  printf("h=%g<=hmin=%g\n", h, hmin);
  return 1;
}

double f(double t, double x)
{
  return x * x;
}

int main()
{
  int n = 1000;
  double a, b, x, h, hmin, t;
  check_table();
  a = 0; b = 1; h = (b - a) / n; hmin = h / 100000;
  x = 1;
  t = 0;
  while (t < 1.0) {
```

```
    if (t + h > 1.0) h = 1.0 - t;
    if (rkf45(&x, &t, x, &h, hmin, 1e-13, f) != 0) {
      fprintf(stderr, "誤差が小さくならない\n");
      exit(1);
    }
    printf("t=%25.16f, x=%25.16g\n", t, x);
  }
  printf("%25.16g\n", x);
  return 0;
}
```

このプログラムでは爆発の例に出した初期値問題

$$\frac{dx}{dt} = x^2, \quad x(0) = 1$$

を解いているが、(実行してみると[12]) 爆発時刻 $t = 1$ のところで、刻み幅がどんどん小さくなっていくのが分かる。

なお、具体的な時間刻み幅の制御について具体的に書いてある本は多くない (そのためもあって、私自身よく分かっていない部分もあり、上記のプログラムも完成品ではない)。その点で森 [9] は貴重である。そこに掲載されているプログラムのアルゴリズムは (FORTRAN プログラムを C 風に書き直して示せば) 以下のようなものである (と思う。読み違いが無ければ。)。なるほどと思うところもあるが、今一つ納得できていないところもある。

```
maxiter = 100;
epsmac = 1e-15;
// t0 から tn まで解く
// eps は許容絶対誤差、epsmac は計算機イプシロン程度の数
epsv = max(eps, epsmac); // はて、相対誤差ならともかく、絶対誤差で筋が通る?
// eb は単位時間当り許される誤差
eb = fabs(epsv / (tn - t0));
//
t = t0;
h = tn - t0;
//
for (iter = 1; iter <= itermax; iter++) {
  // とにかく刻み幅 h で進んでみて、局所打ち切り誤差の推定もする
  rkf(t, h, x0, &xn, &errmax);
  // 推定誤差が時間刻み幅 h あたりの許容誤差よりも小さいかどうかチェック
  if (errmax <= fabs(eb*h)) { // 小さい場合
    // 時刻を進める
    t += h;
    // 進んだ分だけ残り時間は減らす (最終時刻 - 現在時刻)
    h = tn - t;
    // 残り時間がなければ (計算しきったら) 戻る
    if (fabs(h) <= epsmac)
      return;
    // まだ残り時間があれば続行
    x0 = xn;
```

[12] $t = 0.7$ くらいから、刻み幅の調節が実際に動き始め、$t = 1$ の手前でぐんぐん小さくなっていく。コンパクトなスペースで紹介するにはどうしたら良いか分からないので結果はここには載せない

```
    }
    else {
      // 推定誤差が時間刻み幅 h あたりの許容誤差よりも大きい場合
      // 推定誤差と許容絶対誤差の大小を比較
      if (errmax > epsv)
        // 推定誤差が許容絶対誤差より大きければ刻み幅を一気に 10% に縮める
        h *= 0.1;
      else {
        h *= 0.9 * sqrt(sqrt(eb * fabs(h) / errmax));
        if (fabs(h) < epsmax)
          break;
        }
      }
    }
  }
  fprintf(stderr,
         "rkf45(): t=%g で困難に出会いました。計算を中断します。", t);
  return 1;
```

(なお、この手の問題のシミュレーションについては、伊理・藤野 [10] にも何か書いてあったような気がする。陳蘊剛さんの爆発のシミュレーションはどうだったかな…そのうちに一度まじめに調べてみよう。まあ、これは独り言みたいなものです。)

4.7.4 RKF45 自作プログラム開発

testrkf3.c

```
/*
 * testrkf3.c
 */

#include <stdio.h>
#include <math.h>
#include "rkf3.h"

void myf(double t, double *x, double *f)
{
  f[0] =   x[1];
  f[1] = - x[0];
}

int main()
{
  int i, n;
  double a = 0, b = 1;
  double x[2], newx[2], localerror, error, diff, lastdiff, lasterror;
  double k[2][RKF45_STAGE], work[2];
  double h;
  printf("真の値=%20.15f\n", cos(b));
  printf("    n                  x    誤差  推定誤差\n");
  for (n = 1; n <= 10000; n *= 2) {
    x[0] = cos(a); x[1] = - sin(a);
    h = (b - a) / n;
    error = 0;
    for (i = 0; i < n; i++) {
```

```
      bf_rkf45(2, myf,
       a + i * h, x, h, newx, &localerror,
       k, work);
      x[0] = newx[0];
      x[1] = newx[1];
      error += fabs(localerror);
    }
    diff = fabs(x[0] - cos(b));
    printf("%5d %18.15f %7.1e %7.1e", n, x[0], diff, error);
    if (n != 1) {
      printf(" %4.1f 分の 1 %4.1f 分の 1", lastdiff / diff, lasterror / error);
    }
    printf("\n");
    lastdiff = diff; lasterror = error;
  }
  return 0;
}
```

rkf3.h

```
/*
 * rkf3.h
 */

#define RKF45_STAGE (6)

typedef void function(double t, double *x, double *f);
int bf_rkf45(int d, function f,
     double t, double *x, double h,
     double *nextx, double *error,
     double k[][RKF45_STAGE], double *work);
```

rkf.c

```
/*
 * rkf3.c --- RKF45 (多次元, 何次元でも動くように)
 */

#include <stdio.h>
#include <math.h>   /* fabs() */
#include "rkf3.h"

static double rkf45_alpha[RKF45_STAGE] = {
  0.0, 1.0/4, 3.0/8, 12.0/13, 1.0, 1.0/2};

static double rkf45_beta[RKF45_STAGE][RKF45_STAGE-1] = {
  {   0.0,          0.0,                     0.0,         0.0},
  {   1.0/4,        0.0,                     0.0,         0.0},
  {   3.0/32,       9.0/32,                  0.0,         0.0},
  {1932.0/2197, -7200.0/2197, 7296.0/2197,    0.0,         0.0},
  { 439.0/216,     -8.0,      3680.0/513,  -845.0/4104,   0.0},
  {  -8.0/27,       2.0,     -3544.0/2565, 1859.0/4104, -11.0/40}
};

static double rkf45_mu5[RKF45_STAGE] = {
  16.0/135, 0.0, 6656.0/12825, 28561.0/56430, -9.0/50, 2.0/55
};

static double rkf45_mu4[RKF45_STAGE] = {
  25.0/216, 0.0, 1408.0/2565,  2197.0/4104,   -1.0/5,  0.0
};

static double rkf45_mu_diff[RKF45_STAGE] = {
```

```
  -1.0/360, 0.0, 128.0/4275, 2197.0/75240, -1.0/50, -2.0/55
};

static double sqr(double x) { return x * x; }

#include "rkf_utilities3.c" /* dotprod(), etc. */

/*
 * d 次元の ODE x'(t)=f(t,x) を解く。
 * 時刻 t で値が x だとして、h だけ時間を進めたときの値 nextx を計算する。
 *
 * 注意:
 *   x と next x は同じメモリー領域を指さないようにしておくこと。つまり
 *     bf_rkf45(d, f, t, x, h, x, err, k, w);
 *   のようにして呼び出すと結果は壊れてしまう。
 *   --- こういう仕様にした方が全体に無駄が少なくなると考えた。
 *
 * work は問題の次元 d だけの長さの double 型の作業用領域
 */

int bf_rkf45(int d, function f,
     double t, double *x, double h,
     double *nextx, double *error,
     double k[][RKF45_STAGE], double *work)
{
  int i, j, s = RKF45_STAGE;

  /* x と nextx が重ならないことをチェック */
  if (x == nextx) {
    fprintf(stderr, "bf_rkf45(): x と nextx は同じメモリー領域ではいけない！ ");
    *error = 1e+10;
    return -1;
  }
  /* k1,k2,k3,k4,k5,k6 を計算する */
  for (i = 0; i < s; i++) {
    double *xx = nextx;  /* ちょっとの間使わせてもらう (お行儀が悪いけど) */
    double *kk = work;
    for (j = 0; j < d; j++)
      xx[j] = x[j] + dotprod(i, rkf45_beta[i], k[j]);
    f(t + rkf45_alpha[i] * h, xx, kk);
    for (j = 0; j < d; j++)
      k[j][i] = kk[j] * h;
  }
  /* 次のステップの値を 5 次公式で計算する＆ 4 次公式の誤差を推測する */
  for (j = 0; j < s; j++) {
    nextx[j] = x[j] + dotprod(s, rkf45_mu5, k[j]);
    work[j]  = dotprod(s, rkf45_mu_diff, k[j]);
  }
  /* 推定誤差のノルム */
  *error = 0;
  for (j = 0; j < d; j++) *error += sqr(work[j]);
  *error = sqrt(*error);

  /* もし 4 次の値が欲しければ */
  for (j = 0; j < d; j++)
    work[j] -= nextx[j];
  return 0;
}
```

testrkf1 の実行結果

```
oyabun% ./testrkf3
真の値=    0.540302305868140
```

```
   n                    x     誤差  推定誤差
   1  0.541185897435897 8.8e-04 1.4e-03
   2  0.540325560014864 2.3e-05 8.2e-05 38.0分の1 16.8分の1
   4  0.540302920658938 6.1e-07 5.0e-06 37.8分の1 16.2分の1
   8  0.540302323044084 1.7e-08 3.1e-07 35.8分の1 16.1分の1
  16  0.540302306371086 5.0e-10 2.0e-08 34.2分の1 16.0分の1
  32  0.540302305883314 1.5e-11 1.2e-09 33.1分の1 16.0分の1
  64  0.540302305868605 4.7e-13 7.6e-11 32.6分の1 16.0分の1
 128  0.540302305868154 1.4e-14 4.8e-12 32.3分の1 16.0分の1
 256  0.540302305868140 4.4e-16 3.0e-13 32.5分の1 16.0分の1
 512  0.540302305868139 6.7e-16 1.9e-14  0.7分の1 16.0分の1
1024  0.540302305868140 4.4e-16 1.2e-15  1.5分の1 15.9分の1
2048  0.540302305868141 1.3e-15 7.7e-17  0.3分の1 15.2分の1
4096  0.540302305868142 2.3e-15 9.1e-18  0.6分の1  8.4分の1
8192  0.540302305868144 4.1e-15 5.7e-18  0.6分の1  1.6分の1
oyabun%
```

`testrkf1` の実行結果の分析　区間 $[0,1]$ を 1 等分した場合 ($h=1$) でもそれなりに解けているのがすごい。h が極端に大きいときと極端に小さいときを除くと、h を半分にするごとに誤差は 32 分の 1 程度になっていて、公式が 5 次の精度であることを裏付けている。一方、推定される誤差は 4 次公式と 5 次公式の差を使っているので、4 次公式の誤差を推定したものなので、h が半分になるごとに 16 分の 1 程度になっている。

5　典型的なスキーム (2) 線形多段法

(この節は工事中というか、大工事が必要と考えて欲しい。)

k 段法の公式のところで、Φ が線形の場合、すなわち

$$d_0 x_j + d_1 x_{j+1} + \cdots + d_k x_{j+k} = h\left(\beta_0 f_j + \beta_1 f_{j+1} + \cdots + \beta_k f_{j+k}\right)$$

を線形多段法 (linear multistep method) と言う。ただし $f_j := f(t_j, x_j)$.

$$\begin{cases} \text{explicit(陽的)} \overset{\text{def.}}{\Leftrightarrow} \beta_k = 0 \\ \text{implicit(陰的)} \overset{\text{def.}}{\Leftrightarrow} \beta_k \neq 0 \end{cases}$$

$\rho(\lambda) \overset{\text{def.}}{=} d_0 + d_1\lambda + \cdots + d_k\lambda^k$ の根 $\lambda_1, \cdots, \lambda_k$ について

$$|\lambda_i| \le 1 \quad (i=1,\cdots,k), \quad |\lambda_i| = 1 \text{ となる } \lambda_i \text{ は単根。}$$

という条件が安定条件である。

多段法には以下の特徴がある。

(1) 出発にあたって、未知の値 $x_1, \cdots, x_{k-1}$ を何らの方法で求めねばならない。

(2) 計算の途中で stepsize h を変更するのが難しい。

(3) ステップあたりの f の計算回数が少ないままで、高次の公式が作れる。

(4) 安定性に注意が必要。

5.1　PC (予測子修正子法)

(準備中)

6 その他の方法

6.1 Taylor 法

f の Taylor 展開を利用する方法である。簡単で $\left(\frac{d}{dx}\right)^k f$ の計算しやすい f に対して有効である。

例 6.1 (Bessel 関数) これは一松 [7] に載っていた話。Bessel 関数 J_n は、Bessel の微分方程式

$$x^2y'' + xy' + (x^2 - n^2)y = 0 \tag{7}$$

を満たす。この方程式を一回微分すると

$$x^2y''' + 3xy'' + (x^2 - n^2 + 1)y' + 2xy = 0.$$

$k \geq 2$ に対して、(7) を (Leibniz の公式を用いて) k 回微分すると

$$x^2y^{(k+2)} + k\cdot 2x\cdot y^{(k+1)} + \frac{k(k-1)}{2}\cdot 2\cdot y^{(k)} + xy^{(k+1)} + k\cdot 1\cdot y^{(k)} + (x^2 - n^2)y^{(k)} + k\cdot 2x\cdot y^{(k-1)} + \frac{k(k-1)}{2}\cdot 2\cdot y^{(k-2)} = 0.$$

整理して

$$x^2y^{(k+2)} + (1+2k)xy^{(k+1)} + (x^2 - (n^2 - k^2))y^{(k)} + 2kxy^{(k-1)} + k(k-1)y^{(k-2)} = 0 \quad (k \geq 2)$$

を得る。(ちなみにこの式は、$k=1$ のとき $k(k-1)y^{(k-2)} = 0$, $k=0$ のとき $2kxy^{(k-1)} + k(k-1)y^{(k-2)} = 0$ とみなせば、任意の $k \geq 0$ について成立する。) それゆえ $a_k := J_n^{(k)}(x_0)$ とおくと、

$$\begin{cases} a_2 + \dfrac{1}{x_0}a_1 + \left(1 - \dfrac{n^2}{x_0^2}\right)a_0 = 0, \\ a_3 + \dfrac{3}{x_0}a_2 + \left(1 - \dfrac{n^2-1}{x_0^2}\right)a_1 + \dfrac{2}{x_0}a_0 = 0, \\ a_{k+2} + \dfrac{2k+1}{x_0}a_{k+1} + \left(1 - \dfrac{n^2-k^2}{x_0^2}\right)a_k + \dfrac{2k}{x_0}a_{k-1} + \dfrac{k(k+1)}{x_0^2}a_{k-2} = 0 \end{cases}$$

という漸化式を得る[13]。ゆえに $a_0 = J_n(x_0)$, $a_1 = J_n{}'(x_0)$ が分かれば、順に $a_2, a_3, \cdots$ が計算できる。ゆえに x_0 の回りの望む階数までの Taylor 展開

$$J_n(x) \fallingdotseq \sum_{k=0}^{m} \frac{a_k}{k!}(x - x_0)^k$$

が計算できる。適当な x_1 に対して、

$$J_n(x_1) \fallingdotseq \sum_{k=0}^{m} \frac{a_k}{k!}(x_1 - x_0)^k, \quad J_n{}'(x_1) \fallingdotseq \sum_{k=1}^{m} \frac{a_k}{(k-1)!}(x_1 - x_0)^{k-1}$$

を計算して、再び上と同様にして $\{J_n^{(k)}(x_1)\}_{k=0,1,2,\cdots}$ が計算できるので、x_1 を中心とする J_n の Taylor 展開が計算できる。以下これを繰り返すことで、広い範囲に渡って J_n の値が計算できる。■

6.2 補外法

補外法 (**extrapolation method**) と呼ばれる一群の方法がある。特に、**Richardson** 補外に基づく **Bulirsch-Stoer** 法が有力であるとか。“Numerical Recipes” に Bulirsch-Stoer 法のプログラムが載っている。

(Stoer-Bulirsch [11] に載っているのがそう？ あまり詳しくないようだが。)

[13] 一松先生の本には、最後の項を x_0^2 で割るのを忘れるという、軽い誤植があります。

7　数値的安定性

今まで区間 $[a,b]$ を固定して、分割数 $N \to +\infty$ とした ($h=(b-a)/N$) ときの収束を考えた。応用上は長時間解を追跡したい場合がある。この場合、刻み幅 h を固定して $n \to +\infty$ ($t \to \infty$) としても「変なこと」が起こらないようにしたい。

注意 7.1 (「安定」に関する注意) 常微分方程式の数値解法の話では「安定」という語が何度も出て来たが、状況により違う意味で使われることが多い。まとめておくと

1. 微分方程式の (平衡点, あるいは周期解の) 安定性　(離散化とは無関係)
2. 数値解法の安定性　(離散化に起因する不安定性が起こらない; 丸め誤差とは無関係)
3. 数値的安定性　(丸め誤差の増幅に起因する)

ただし、それぞれの場合にさらに細かい分類がある。■

簡単なテスト問題に適用したときの数値的安定性を調べる、というのが基本的な作業方針 (テスト問題の枠を離れたときにどうなるかの保証はないわけで、そういう意味ではあまりいばれないが)。

7.1　線形安定性解析

$\lambda \in \mathbf{C}$ を $\mathrm{Re}\lambda < 0$ なる定数として

$$\begin{cases} x'(t) &= \lambda x(t) \quad (t \in I \overset{\text{def.}}{=} (a,+\infty)) \\ x(a) &= x_0 \end{cases}$$

について適用してみる。この解は $x(t)=x_0 e^{\lambda t}$ であり、$|x(t)|=|x_0|\,e^{\mathrm{Re}\,\lambda t}$ であるから、$\lim_{t\to\infty} x(t)=0$ である。ところが、数値解法ではしばしば

$$\exists h_1>0,\ \exists h_0 \in (0,h_1) \quad \text{s.t.} \quad \begin{cases} \forall h > h_1 & \lim_{j\to\infty} |x_j| = +\infty \\ 0<\forall h<h_0 & \lim_{j\to\infty} |x_j| = 0 \end{cases}$$

ということが起こる。

1) 中点則の場合 (leapfrog method とも言うらしい). これはいわゆる中心差分近似

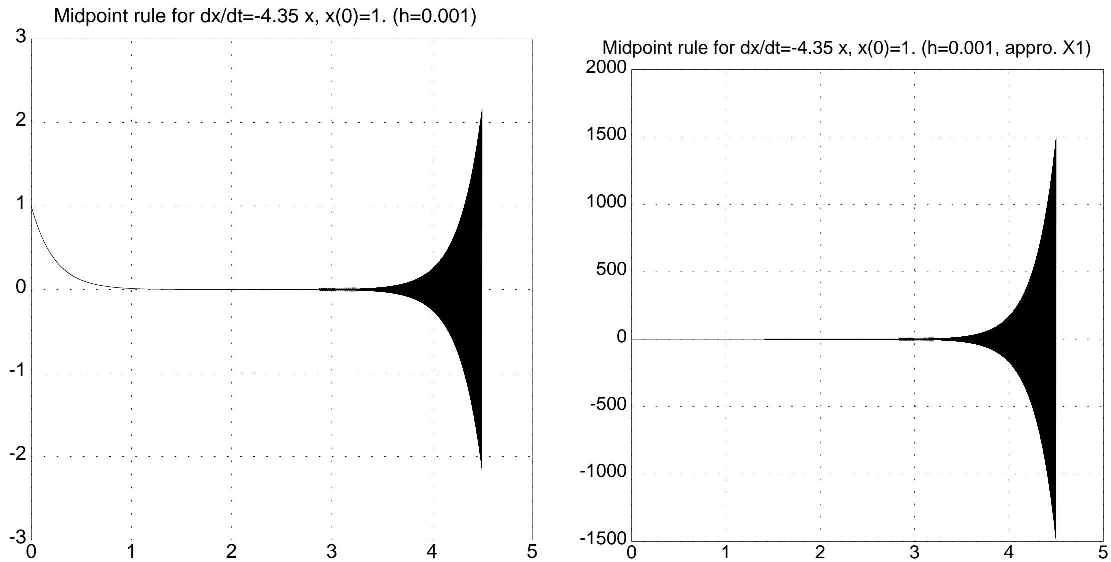
$$x'(t) = \frac{x(t+h)-x(t-h)}{2h} + O(h^2)$$

を用いて作った公式

$$\begin{cases} x_{j+1} = x_{j-1} + 2hf_j, \\ x_1 \text{ は適当に定める} \end{cases}$$

のことを指す。これは任意の $h>0$ に対して $\lim_{j\to\infty}|x_j|=+\infty$ を満たす。

中心差分近似に基づく中点公式 $x_{n+1}=x_{n-1}+hf(t_n,x_n)$ で $x'=\lambda x$, $x(0)=1$ を解く ($\lambda=-4.35$)。左側は $x_1=e^{\lambda h}$ としたもの、右側は $x_1=x_0+hf(t_0,x_0)$ としたもの。



2) R-K 型公式の場合. 公式の段数を s, 次数を m とすると、

$$x_{j+1} = R(\lambda h)x_j$$

のように書ける。ここで $R(z)$ は

$$R(z) = \frac{s\ \text{次以下の多項式}}{s\ \text{次以下の多項式}}, \quad |R(z) - e^z| = O(|z|^m) \quad (|z| \to 0)$$

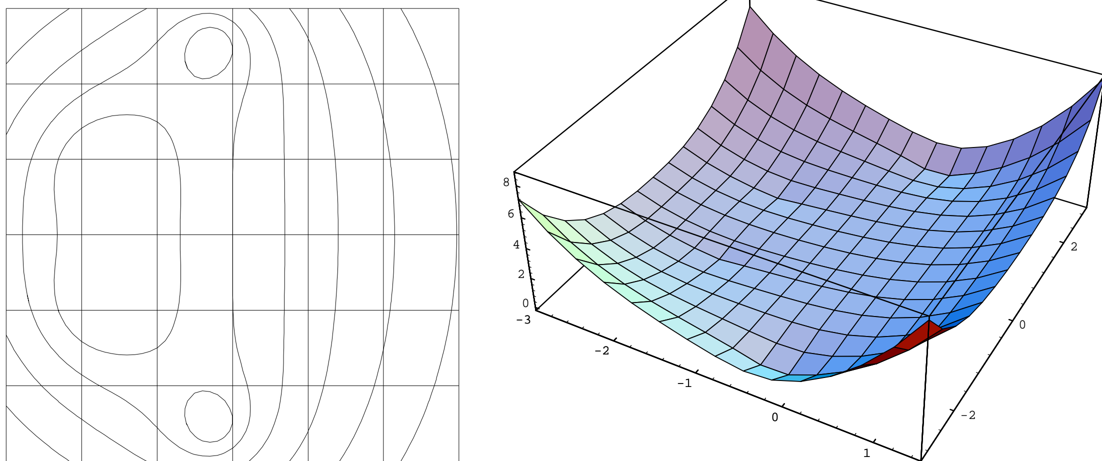
なる z の有理式である (特に公式が explicit の場合には、$R(z)$ は z の多項式になる)。例えば Euler 法の場合 $R(z) = 1 + z$, 古典的 Runge-Kutta 法の場合、$R(z) = 1 + z + \dfrac{z^2}{2} + \dfrac{z^3}{3!} + \dfrac{z^4}{4!}$. この $R(z)$ に対して

$$\mathcal{R} \stackrel{\text{def.}}{=} \{z \in \mathbf{C}; |R(z)| < 1\}$$

を絶対安定領域といい、

$$\lambda h \in \mathcal{R} \Longrightarrow \lim_{j \to +\infty} x_j = 0$$

がなりたつ。絶対安定領域が左半平面 $\{z \in \mathbf{C}; \operatorname{Re} z < 0\}$ を含むような公式を **A-**安定 (**A-stable**) という。A-安定な公式では、$\operatorname{Re} \lambda < 0$ なるとき、任意の $h > 0$ に対して数値解が 0 に収束する。次に示すのは Runge-Kutta 法の絶対安定領域である。カラーでお見せできないのが残念ですが。



なお、レベル 1 の等高線が負軸と交わる点の座標は (巾根で書けはするのだけれど...)、

$$-2.785293563405281623529759190085463034764757896 7169 \cdots.$$

なお、後述の硬い方程式の項を参照せよ。

3) LM 法の場合. 公式

$$(\alpha_0 - z\beta_0)x_j + (\alpha_0 - z\beta_1)x_{j+1} + \cdots (\alpha_0 - z\beta_k)x_{j+k} = 0$$

に対して特性方程式を

$$(\alpha_0 - z\beta_0) + (\alpha_0 - z\beta_1)\xi + \cdots (\alpha_0 - z\beta_k)\xi^k = 0$$

で定義し、その根を $\xi_\ell(z)$ $(\ell = 1, \cdots, k)$ とする。この場合

$$\mathcal{R} \stackrel{\text{def.}}{=} \{z \in \mathbf{C}; |\xi_\ell(z)| < 1\}$$

とおくと

$$\lambda h \in \mathcal{R} \Longrightarrow \lim_{j\to\infty} x_j = 0.$$

8 Stiff problem (硬い問題)

以下は工学的見地からの説明である (新しい言葉に説明がついているが、数学的な定義になっていないものが多い)。

時定数(じていすう) (time scale) とは、解が $\dfrac{1}{e}$ に減衰するのに必要な時間のことである[14]。 $x(t)$ が τ を正の定数として

$$x(t) = \exp\left(-\frac{t}{\tau}\right)$$

のように表されているのならば τ が時定数である。

定義 8.1 (硬い方程式のあいまい定義 — その 1) ある安定な微分方程式が解くべき全区間に比較して極めて小さい時定数をもつ、指数関数的に減衰する解を一つの特解として持つ時、その方程式は **stiff** である (硬い方程式である) と呼ばれる。

定義 8.2 (硬い方程式のあいまい定義 — その 2) 一つの問題の中に時定数が大きな部分と、小さな部分がある時、その方程式は **stiff** である (硬い方程式である) と呼ばれる。

こういう問題を数値解法で解く場合、数値的安定性の要請から小さな時定数にあわせて h を選ぶと、なかなか計算が進まない。これは数値解法に特有の困難である (もとの問題自身は安定である)。

例 8.3 $\lambda_1 \ll \lambda_2 < 0$ なる定数 λ_1, λ_2 に対して、方程式

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

は硬い。 ■

A-stable な方法があればよいが、

[14]半減期というものに似ている。

定理 8.4 (Germund Dahlquist) (i) 陽的 Runge-Kutta 法は A-stable になり得ない。

(ii) 陽的線型多段法は A-stable になり得ない。

(iii) 陰的線型多段法も 3 次以上の公式は存在しない。

(iv) 2 次 A-stable 陰的線型多段法の中では台形則が「最適」である。

のような「否定的な」結果がある。そこで対応策として、

- A-stable はあきらめ、別の (もう少し弱くした) 安定性の概念を考える。Stiff stability (S-安定性) の概念。**Gear**(ギア) の方法。
- A-stable な陰的 Runge-Kutta 法はかなり次数の高い公式が作られている。

などが考えられる。

9 参考書

[1] 笠原 [12] は、明治大学数学科の常微分方程式論の講義で、長くテキストとして採用されていた。数学科の教科書として標準的な内容である。

[2] ハイラー・ネルセット・ヴァンナー [13], ハイラー・ヴァンナー [14] は、常微分方程式の初期値問題の数値解法に関する定評のある文献の、まっとうな翻訳である。著者の Hairer のサイト `http://www.unige.ch/~hairer/software.html` にプログラムが掲載されている。
この本とこのサイトの使い方を説明する方が生産的なのかもしれない (2019/3/6 久しぶりに加筆)。

[3] 三井 [15] は、常微分方程式の数値解析の専門家が書いた数少ない和書であり、特に理論的な解説をきちんとしてある本として和書ではユニークであった。これをコンパクトにしたものに同じ著者の三井 [2] がある。これは三井 [16] の単行本化である。コンパクトではあるが書かれたのが新しいだけの利点があると思う。

[4] 戸川 [17] には、色々な Runge-Kutta 型公式 (made in Japan も多い！) のプログラムがたくさん載っている。

[5] 渡部・名取・小国 [18] は、題名からするとプログラムのみの本のようであるが、そうではなく、常微分方程式の項は杉原正顕氏によるすぐれた解説がある。

[6] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Numerical Recipes (in C), Cambridge University Press, 邦訳 技術評論社.
Bulirsch–Stoer 法が最善の方法であると信じている著者たちのプログラムが載っている (日本語で読める本としては珍しいと思われるので貴重[15])。

[7] Geometric Numerical Integration は何と訳すのかな。Hairer-Lubich-Wanner [19], 初版の書評 McLachlan [20] (2019/3/6 加筆)

[15] この本は非常に有名だが、叩く人も多くて「悪評」だけ読んでいると敬遠してしまいそう。しかし参考になるところは多い。実際この本よりも役立たずの本は世の中にたくさんある。つまり「この本一冊で OK だ」という捉え方をする人 (欧米では多いらしい) に警告を発するために批判されているらしい。

Hairer-Hairer [21] も見てみよう。
やはり http://www.unige.ch/~hairer/software.html に Fortran, MATLAB のプログラムがある。

10 おまけ — 実際的な誤差の推測

急速に α に収束する列 $\{x_n\}_{n\in\mathbf{N}}$ があるとき、

$$\varepsilon_n \overset{\text{def.}}{=} \|x_n-\alpha\|$$

で定義される誤差の大きさについて、十分先の番号 n に対しては

$$\varepsilon_{n+1} \ll \varepsilon_n$$

が成り立つので、

$$\|x_n-x_{n+1}\| \le \|x_n-\alpha\|+\|x_{n+1}-\alpha\| = \varepsilon_n+\varepsilon_{n+1} \fallingdotseq \varepsilon_n.$$

よって

$$\|x_n-x_{n+1}\|$$

を x_n の誤差の大きさ ε_n の見積りとすることが出来る。

急速に収束しない列の場合はどうか？ 例えば n を分割数とした時の差分法の解 $u^{(n)}$ などでは、この仮定が成り立たないと思われる。そういう場合は例えば

$$x_j \overset{\text{def.}}{=} u^{(2^j)}$$

とすることによって、同じテクニックが使える。つまり例えば $n=512$ のときの近似解と $n=1024$ の時の近似解の差の大きさを、$n=512$ の時の近似解の誤差の大きさの見積りとすることが出来る。

A 数値計算するための情報

A.1 はじめに

常微分方程式の初期値問題をコンピューターで解くとき、次の二点が問題になることが多い、と思う。

1. 解の可視化が必要になる可能性が高いが、それをどう実現するか。
2. 問題の次元が高い場合、ベクトルを扱える言語を使うことが望ましいが、何を選択するか。

問題の次元が 1 や 2 であれば、言語や処理系は何でも良いような気がする。それこそ「仮称 (十進)BASIC」[16]などを使うと、とりあえずグラフィックスが使えるので気軽に試せて良いと思われる。

ずっと以前の相場では、FORTRAN 77 くらいのレベルの低いプログラミング言語を使ってプログラムを苦労して書いて、可視化は XY プロッターや、それと同程度の機能を提供するグラフィックス・ライブラリィを利用する、ということだったと思う (例えば、森 [22] などを見よ)。やり方がまずいと、対話性の低いプログラムが出来やすいと思われる。(脱線かも知れないが、ふと、グリック [23] の中のエピソードが浮かんできた。)

今ではもう少し上手いことが出来ると思われるが、案外、それと同程度のことしかやっていない場合が多いかもしれない。例えば、最近の日本の大学の風潮として、C 言語でプログラムを書くという場面が多そうだが、それは FORTRAN 時代とあまり変らない気がする。

実は、身の回りで常微分方程式の初期値問題を真剣に解く状況に遭遇したことがないので、良く分からないが、「気軽な試み」はいくつかしている。

[16] http://hp.vector.co.jp/authors/VA008683/

A.2 C 言語+グラフィックス・ライブラリィ

実に色々ある。以前だったら、全部実際に試して比較してみているところであるが、今はそういう暇がない。

- GLSC[17]
  2 変数関数のグラフの鳥瞰図が気軽に描ける点が「受けている」理由なのかと思われる。「GLSC の紹介」[18]

- PGPLOT[19]
  正直に白状すると、本当の意味で使った経験がない。世の中の評価は高いようなので、これが手堅いのか？ という気がしているのだけど。

- EGGX/ProCALL[20]
  X Window System 用。ある時期使ってみたことがある。気に入っている点も多い。

- GrWin[21]
  Windows 用である。マウスなどが使える。

- **nxgraph.h**[22] — 「計算物理のための C/C++言語入門」[23]

A.3 C 言語+gnuplot

UNIX 風の環境では、gnuplot のインストールが簡単で、C からパイプを使ったプロセス間通信で gnuplot をコントロールすることが出来る。gnuplot ですべて済むとも思えないが、それで済む場合は、それを使うのが楽であるし、便利であると思われる。

「C から gnuplot を呼び出す」[24]

なお、以前の gnuplot は「簡単なグラフを簡便に書くためのツール」であったが、今では gnuplot 自身で Runge-Kutta 法などの計算を行なわせることが可能である。例えば、「gnuplot で微分方程式を解く」[25] が参考になる。

A.4 Java

あまり自信はないが、ほどほどの効率で、対話性の高いプログラムを書くためには Java が便利ではないか？ と思っている。いわゆる GUI が比較的簡単に実現できる。研究室で作成したプログラム例をあげておく。

- $\dfrac{d}{dt}\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}a&b\\c&d\end{pmatrix}\begin{pmatrix}x\\y\end{pmatrix}$ のシミュレーション。http://nalab.mind.meiji.ac.jp/~mk/labo/java/prog/ODE1.html

[17] ftp://ftp.st.ryukoku.ac.jp/pub/ryukoku/software/math/
[18] http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-glsc/intro-glsc.html
[19] http://www.astro.caltech.edu/~tjp/pgplot/
[20] http://www.ir.isas.jaxa.jp/~cyamauch/eggx_procall/
[21] http://spdg1.sci.shizuoka.ac.jp/grwinlib/
[22] http://www-cms.phys.s.u-tokyo.ac.jp/~naoki/CIPINTRO/nxgraph.html
[23] http://cms.phys.s.u-tokyo.ac.jp/~naoki/CIPINTRO/
[24] http://nalab.mind.meiji.ac.jp/~mk/labo/howto/intro-gnuplot/node21.html
[25] http://www.ss.scphys.kyoto-u.ac.jp/person/yonezawa/contents/program/gnuplot/diff_eq.html

- 渦糸のシミュレーション。http://nalab.mind.meiji.ac.jp/~mk/labo/java/uzu.html
(2003 年頃)

- オレゴネーターhttp://nalab.mind.meiji.ac.jp/~mk/labo/report/open/2002-naitou-prog/Oregonator.html
(これも 2003 年頃)

残念ながらここで紹介することは出来ないのだが、実は他所の研究室で、非常に対話性の高い、見事なプログラムを見たことがある。

(2019 年筆 Java について、個人的に 2000 年頃感じていた期待はいつの間にかしぼんでしまった。一つの理由として、Internet 上で、主にセキュリティの観点から、Java が利用しにくい状況になったことがある。もう一つは、たとえ Java アプレットで気軽にシミュレーションが出来ても、それだけでは「イマイチぱっとしない見もの」に過ぎず、意外と教育 (学習) 効果が上がらないらしい、と分かったことがあげられる。)

A.5 C++ の利点

C++ は C と大差ないように思えるかも知れないが、ベクトルを利用しやすくなっているので、案外大きな差があるのではないか？ と想像している。この点で、牧野 [24] に興味を持っている。

グラフィックスについても、何か面白い工夫がありうると思っているのだが…

Eigen というのを知った (2013/12, http://nalab.mind.meiji.ac.jp/~mk/knowhow-2013/node32.html)。これを使うと良いかも。

久しぶりに卒業研究で常微分方程式のシミュレーションをする学生が現れた (複数)。C++ & Eigen & GLSC というラインアップでプログラムを作ってもらった (2019/3/6 加筆)。内容によっては GLSC3D を使うと良いのかもしれない (今回は投影図で済ませた)。

A.6 Ruby

(これは 2011 年頃に書いた。)

前項に続き、牧野センセがらみになるが、Ruby を使うのも面白い。

牧野 [25] では、Ruby+NArray[26] +PGPLOT[27] +Ruby/PGPLOT[28] +rongo[29] というライン・アップで常微分方程式のシミュレーション＆可視化をしている。

そのココロは、

- 簡単な常微分方程式ならば、効率は大した問題ではないから、インタープリター である Ruby くらいでも十分だ。

- むしろベクトルなどが気軽に扱えるのは便利だ。NArray を使えば効率も十分高いし。

- rongo のように言語を拡張してグラフィックス機能を持たせたように出来るのは、嬉しい。

というところだろうか (私のいいかげんなまとめが信用できない人は、上の本を読んで下さい) 。

- Rubyist Magazine「数値計算と可視化」[30]

[26] http://narray.rubyforge.org/index.html.ja
[27] http://www.astro.caltech.edu/~tjp/pgplot/
[28] http://raa.ruby-lang.org/list.rhtml?name=ruby-pgplot
[29] http://jun.artcompsci.org/softwares/rongo/
[30] http://jp.rubyist.net/magazine/?0006-RLR

- Ruby for Scientifit Computing[31]

Ruby のような言語から、各種のライブラリィを簡単に呼び出すというのは面白い。
Scipy[32] なんてのもある。

B　眠りから覚めてみたら (2022/6)

現在、現象数理学科というところに所属しているが、学部のゼミの様子が数学科の時と違っていて、試行錯誤 (悪戦苦闘) している。色々あって、このところゼミでは最初に常微分方程式の初期値問題をやっている。という訳で、長いこと放っておいたものをひっぱり出す羽目になっている。

今さら C 言語でプログラムを書く気はなくて、かと言って Java という気にもなれず、Python, Julia, Ruby (or Crystal) のどれがいいかなあ、と迷っているが、学生は C 言語よりは Python という人が多いので、何となく Python に流れている。

参考文献

[1] 桂田祐史：常微分方程式の初期値問題の数値解法入門, http://nalab.mind.meiji.ac.jp/~mk/labo/text/num-ode.pdf (1995〜2011, 2021 小改訂).

[2] 三井斌友：常微分方程式の数値解法, 岩波書店 (2003),「微分方程式の数値解法 I」岩波講座応用数学 (1993) の単行本化.

[3] 三井斌友, 小藤俊幸(ことうとしゆき), 齊藤善弘(よしひろ)：微分方程式による計算科学入門, 共立出版 (2004).

[4] E. ハイラー, S. P. ネルセット, G. ヴァンナー：常微分方程式の数値解法 I 基礎編, シュプリンガー・ジャパン (2007).

[5] E. ハイラー, G. ヴァンナー：常微分方程式の数値解法 II 発展編, シュプリンガー・ジャパン (2008).

[6] 桂田祐史, 佐藤篤之：力のつく微分積分 — 1 変数の微積分, 共立出版 (2007).

[7] 一松信(ひとつまつしん)：数値解析, 朝倉書店 (1982/10).

[8] Fehlberg, E.: Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme, *Computing*, Vol. **6**, pp. 61–71 (1970).

[9] 森正武：FORTRAN 77 数値計算プログラミング, 岩波書店 (1986, 1987).

[10] 伊理正夫, 藤野和建(よりたけ)：数値計算の常識, 共立出版 (1985).

[11] Stoer, J. and Bulirsch, R.: *Introduction to numerical analysis, second edition*, Springer-Verlag (1992), (R. Bartels, W. Gautschi, and C. Witzgall による翻訳である。).

[12] 笠原晧司(こうじ)：微分方程式の基礎, 数理科学ライブラリー, 朝倉書店 (1982).

[31] http://pub.cozmixng.org/~the-rwiki/rw-cgi.rb?cmd=view;name=Ruby+for+Science
[32] http://www.scipy.org/

[13] E. ハイラー, S. P. ネルセット, G. ヴァンナー：常微分方程式の数値解法 I 基礎編, シュプリンガー・ジャパン (2007), 三井斌友監訳.

[14] E. ハイラー, G. ヴァンナー：常微分方程式の数値解法 II 発展編, シュプリンガー・ジャパン (2008), 三井斌友監訳.

[15] 三井斌友(たけとも)：数値解析入門, 朝倉書店 (1985).

[16] 三井斌友：微分方程式の数値解法 I, 岩波講座 応用数学, 岩波書店 (1993).

[17] 戸川隼人：UNIX ワークステーションによる科学技術計算ハンドブック 基礎篇 C 言語版, サイエンス社 (1992), http://www.mscom.or.jp/~tog/anna/ でプログラムが公開されていたけれど、今はない？

[18] 渡部力, 名取亮, 小国 力監修：Fortran 77 による数値計算ソフトウェア, 丸善 (1989 (1982?), 常微分方程式の項は杉原正顕氏によるすぐれた解説がある。

[19] E. Hairer, C. L. and Wanner, G.: *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential equations*, Springer Series in Computational Mathematics, Springer (2010/3/11), ISBN: 978-3642051579, http://www.unige.ch/~hairer/software.html.

[20] McLachlan, R.: Review: Featured Review: Geometric Numerical Integration: Structure-Preserving Al gorithms for Ordinary Differential Equations. By E. Hairer, C. Lubich, and G. Wanner. Springer-Verlag, New York, 2002. $84.95. xiii+515 pp., hardcover. ISBN 3-540-43003-2., *SIAM Review*, Vol. 45, No. 4, pp. 817–821 (2003).

[21] Hairer, E. and Hairer, M.: GniCodes — Matlab Programs for Geometric Numerical Integration (2003), http://www.unige.ch/~hairer/preprints/gnicodes.ps.gz.

[22] 森正武：FORTRAN 77 数値計算プログラミング, 岩波書店 (1987).

[23] ジェイムズ・グリック：カオス — 新しい科学をつくる, 新潮文庫, 新潮社 (1991), James Gleick, Chaos — making a new science, Viking Penguin (1987) の上田 睆亮監修, 大貫昌子による訳であるが、 せっかくの翻訳が入手しにくくなっている。中古をいとわなければ入手できる。

[24] 牧野淳一郎：パソコン物理実地指導, 共立出版 (1999).

[25] 牧野淳一郎：とんでる力学, 丸善 (2005).