授業用ノート

降旗 大介 (大阪大学)

目 次

§1. 丸め誤差

数値計算、数値解析 (NA) の誤差
= 丸め err. (用いる数体系依存) + 打切 err. (アルゴリズム依存)
は理解してはよね？

- 打切 err. については、各々の計算アルゴリズムの説明時に解析する。
- ちょい退屈だがガマン。

丸め err. とは何か？ ～ 有限なメモリで数を「近似」して表す為に発生する err.

- 実際どれくらい？
- どうやって見積もる？

を知っておかないといかん。その為に…

固定小数点 (小数点を動かさない)

浮動小数点数 (体系) : 小数点の位置を数によって変える、近似実数表現。

例) $42.195 = 4.2915 \times 10^1$ と表現する。
(42.195: 本来の小数点の位置。4.2915: 見かけ上、小数点がここに移動。)

def β 進 n 桁の場合、浮動小数点 x_f は一般に

$$x_f = \pm \underbrace{\left(\frac{x_1}{\beta} + \frac{x_2}{\beta^2} + \cdots + \frac{x_n}{\beta^n} \right)}_{\text{仮数部}} \times \underbrace{\beta^m}_{\text{指数部}}$$

という形となる。

where
$$\begin{cases} 0 < x_1 \leq \beta - 1, \quad 0 \leq x_2, \cdots, x_n \leq \beta - 1, \\ L \leq m \leq U \end{cases} \quad (x_k, m \text{は整数})$$

- β と n が表現の「細かさ」を、L, U, β が「幅」を決める。
- 浮動小数で表せない数との差が丸め誤差となる。

例) $x = 0.3$ を、単精度で近似してみると…

$\beta = 16, n = 6$ では $0.3 \simeq \left(\frac{4}{16} + \frac{12}{16^2} + \frac{12}{16^3} + \frac{12}{16^4} + \frac{12}{16^5} + \frac{12}{16^6} \right) \times 16^0$

この時、「隣」の浮動小数点数との差は $\frac{1}{16^6} \times 16^0 \simeq 6 \times 10^{-8}$。
つまり、丸め誤差は最大で (0.3 に対しては) 6×10^{-8} 程度。

$\beta = 2, n = 24$ では $0.3 \simeq \left(\frac{1}{2} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \cdots + \frac{1}{2^{24}} \right) \times 2^{-1}$

この時の隣との差は $\frac{1}{2^{24}} \times 2^{-1} \simeq 3 \times 10^{-8}$。

・一般に、(β, n, L, U) は、

	単精度	倍精度
大型機	$n=6$ $\beta=16, L=-64, U=63$	$n=14$
PC etc.	$n=24$ $L=-126$ $U=127$	$n=53$ $L=-1022$ $U=1023$
	$\beta=2$	

Q. 浮動小数は何が有利？(固定小数に比して)。
Q. 1つの浮動小数は何 bit？(4バイトで)
Q. 0.3 の丸め誤差はどれだけ？ ($\frac{\beta}{2} - 1$ 辺 $\frac{\beta}{2}$ 入でやってみる)

Daisuke Furihata

Max/Min of 浮動小数. (絶対値的にゼロで)

$$\begin{cases} F_{max} = \left(\frac{\beta-1}{\beta} + \frac{\beta-1}{\beta^2} + \cdots + \frac{\beta-1}{\beta^n}\right) \times \beta^U = \left(1 - \frac{1}{\beta^n}\right)\beta^U \simeq \beta^U. \\ F_{min} = \frac{1}{\beta} \times \beta^L = \beta^{L-1}. \end{cases}$$

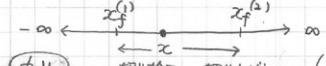
	F_{max}	F_{min}
大型機	$\simeq 16^{63} \simeq 7\times 10^{75}$	$16^{-65} \simeq 6\times 10^{-79}$
PC (単)	$\simeq 2^{127} \simeq 2\times 10^{38}$	$2^{-127} \simeq 6\times 10^{-39}$
PC (倍)	$\simeq 2^{1023} \simeq 9\times 10^{307}$	$2^{-1023} \simeq 1\times 10^{-308}$

Overflow/Underflow

実数 x が、 $|x| > F_{max}$ の時、 Overflow,
$|x| < F_{min}$ 〃 Underflow という.

丸め.

実数 x を浮動小数点数 x_f で近似的に表わすこと，もしくはその方法.



丸め　切り捨て, 切り上げ, (近い方を選ぶ = 四捨五入)　($\frac{\beta}{2}-1$)捨 $\frac{\beta}{2}$入

Machine Epsilon

相対丸め誤差の臨界値 $\varepsilon_M := \max_{F_{min} \le |x| \le F_{max}} |\text{相対丸め誤差}|$ (相対err. := $\frac{\text{err.}}{\text{真値}}$)

$$\left(\simeq \min_{\varepsilon > 0} \left\{ \varepsilon \,\middle|\, \underbrace{(1+\varepsilon) \underset{\eta}{\oplus} 1}_{\text{丸めた結果.}} > 1 \right\} \right)$$

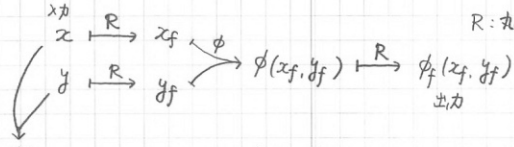
$\longrightarrow$ $x_1, x_2 \in \mathbb{R}$ に対し、 $|x_1 - x_2| < |x_1|\varepsilon_M$ の時は、
x_1 と x_2 の区別が(浮動小数点数上で)無いかもしれない!
(近い数は区別がつかないという可能性)

(ちなみに「逆に」どんなに近い数でも、丸めによって $|x|\varepsilon_M$ 程度の差が
あるとみなされてしまうケースもある. ⇒ どんなに近くても丸めると離れる可能性.)

Q. Overflow しそうな数を挙げてみよ. (意味のある数. 例えば地球上の砂の数とか?)
Q. Underflow 〃
Q. ε_M をおよそで良いので求めてみよ.

基本演算の丸めerr.

単純な演算を対象として、演算に伴って発生する丸めerr.を見る。
例えば、2項演算 $\phi: \mathbb{R}^2 \ni (x,y) \longmapsto \phi(x,y) \in \mathbb{R}$ を考えると、



という図になる。
そして、err.は 出力−真値 であるが、
中間値の $\phi(x_f, y_f)$ を用いて全体とは、引き算すると、

$$\text{丸めerr.} = \phi_f(x_f, y_f) - \phi(x,y)$$
$$= \underbrace{\left\{\phi_f(x_f, y_f) - \phi(x_f, y_f)\right\}}_{\substack{\text{2回目のRによるerr.} \\ \text{入力に関係なく、演算で発生。}}} + \underbrace{\left\{\phi(x_f, y_f) - \phi(x,y)\right\}}_{\substack{\text{1回目のRによるerr.} \\ \text{主に入力値に起因。}}} \quad ④$$

$$\| \text{発生(丸め)err.とよぶ。}$$

発生err. δ については、 $|\delta| \leq c\varepsilon_M |\phi_f(x_f, y_f)|$
という評価が成り立つことが多い。
($c \lesssim 1$ で、正しくは演算で決まる)。

→ 発生相対err.は $c\varepsilon_M$ 以下とみて良い。(あまり恐くない)

恐いのは？

入力値を浮動小数点数の時に出てくる④か、恐いケースがある。

ケース1：桁落ち x, y がとても近い時、かつ、丸め $x_f \neq y_f$ の時に、
$x-y$ と $x_f - y_f$ の相対err.が大きくなる現象。

例) $x = 3.15$, $y = \pi$ として、10進2桁で考えると、
$x_f = 3.2$, $y_f = 3.1$ (4捨5入で) となり、

$x_f - y_f = 0.1 = 1 \times 10^{-1}$、真値は およそ 8×10^{-3}
なので、相対err.は およそ 1000 %！

有名な例) 2次eq. $ax^2+bx+c=0$ の解公式

$$\alpha_\pm = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$$ に対して、

$b^2 \gg |4ac|$ の時に、分子の計算がヤバイ。

例えば、$a=c=1$, $b=200$ とすると、

真値 $\begin{cases}\alpha_+ = -0.005000125\cdots \\ \alpha_- = -199.994999875\cdots\end{cases}$ だが、

2進24ケタ (10進で7~8ケタ分の精度) では

$b=200=(11001000)_2$ で

$(\sqrt{b^2-4ac})_f = (1.1000111111111010111000 1)_2\times 2^7$ より

$-b_f+(\sqrt{b^2-4ac})_f = -(1.01000111)_2\times 2^{-7}$ となり、

ケタ数が減ったことに注意!

$(\alpha_+)_f = -0.00499725341796875$ より、

$err. \simeq 2.87\times10^{-6}$ で、相対 $err. \simeq 6\times10^{-4}$。

となり、数値解 $(\alpha_+)_f$ は10進で4ケタくらいしか正しくない。

(本来は7.8ケタの正しさがあったのに)。

ケース2: 情報落ち (今後も随所に) オーダーの大きく異なる数の和、差を行うと、小さい方の数の情報の多くが反映されないこと。

教科書の例) $\frac{\pi^2}{6}=\sum_{k=1}^{\infty}\frac{1}{k^2}$ に対し、$S_n:=\sum_{k=1}^{n}\frac{1}{k^2}$ をみる。

この時、この式の頭から、$\frac{1}{1^2}$ に $\frac{1}{2^2}$ を足し、さらに $\frac{1}{3^2}$ を…とやっていくと、2進24ケタならば、n≧4096で S_n は変化しなくなる (誤差がこれより良くならない)。

(これを、$\frac{1}{n^2}$ に $\frac{1}{(n-1)^2}$ を足し…と、「足す順番を変える」状況は改善される。)

Q. 上の「有名な例」を自分で確かめてみよう。

Daisuke Furihata

丸め誤差限界.

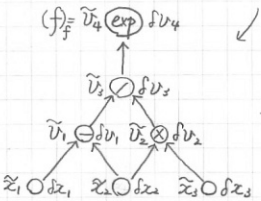
基本演算の組み合わせで関数を実現した場合の、
その関数の丸めerr.の(おおまかな)上界評価 …… のようなもの.

(正しくは、各演算過程で発生したerr.が、それ以降の演算でどう増幅されるかを見積もって合計したもの.)

→ 計算グラフを用いると分かりやすい.

(計算過程をグラフにしたもの
演算がノード.)

例) $f(x_1,x_2,x_3)=\exp\left(\frac{x_1-x_2}{x_2x_3}\right)$ についてみてみよう.



計算グラフ. ノードに演算を、ノードの左に(近似)値を書いておき、
そして、ノードの右側に、
その演算で発生する丸めerr.(近似値−真値)
を書いておく.
(例えば、$\tilde{v}_3 \oslash \delta v_3$ は、
割り算の結果の真値が v_3 で、
近似値が $\tilde{v}_3$ で、err.が δv_3).

この時、$|\delta\cdot|\ll 1$ として、各 $\delta\cdot$ が f までに
どうなるのかを大ざっぱに考える.

例えば δx_2 は、

$$\delta x_2 \xrightarrow{\ominus\text{で}} -\delta x_2 \xrightarrow{①\text{で}} \frac{-\delta x_2}{\tilde{v}_2} \xrightarrow{\exp\text{で}} e^{\tilde{v}_4}\left(\frac{-\delta x_2}{\tilde{v}_2}\right) \simeq -f\cdot\frac{\delta x_2}{x_2x_3}$$

$$\delta x_2 \xrightarrow{\otimes\text{で}} \tilde{x}_3\cdot\delta x_2 \xrightarrow{①\text{で}} -\frac{\tilde{v}_1}{\tilde{v}_2^2}\cdot\tilde{x}_3\delta x_2 \xrightarrow{\exp\text{で}} e^{\tilde{v}_4}\left(-\frac{\tilde{v}_1\tilde{x}_3}{\tilde{v}_2^2}\delta x_2\right)$$

$$\xrightarrow{\text{合計}} \simeq -\frac{f}{\tilde{v}_2}\left(1+\frac{\tilde{v}_1\tilde{x}_3}{\tilde{v}_2}\right)\delta x_2 = -\frac{f}{\tilde{x}_2^2\tilde{x}_3}\delta x_2. \quad (\bigstar)$$

となり、1次オーダーでみると、結果err.に $-\frac{f}{\tilde{x}_2^2\tilde{x}_3}$ 倍程度に大きくなって入ってくる.
この $-\frac{f}{\tilde{x}_2^2\tilde{x}_3}$ を $\frac{\partial f}{\partial x_2}$ と書き、「x_2の摂動に対するfの変化率」と考える.
(v_kも同様)

Q. 上の(★)を自分で計算できるようにせよ.

Q. v_{k+1} (log) v_k ○ δv_k の時、(log)で δv_k は何倍になる? ($\log(v_k+\delta v_k)$ が本来の $\log v_k$ とズレる分の一番粗い近似をみれば分かる)

すると、1次オーダー近似として、

全err.
$$\delta f \simeq \sum_k \frac{\partial f}{\partial v_k} \cdot \delta v_k + \sum_j \frac{\partial f}{\partial x_k} \delta x_k \quad \text{となる.}$$

これに対し、

- 入力値は浮動小数である $(\Leftrightarrow \delta x_k = 0)$
- δv_k を発生err相当で評価可能と仮定する $(\Leftrightarrow |\delta v_k| \le c\varepsilon_M |\tilde{v}_k|)$

のであれば、

関数 f のerr.
$$|\delta f| \lesssim \underbrace{\delta_{Af}}_{\text{丸めerr.限界.}} := c\varepsilon_M \sum_k \left| \frac{\partial f}{\partial v_k} \cdot \tilde{v}_k \right| \quad \text{と評価できよ.}$$

→
- δ_{Af} は具体的に、(δf の計算と共に)計算可能.
- 経験的に、δ_{Af} の評価は悪くないことが知られている.
- 逆に言うと、関数 f を計算した時、その値は δ_{Af} 程度は真値から「ズレ」ている可能性があるので、例えば「$f(x)=0$ を満たす x を求める問題」などは良く考える必要が生じる.
  (δ_{Af} 程度の違いは原理的に検出できない! かも)

Q. $x_1 = 1.2$, $x_2 = 0.89$, $x_3 = 1.3$ として10進2桁で前ページの f の丸めerr.限界を求めてみよ.

§2. 単純な方程式の単純な解を求める。
(非線形方程式の求解)

一般に、$f: \mathbb{R}^n \to \mathbb{R}^n$ に対し、
$f(x) = 0$ を満たす x を求めたい！　としたらどうする？
…… よく出てくる問題、解き方を考える前に…

§2.1. 求解判定。(それは解か？)

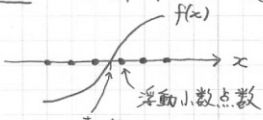
ある x に対し、その x は求めたい解かどうか をどうやって判定する？

素直に、$f(x) \overset{?}{=} 0$ → Yes: x は解である、 / No: x は解ではない、 と判定する … のは ダメ！

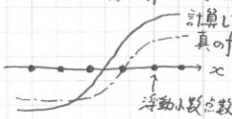
素直な考え方は何故良くないか？

… x も f_x も浮動小数点であって実数全てを尽くしてはいない、よって……

ダメな理由1：真の解を浮動小数点数の時、解を見失う！ かも。



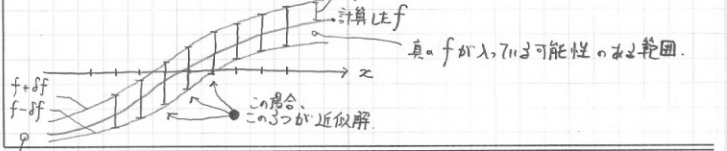
ダメな理由2：f の計算過程で丸め err. が発生するので、
例え真の解∈浮動小数点数でも解を見失う！ かも。



では、どうすべき？
(健全な判定)

上の2つの理由を考えると、例えば次のように考えると良い。

$f_i(x)$ の丸め誤差限界を δf_i として、
$|f_i(x)| \le \delta f_i$ for $\forall i$ $\overset{\text{判定}}{\Rightarrow}$ x は(近似)解である



式は簡単に理解できる。上の範囲内に $y=0$ (⇔ x 軸) があったらそこに真の解が「あるかもしれない」のだから、それを式にすると、$f_i - \delta f_i \le 0 \le f_i + \delta f_i$ となる。
これを変形すると、$-\delta f_i \le f_i \le \delta f_i$ となり、等式となる。

Q. 上の健全な判定で、解を見失うケースを考えよ。
そして、その時はどうすべきか？

Daisuke Furihata

アルゴリズム（解法）

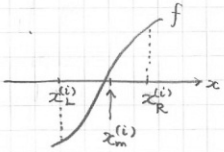
§2.2. $f:\mathbb{R}\to\mathbb{R}$ のケース、
§2.2.1. 囲い込み、

$f:\mathbb{R}\to\mathbb{R}$ は特に薬、f が連続という仮定の下でしばらくやってみよ。

「囲い込み」とは、解の存在する範囲を特定し、その後、その範囲を小さくしていく手法、

二分法、

囲い込み＋反復法（近似解をナンボづつ改善する方法）の最も原始的な方法、



1. どうにかして、$f(x_L^{(0)})\cdot f(x_R^{(0)})<0$ となる $(x_L^{(0)}, x_R^{(0)})$ を見つける。
↓ $i=0$ として
2a. $x_m^{(i)} := \frac{1}{2}(x_L^{(i)}+x_R^{(i)})$ とする。
2b. $f(x_m^{(i)}) \overset{?}{=} 0$ の「判定」。
Yes → $x_m^{(i)}$ を近似解として終る。
No →
2c. $f(x_L^{(i)})\cdot f(x_m^{(i)}) \overset{?}{<} 0$
Yes → $\begin{cases} x_L^{(i+1)} := x_L^{(i)} \\ x_R^{(i+1)} := x_m^{(i)} \end{cases}$
No → $\begin{cases} x_L^{(i+1)} := x_m^{(i)} \\ x_R^{(i+1)} := x_R^{(i)} \end{cases}$
2d. i を1増やして、2a へ戻る。（反復）

注）誤差 ε_i は、$x_R^{(i)}-x_L^{(i)}$ である。

（反復の最終停止）

注）. $[x_L^{(i)}, x_R^{(i)}]\ni$ 解 は分かっているので、解の許容誤差が ε の場合、$x_R^{(i)}-x_L^{(i)}<\varepsilon$ となった時点で $x_m^{(i)}$ を近似解として良い、

特徴、

- 解を常に囲い込んでいる。⇒ 失敗がない、
- 誤差は $\varepsilon_{i+1}=\frac{1}{2}\varepsilon_i$ と変化する、（一次収束）
- $\varepsilon_i=\left(\frac{1}{2}\right)^i\varepsilon_0$ となるので、許容誤差 ε に対し、
$\varepsilon_i<\varepsilon \Leftrightarrow \log_2\left(\frac{\varepsilon_0}{\varepsilon}\right)<i$ となることから、
ナンボでも $\log_2\left(\frac{\varepsilon_0}{\varepsilon}\right)+1$ 回 反復すれば、必ず計算が終了する、
（計算がどれくらいで終わるか、事前に見積もれる）

誤差が反復に伴って
$\varepsilon_{i+1}\simeq C\cdot(\varepsilon_i)^m$
となる時、m 次収束
という、この C を
収束率 という、

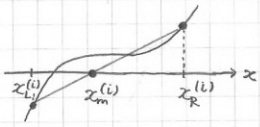
Q. $x=\sqrt{2}$ を知る為に、$f(x)=x^2-2=0$ の解を求める手法が使える、二分法で、
$x_L^{(0)}=1$, $x_R^{(0)}=2$ として、$x_m^{(3)}$ を求めてみよ、

Q（Challenge）. $f(x)=3x^2+1+\frac{1}{\pi^4}\log_e(\pi-x)^2$ の根を求めてみよ、（手で近似値を求めよと難しさが分かる）

Daisuke Furihata

挟みうち法

囲い込み + 反復で、f を1次近似して近似根を求め、その点を新たな区間の端とする。



1. どうにかして、$f(x_L^{(0)}) \cdot f(x_R^{(0)}) < 0$ なる $(x_L^{(0)}, x_R^{(0)})$ を求める。

↓ $i=0$ として

2a. $(x_L^{(i)}, f(x_L^{(i)}))$ と $(x_R^{(i)}, f(x_R^{(i)}))$ の間を直線で結び、x 軸との交点を $x_m^{(i)}$ とする。
(f の一次近似)

2b. $f(x_m^{(i)}) \overset{?}{=} 0$ の「判定」。

No → 2c. (二分法の2cと同じ)

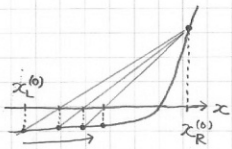
Yes → $x_m^{(i)}$ が近似解、終了。

2d. i を1増やして、2aへ戻る。

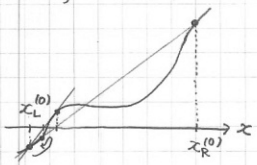
反復

特徴

- 囲い込んでいるので失敗しない。
- 二分法より速いか遅いかは問題次第 (f に依存する)



こういうケースは
二分法より遅い。



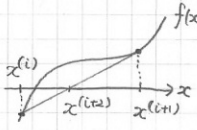
こういうケースは
二分法より速い。

Q. $f(x) = x^2 - 2$ に対し、$(x_L^{(0)}, x_R^{(0)}) = (1, 2)$ として、挟みうち法で $x_m^{(2)}$ を求めてみよ。

§2.2.2 囲い込まない。

割線法。

挟みうちに似ているが、(区間を更新するのではなく) 近似点を更新していく。



1. どうにかして、$f(x^{(0)}) \cdot f(x^{(1)}) < 0$ となる $x^{(0)}, x^{(1)}$ を求める。
↓ $i=0$ として
2a. $(x^{(i)}, f(x^{(i)}))$ と $(x^{(i+1)}, f(x^{(i+1)}))$ を直線で結び、x 軸との交点を $x^{(i+2)}$ とする。
(f の一次近似)
2b. $f(x^{(i+2)}) \overset{?}{=} 0$ の「判定」
No, Yes → $x^{(i+2)}$ を近似解として終了。
2c. i を 1 増やして 2a へ戻る。
反復。

収束次数。

Th. $i \to \infty$ で、$|\varepsilon_{i+1}| \simeq |\varepsilon_i|^{\frac{1+\sqrt{5}}{2}}$ である。(つまり、$\frac{1+\sqrt{5}}{2}$ 次収束)

$\frac{1+\sqrt{5}}{2} \simeq 1.618\cdots > 1$ なので、二分法より速い。
(黄金比)

Proof) $x := x^{(i)}$, $x^+ := x^{(i+1)}$, $x^{++} := x^{(i+2)}$, $\varepsilon = \varepsilon_i$, $\varepsilon^+ := \varepsilon_{i+1}$, $\varepsilon^{++} := \varepsilon_{i+2}$ として、
真の解を α とすると、
2a の手順より $-f'(\alpha)\varepsilon^{++} \simeq \dfrac{\varepsilon^+ f(x) - \varepsilon f(x^+)}{\varepsilon^+ - \varepsilon}$ $\sim$ ① とする。
次に、$f(x)$ を $x^{(i)}, x^{(i+1)}$ で 2 次式で近似し、α での値を求めると
$$0 \simeq \frac{\varepsilon^+ f(x) - \varepsilon f(x^+)}{\varepsilon^+ - \varepsilon} + C_1 \varepsilon \varepsilon^+ \quad \sim ②$$
とする。(C_1 は 2 次項の係数)
よって、①, ② より $\varepsilon^{++} \simeq \left(\dfrac{C_1}{f'(\alpha)}\right) \varepsilon \cdot \varepsilon^+$. $\sim ③$
ここで、$\varepsilon_{i+1} \simeq k\varepsilon_i^a$ と仮定して ③ に代入すると、
$$k^{a+1}\varepsilon^{a^2} \simeq \frac{C_1}{f'(\alpha)} \cdot k\varepsilon^{a+1} \Leftrightarrow \begin{cases} a^2 \cong a+1 \left(\Leftrightarrow a = \frac{1+\sqrt{5}}{2}\right) \\ k^a \cong C_1 / f'(\alpha) \end{cases}$$
$a < 0$ は「収束しない」ので、$a \simeq \frac{1+\sqrt{5}}{2}$.

Daisuke Furihata

特徴
- 計算方法はシンプル.
- 囲い込んでいないので、失敗するかも.
- 収束次数 > 1 なので、うまくいく時は二分法より速く収束する.

Q. 割線法が失敗するケースを考えよ.

Q. $f(x) = x^2 - 2$ に対し、$x^{(0)} = 1$, $x^{(1)} = 2$ として 割線法で $x^{(4)}$ を求めよ.

Daisuke Furihata

§2.3. $f: \mathbb{R}^n \to \mathbb{R}^n$ （$f: \mathbb{R} \to \mathbb{R}$ の手法のいくつかは使えない…）

§2.3.1 縮小写像原理. 反復法を $x^{(i+1)} = g(x^{(i)})$ とみなせて、かつ、g の不動点 = f の根 のケース、と考える.

（復習） Lipshitz連続 g は D で Lipshitz 連続 $\overset{\text{def}}{\Leftrightarrow} \{ \| g(x) - g(y) \| \leq L \| x - y \| \quad \text{for } x, y \in D. \}$

縮小写像. $L < 1$ であれば Lipshitz連続な写像.

Th.
$g(x) \in D$ for $\forall x \in D$ が縮小写像ならば、
1. g の不動点が D にただ1つ存在する（一意存在性）
2. g による反復列 $\{ x^{(i+1)} = g(x^{(i)}) \}$ は、$x^{(0)} \in D$ ならば g の不動点に収束率 L で一次収束する.

より ゆるい Th.

Proof) $\| x^{(i+1)} - x^{(i)} \|$ がおさえられるので Cauchy列となることさえ分かれば easy. 略.

Brower の不動点定理.

Th
D を有界凸として、$g(x) \in D$ for $\forall x \in D$ が連続ならば、
g は D に必ず不動点をもつ.

注) 一意性はない.

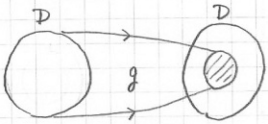
Lemma. $g: D \to D$ が D 上で微分可能で、かつ、D 上で
$|J| < L$ （J: Jacobian, $|J|$ は作用素ノルムで）
ならば、g は D 上で Lipshitz連続で、L は Lipshitz定数.

⇓

これを上手に使えば、近似解を反復法にて求めることができる.

Daisuke Furihata

縮小写像反復法.
($f(x)=0$ の近似解を求める)



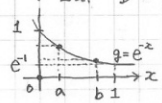
1. g の不動点 $=f$ の根 となるように g を設定する.
2. どうにかして、$g: D \to D$ が縮小写像になるようにする.
   (2a) $g(x) \in D$ for $\forall x \in D$ を示す.
   (2b) $\|g(x)-g(y)\| \le L\|x-y\|$, $L<1$ を示す.
   (Lemma を用いて、$|J|<1$ を示しても良い)
3. $x^{(i+1)} = g(x^{(i)})$ と反復して近似解を求める ($x^{(0)} \in D$)

特徴.
- 囲い込んでいるので失敗しない.
- $\varepsilon_i \simeq L^i \cdot \varepsilon_0$
- 手順の 1, 2 が数学的でやや大変か.
- 計算はシンプル!

例.
$f(x)=e^{-x}-x$ の根を求める.

手順 1. $g(x) := e^{-x}$ とおくと、g の不動点 ($x=g(x)$) は f の根.

↳ 2a. $D=[a,b]$, ただし $0<a<b<1$ とおく.



$a \le g(b)$, $g(a) \le b$ ならば良い. $e^{-1} \simeq 0.3679\cdots$

で、$\min_{0\le x\le 1} g(x) = e^{-1}$ より、$a := e^{-1}$ とおくと $a < g(b)$.

次に、$g(a) = e^{-a} < 1$ ($a>0$ の為) であるので、

$(0.6922\cdots \simeq) e^{e^{-1}} < b < 1$ を満たす b が適当にとれば、よって O.K.

↳ 2b. $g'(x) = -e^{-x}$ より、$|J| = |e^{-x}| < 1$ for $x \in D$ より O.K.

↳ 3. 具体的には、2a の考察より. $a \le 0.3679$, $e^{-a} < b < 1$ とすれば良いので、$a=0.36$ とおくと、$e^{-0.36} \simeq 0.6977$ なので、$b=0.7$ ととれ、$D=[0.36, 0.7]$ とする (例).

よって、初期値 $x^{(0)} \in D$ を、例えば $x^{(0)}=0.5$ として、

$x^{(1)} = e^{-0.5} \simeq 0.6065$, $x^{(2)} = e^{-0.6065} \simeq 0.5453$,

$x^{(3)} \simeq 0.5797$ ($=g(x^{(2)})$), $x^{(4)} \simeq 0.5601$, $x^{(5)} \simeq 0.5711$, $\cdots$

$\longrightarrow x^{(50)} \simeq 0.567143\cdots$ として近似解が求まる.

Q. $f(x)=x^2-2$ の根を縮小写像法で求めよ. ただし、ナレエ夫が必要で、例えば $g(x) := x - \frac{1}{2}f(x)$ などとしないといけないことに注意せよ. (この時、D は $[1, 1.9]$ あたりか?)

§2.3.2 加速法と組み合わせ.

加速法.

(収束しない数列を扱うこともできるが、分かり易さの為)

(収束する) 数列を変換して、より速く収束する数列にする方法.

一次収束列 $\{x_1, x_2, \cdots\}$ s.t. $|x_{i+1} - \alpha| \le L|x_i - \alpha|$ (α が収束先.) を例にして説をすすめる. ($L < 1$)

$\Rightarrow$ $e_i := x_i - \alpha$ とすると、$i \to$ 大で $e_{i+1} \simeq L e_i \sim L^{i+1} e_0$ と考えて、

$x_i \simeq \alpha + CL^i + \gamma_i$ ($\gamma_i \xrightarrow{i\to\infty} 0$) と思われる. つまり、$CL^i$ が分かれば $x_i - CL^i \simeq \alpha + \gamma_i$ と出来て、α の「より良い」近似が得られる…

Richardson 加速.

肝心の L を知っているとして使える加速.

$$\begin{cases} x_{i+1} \simeq \alpha + CL^{i+1} \\ Lx_i \simeq L\alpha + CL^{i+1} \end{cases} \text{ より、} \quad \frac{x_{i+1} - Lx_i}{1-L} \simeq \alpha \quad \text{と考えて、(16.1)}$$

$$y_i := \frac{x_i - Lx_{i-1}}{1-L} \quad \text{と変換する方法.} \ (i \ge 2)$$

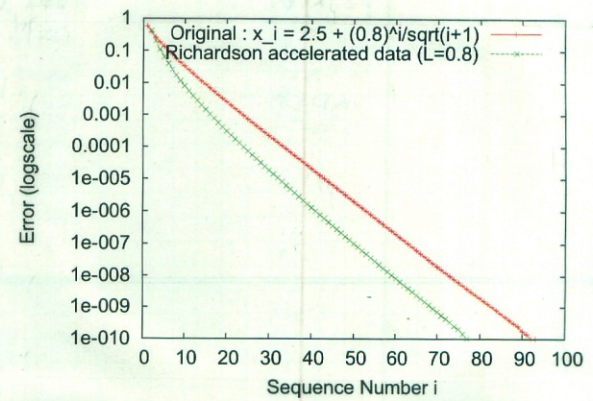
($\{x_1, x_2, \cdots\}$ よりも、$\{y_2, y_3, y_4, \cdots\}$ の方が速く α に近づくと期待できる).

注). 本来は、Richardson 加速の対象は $\{x_i \simeq \alpha + C_1 L_1^i + C_2 L_2^i + C_3 L_3^i + \cdots\}$ であり、

$\{x_i\} \xrightarrow[L_1 \text{とみて}]{} \{y_i\} \xrightarrow[L_2 \text{とみて}]{} \{z_i\} \xrightarrow[L_3 \text{とみて}]{} \cdots$ と、くり返して加速するものである.

例1

$\left\{x_i = 2.5 + \frac{(0.8)^i}{\sqrt{i+1}}\right\}_{i=1}^{\infty}$ を対象としてみよう. $L = 0.8$ として Richardson 加速を行って $\{y_i\}_{i=2}^{\infty}$ を作ると… 簡単な計算で誤差が 1ケタ下がる! これはおいしい.



Q. 右のようなグラフを自分でも作ってみよ.

Daisuke Furihata

Aitken 加速

必要な定数 L を知らなくても、推測して加速。

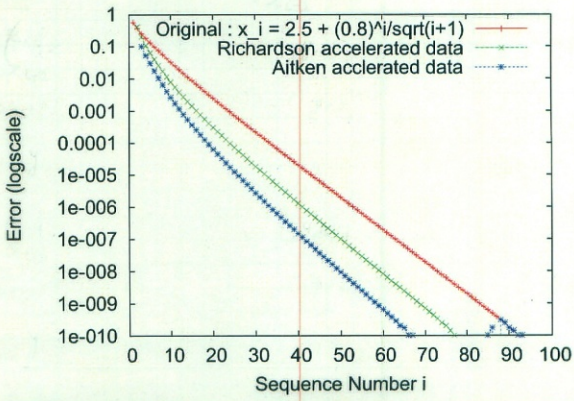
$$\begin{cases} x_{i+1} - x_i \simeq CL^i(L-1) \\ x_{i+2} - x_{i+1} \simeq CL^{i+1}(L-1) \end{cases} \quad \text{より、} \quad L \simeq \frac{x_{i+2} - x_{i+1}}{x_{i+1} - x_i}$$

と推測して、おとは Richardson 加速を適用する。まとめると、

$$y_i := x_i - \frac{(x_i - x_{i-1})^2}{x_i - 2x_{i-1} + x_{i-2}} \qquad (i \geq 3) \qquad \text{となる。} \quad (17.1)$$

($\{x_1, x_2, x_3, x_4, \cdots\} \longrightarrow \{y_3, y_4, \cdots\}$ となる).

例2. 先のと同様に、$\left\{ x_i = 2.5 + \frac{(0.8)^i}{\sqrt{i+1}} \right\}$ に対して Aitken 加速を行ってみる。

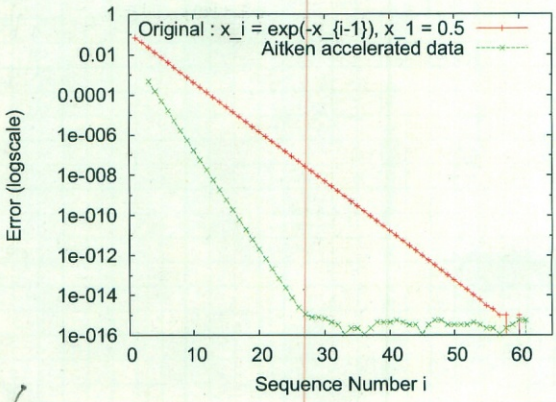


注) 何故 Aitken 加速の方が良いのか？ 考えてみよう。

⋮

などの加速法を、例えば 縮小写像法 と組みあわせることで、
より収束の速い（つまり、より真の解に近づくのが速い）近似解列 が得られる。

例3. 例えば、p15 の $f(x) = e^{-x} - x$ の根を求める縮小写像法の近似解列を、
$\{x_i\} = \{0.5,\ g(0.5),\ g(g(0.5)),\ \cdots\}$ として、これに Aitken 加速を行ってみる。
($g(x) = e^{-x}$)



↓

Aitken加速でこんなに良くなる！ これは知らないと損だ．

注) 一般に、縮小写像法の写像 g の計算コストの方が Aitken加速(17.1)のコストよりも（はるかに）大きい．よって、加速した方が、より小さい計算コストで同じ精度を実現できることになる．

例えば エラー $\simeq 10^{-8}$ としたいとすると、上の図から

- 本来：30回程度の g 反復
- Aitken加速あり：10回程度 〃

となり、大雑把にいって 計算コストが 1/3 程度で済むことになる．

Q p.15 の Q の縮小写像法によって得られるデータと、そのデータに Aitken加速をして得られるデータを上のようにして比較せよ．

Daisuke Furihata

Steffensen反復.

$x_i,\ g(x_i),\ g(g(x_i))$

↓ Aitken加速

x_{i+1}

↓

$x_{i+1},\ g(x_{i+1}),\ g(g(x_{i+1}))$

↓ Aitken加速

x_{i+2}

↓

$x_{i+2},\ g(x_{i+2}),\ g(g(x_{i+2}))$

↓

⋮

($g(x)$の不動点を求める為に) とする方法. 縮小写像とAitkenを「混ぜた」ものの式で書くと,

$$x_{i+1} = g(g(x_i)) - \frac{\{g(g(x_i)) - g(x_i)\}^2}{g(g(x_i)) - 2g(x_i) + x_i}$$

収束次数

Th. $x_i \simeq \alpha$ (収束先, 不動点) 且つ $g'(\alpha) \neq 1$ ならば,

$$|x_{i+1} - \alpha| \simeq C |x_i - \alpha|^2 \quad (2\text{次収束})$$

Proof). $\varepsilon := x_i - \alpha$, $\varepsilon^+ := x_{i+1} - \alpha$ とすると, (面倒なので x_i を x と書いて)

$$g(x) = g(\alpha+\varepsilon) = g(\alpha) + g'(\alpha)\varepsilon + O(\varepsilon^2) = \alpha + C\varepsilon + O(\varepsilon^2). \quad (C := g'(\alpha))$$

$$g(g(x)) = g(\alpha + C\varepsilon + O(\varepsilon^2)) = g(\alpha) + g'(\alpha)\left(C\varepsilon + O(\varepsilon^2)\right) + O(\varepsilon^2)$$

$$= \alpha + C^2\varepsilon + O(\varepsilon^2). \quad (\text{これらを上の式に代入して},)$$

$$x_{i+1} = \alpha + C^2\varepsilon + O(\varepsilon^2) - \frac{\left[\{\alpha + C^2\varepsilon + O(\varepsilon^2)\} - \{\alpha + C\varepsilon + O(\varepsilon^2)\}\right]^2}{\{\alpha + C^2\varepsilon + O(\varepsilon^2)\} - 2\{\alpha + C\varepsilon + O(\varepsilon^2)\} + \alpha + \varepsilon}$$

$$= \alpha + C^2\varepsilon + O(\varepsilon^2) - \frac{\{C(C-1)\varepsilon + O(\varepsilon^2)\}^2}{(C-1)^2\varepsilon + O(\varepsilon^2)}$$

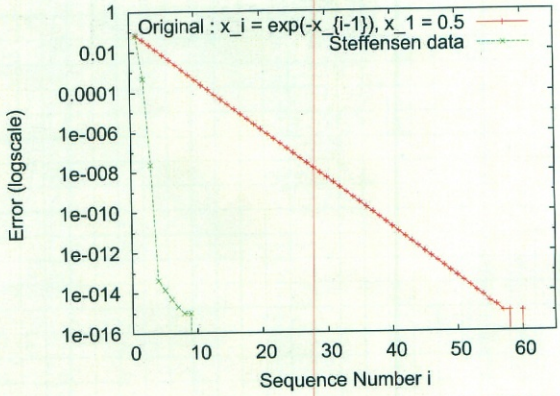
$$= \alpha + C^2\varepsilon + O(\varepsilon^2) - \frac{C^2(C-1)^2\varepsilon^2}{(C-1)^2\varepsilon} + O(\varepsilon^2)$$

$$= \alpha + O(\varepsilon^2) \quad \text{となるので},$$

$$\varepsilon^+ = x_{i+1} - \alpha = O(\varepsilon^2). \quad \square$$

Q. p.18の $f(x) = e^{-x} - x$ の根を求める為の縮小写像 $g(x) = e^{-x}$ に対し, $x_1 := 0.5$ として Steffensen反復を行い, 単なる $\underbrace{g(g(\cdots}_{i\text{回}}(x_1))\cdots)$ と収束性を比較せよ.

p.19のQの解のerrorをグラフにしたもの
→



…… Steffensen法 恐るべし.

Q. p.18のグラフとp.20のグラフを比較して、各々（平均で）gの計算1回でどれだけerrorの桁を下げられるか調べてみよ.（効率の比較）
（例えば、errorが10^{-3}の段階から、gの計算を5回行ってerrorが10^{-6}になったとしたら、3桁下げたとして、3/5桁/回の効率である、と考える）

Daisuke Furihata

§2.3.3 Newton法. Taylor展開の1次近似までを用いる素直な方法. 反復法の一種.

$f: \mathbb{R}^n \to \mathbb{R}^n$ に対し、$f(x) = 0$ となる x を求める為に.

$x^{(i+1)} = x^{(i)} - J^{-1} \cdot f(x^{(i)})$ $\left(J \text{は} f \text{のヤコビ行列}. J = \left\{ \frac{\partial f_i}{\partial x_j} \right\} \right)$.

とする方法.

そのココロは？

現状：$f(x^{(i)}) \neq 0$ に対し、

期待：$f(x^{(i+1)}) \simeq 0$ と考えて、 $x^{(i+1)} = x^{(i)} + \Delta$ とすると、

$0 \simeq f(x^{(i+1)}) = f(x^{(i)} + \Delta) \simeq f(x^{(i)}) + J(x^{(i)}) \cdot \Delta$ より、

$\Delta \simeq - J^{-1} \cdot f(x^{(i)})$.

落とし穴 !!

$J^{-1} \cdot f(x^{(i)})$ 部分の計算の為に J^{-1}、即ち J の逆行列を計算するのは NG!

(理由) $J^{-1} f$ を求めるには、$Ja = f$ という連立一次方程式を解けば良いだけなので計算量が小さいため、(J^{-1} を求めるよりも) J に0要素が多いとこの差はより広がる.

ちなみに、掃き出し法で J^{-1} を求めるには n^3 回ぐらいの乗算が必要.

2次収束するよ！

Th. Newton法は(うまくいく時は)2次収束する.

Proof). (手抜きバージョン). 真の解 α に対し、$x^{(i)} = \alpha + \varepsilon$, $x^{(i+1)} = \alpha + \varepsilon^+$ とする.

$0 = f(\alpha) = f(\underset{x^{(i)} \text{のコト}}{x} - \varepsilon) = f(x) - J(x) \cdot \varepsilon + O(|\varepsilon|^2)$.

$\Rightarrow J(x) \cdot f(x) = \varepsilon + O(|\varepsilon|^2)$.

よって、$x^+ = x - J^{-1} f = (\alpha + \varepsilon) - (\varepsilon + O(|\varepsilon|^2)) = \alpha + O(|\varepsilon|^2)$

$\Leftrightarrow \varepsilon^+ = O(|\varepsilon|^2)$. ∎

特徴.

- 汎用性が高い.
- f の微分 (J) が必要.
- 失敗するかも.
- 2次収束. (うまくいく時はそこそこ速い)
- J が正則でないと使えない.

Q. p.15の例の $f(x) = e^{-x} - x$ の根を、Newton法で求めてみよう. $x^{(0)} = 0.5$ として良いだろう.

Q. p.15のQの $f(x) = x^2 - 2$ 〃 、$x^{(0)}$ は自分で決めよう.

$x \to \alpha$ で J が特異になる場合は？

そもそも
J^{-1} が存在しないと使えない ということは、

$J \xrightarrow{x \to \alpha}$ 特異 の時も何かがおかしくなる可能性が高い。

例.

収束が2次にならない例、1次元問題で、α が M 重根の場合。

$f(x) \underset{\alpha \text{付近で}}{\simeq} C(x-\alpha)^M \{1 + d(x-\alpha)\}$ 、$C \neq 0$ と書けるので、

$J = f' \simeq C(x-\alpha)^{M-1}\{M + d(M+1)(x-\alpha)\}$ となるので、$x = \alpha + \varepsilon$ として

$$x^+ = x - J^{-1}f \simeq (\alpha+\varepsilon) - \frac{C\varepsilon^M\{1+d\varepsilon\}}{C\varepsilon^{M-1}\{M+d(M+1)\varepsilon\}}$$

$$= (\alpha+\varepsilon) - \frac{\cancel{C}\varepsilon^{\cancel{M}}}{\cancel{C}\varepsilon^{\cancel{M-1}} \cdot M}\{1+d\varepsilon\}\left\{1+d\left(1+\frac{1}{M}\right)\varepsilon\right\}^{-1}$$

$$\simeq (\alpha+\varepsilon) - \frac{\varepsilon}{M}(1+d\varepsilon)\left\{1-d\left(1+\frac{1}{M}\right)\varepsilon\right\} \quad \text{Taylor展開}$$

$$= \alpha + \left(1-\frac{1}{M}\right)\varepsilon + \frac{d}{M^2}\varepsilon^2 + O(\varepsilon^3)$$

$\Leftrightarrow$ $\varepsilon^+ \simeq \left(1-\frac{1}{M}\right)\varepsilon + \frac{d}{M^2}\varepsilon^2 + O(\varepsilon^3)$ となるので、

ただし、
- $M \neq 1$ の場合、1次収束
- $M = 1$ 〃 2次収束

となる。

M が分かっていれば対策はある。

Newton法を modify して、

$$x^+ = x - M \cdot \frac{f(x)}{f'(x)}$$

とすると2次収束する。

Daisuke Furihata

例.
(Newton法)

$f:\mathbb{R}^2\to\mathbb{R}^2$ として、 $f(x)=\begin{pmatrix}x^2+y^2-1\\ \sqrt{3}x-y\end{pmatrix}$ $\left(x=\begin{pmatrix}x\\y\end{pmatrix}\right)$ の根を求めよう.

$J(x)=\begin{pmatrix}\frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y}\\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y}\end{pmatrix}=\begin{pmatrix}2x & 2y\\ \sqrt{3} & -1\end{pmatrix}$ より、この場合は手で逆行列が求まって、

$J^{-1}(x)=\frac{1}{2x+2\sqrt{3}y}\begin{pmatrix}1 & 2y\\ \sqrt{3} & -2x\end{pmatrix}$ となるので、

$$\begin{pmatrix}x\\y\end{pmatrix}^{+}=\begin{pmatrix}x\\y\end{pmatrix}-\frac{1}{2x+2\sqrt{3}y}\begin{pmatrix}1 & 2y\\ \sqrt{3} & -2x\end{pmatrix}\begin{pmatrix}x^2+y^2-1\\ \sqrt{3}x-y\end{pmatrix}$$

が Newton法の反復式ということになる.

例えば $\begin{pmatrix}x\\y\end{pmatrix}=\begin{pmatrix}0.5\\0.5\end{pmatrix}$ を初期値として上の反復式を用いてみると、

$\begin{pmatrix}0.5\\0.5\end{pmatrix}\to\begin{pmatrix}0.549038\cdots\\0.950961\cdots\end{pmatrix}\to\begin{pmatrix}0.502189\cdots\\0.869818\cdots\end{pmatrix}\to\begin{pmatrix}0.500004\cdots\\0.866033\cdots\end{pmatrix}\to\begin{pmatrix}0.500000\cdots\\0.866025\cdots\end{pmatrix}$

$\to\begin{pmatrix}0.5\\0.866025\end{pmatrix}$ とまり、5回の反復で解に収束する.

(他に初期値を $\begin{pmatrix}2\\0\end{pmatrix}$ とすると、7回で収束する、… など、初期値で動作が変わる)

Q. $f(x)=\begin{pmatrix}2x^2+y^2-1\\ x-\sqrt{3}y\end{pmatrix}$ の根をNewton法で求めよう.

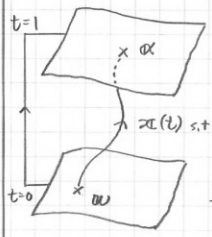
Daisuke Furihata

(連続変形法)
§2.3.4 ホモトピー法
…おそらく最強

$f(x)=0$ の根 α を求めるにあたり、
根 ω がわかっては $g(x)$ を考え、 f と g を「連続につないで」

$$h(x,t) := \begin{cases} g(x) & : t=0 \text{ の時} \\ f(x) & : t=1 \text{ 〃} \\ g \text{ と } f \text{ の混合} & : 0<t<1 \end{cases}$$

一般に $(0\le t\le 1)$ とする。



そして、$h=0$ となる軌跡 $x(t)$ を ω から α まで
t を変えて追跡する。($t: 0\to 1$)

(x,t)空間。

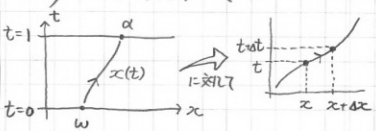
安直な h の作り方。

(あまりお勧めでないが) 例えば、適当に ω を決めて $g(x)=f(x)-f(\omega)$ とする方法がある。こうしておいて、 $h(x,t):=tf(x)+(1-t)g(x)$ とすると上の条件を満たす。

$$(=f(x)+(t-1)f(\omega)) \quad \sim(*)$$

曲線 h の追跡方法。
① 安直バージョン。

easy だが、失敗多し。($x(t)$ が t に対し unique 一価 であれば問題ない。)



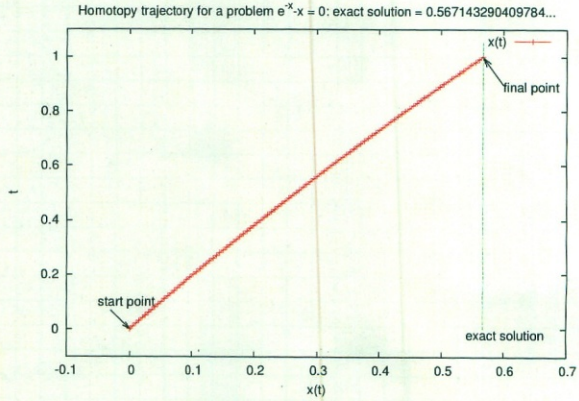
「$(x(t),t)$ の近くに
$(x(t+\Delta t),t+\Delta t)$ があると期待して」

(x,t) を初期値として、
$(\xi,t+\Delta t)$ s.t. $h=0$ なる ξ を
Newton法などで求める。
(初期値が解に近ければ Newton法は有効なので)

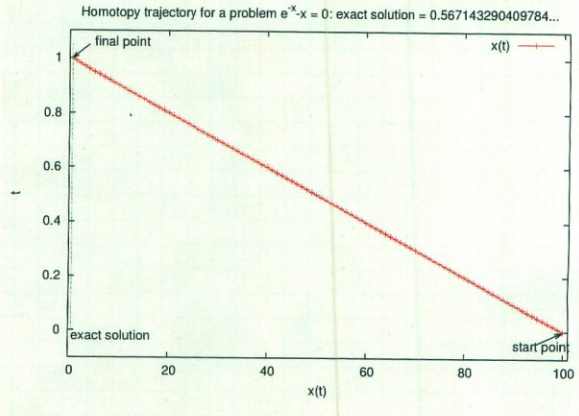
注). この安直な方法と、上の安直な h $(*)$ を合わせると、
「定数ぶんだけ問題を変えて真の解に近づく Newton法」のようになる。

Q. p.15の例の $f(x)=e^{\frac{1}{2}x}$ の根を、上の安直+安直バージョンホモトピー法で求めてみよう。
ω は例えば 0 としてみるとか、思いっきり 100 としてみるとか、

p.24のQに対し、
初期値 = 0.0 として
計算したもの

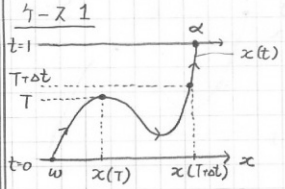


同様に初期値 = 100.0
としたもの。



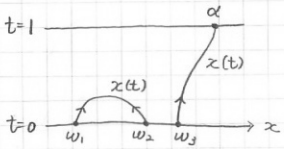
(hの追跡の)
安直バージョンの
失敗するケース.

ケース1



軌跡 x(t) が、左図のように t に対して
多価となる場合、
$(x(T), T) \longrightarrow (x(T+\Delta t), T+\Delta t)$
の間が「大きく離れ」、軌跡を追跡できなくなる。

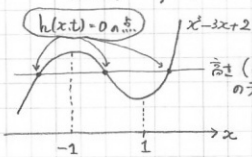
ケース2 (ケース1の変形とも言える)



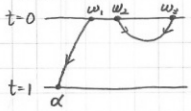
軌跡が t=1 にいく前に消えてしまう場合.

失敗の例.

例えば $f(x) = x^2 - 3x + 3$ の根を求めようとして安直に h を作り、
$h(x,t) = f(x) + (t-1)f(\omega)$ としたとしよう



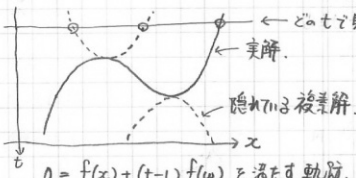
すると、h=0 の点は左図のようになるので、
x(t) は例えば



のようになり、初期値が ω_2 や ω_3 だったら
困る.(ケース2).

どうすべきか?
改善方法 その1.

失敗ケースの状況では、解が「隠れている」ケースもある.
そうした場合は、隠れた解も探索すれば良い.
例えば、上の例では、「複素解」が隠れている為、$x \in \mathbb{C}$ として
解を追跡すると良い.

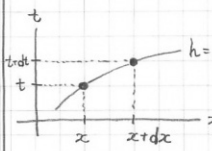


$0 = f(x) + (t-1)f(\omega)$ を満たす軌跡.

Daisuke Furihata

改善方法 その2.

= ② まっとうに軌跡を追跡するバージョン.



軌跡上では $h = Const.(=0)$ であるので、

$$dh = 0$$

であるので、これに着目する、つまり、

$$dh = 0 \Leftrightarrow \sum_{j=1}^{n} \frac{\partial h_i}{\partial x_j} dx_j + \frac{\partial h_i}{\partial t} dt = 0 \quad (i = 1 \sim n) \qquad (27.1)$$

という n 連立方程式を手がかりとする.

未知数 $dx_1, dx_2, \cdots, dx_n, dt$ $(n+1$ 個$)$ を定める為に、(27.1) $(\leftarrow n$ 個を制限する$)$ ともう1つ、適当に「移動距離」を定めて、

$$\begin{cases} (27.1) \\ \sum_{i=1}^{n} (dx_i)^2 + (dt)^2 = Const. \end{cases} \qquad (27.2)$$

粗く追跡して…

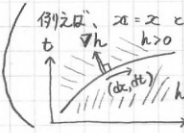
を解いて (dx, dt) を定めて、$(x,t) \xrightarrow{\text{軌跡を追跡}} (x+dx, t+dt)$ とする.

注) ただし、(27.2) は (dx, dt) の正の符号の自由度が残るので、

$$\begin{vmatrix} dx_1 & dx_2 & \cdots & dx_n & dt \\ \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} & \cdots & \frac{\partial h}{\partial x_n} & \frac{\partial h}{\partial t} \end{vmatrix}$$ (27.3)

の符号を変えないように (dx, dt) を決めると、軌跡の上を一定の向きに進むような符号が決められる.

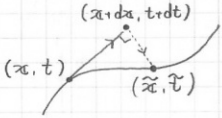
(例えば、$x = x$ として



$\begin{vmatrix} dx & dt \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial t} \end{vmatrix}$ は ∇h と (dx, dt) の外積なので、意味が分かるだろう.)

$h=0$ にのせる.

上の $(x+dx, t+dt)$ は一般に $h = 0$ を満たさないので、$(\tilde{x}, \tilde{t})$ を ($h=0$ を満たす) 次の式を解いて (Newton 法などで) 求める.



$$\begin{cases} h(\tilde{x}, \tilde{t}) = 0 \\ \left(\tilde{x} - (x+dx), \tilde{t} - (t+dt)\right) \perp (dx, dt) \end{cases} \qquad (27.4)$$

これで 1回更新.

こうして、$(x,t) \longrightarrow (\tilde{x}, \tilde{t})$ として、軌跡を追跡できたことになる.

全体.

これをくり返して、$t \simeq 1$ にまだ、その辺りの近似値から 初期値を作って $f(x) = 0$ の Newton 法による近似解を求めれば良い.

Q. $f(x) = x^2 - 3x + 3$ の根を $h(x,t) = f(x) + (t-1) f(w)$, w: 初期値 として ホモトピー法で求めよ. $w = 3$ 辺りが良いだろう.

Daisuke Furihata

p.27のQとやってみる.

(27.2)は $\begin{cases} f'(x)dx + f(\omega)dt = 0 \\ dx^2 + dt^2 = \Delta \end{cases}$

で、$f(\omega) \neq 0$ は確実なので ($f(\omega)=0$ ならば ω が解だ！)

$dt = -\dfrac{f'(x)}{f(\omega)}dx$ を2行目に代入して、

$$\begin{cases} dx = \pm\sqrt{\dfrac{\Delta}{1+\gamma^2(x)}} \\ dt = \gamma\, dx \end{cases}, \quad \gamma(x) := -\frac{f'(x)}{f(\omega)}.$$

そして、符号だが

$$\begin{vmatrix} dx & dt \\ f'(x) & f(\omega) \end{vmatrix} = f(\omega)dx - f'(x)dx = f(\omega)dx - f'(x)\cdot\gamma(x)\,dx$$

$$= \underbrace{\frac{1}{f(\omega)}\left[f^2(\omega) + \{f'(x)\}^2\right]}_{(x,t)\text{に関係なく一定符号}} dx \quad \text{なので、}$$

dx の符号が一定ならば良い。そしてそれは start 地点では $dt > 0$ として考えれば

$$dt\Big|_{\text{start}} = \gamma(\omega)\,dx = -\frac{f'(\omega)}{f(\omega)}dx > 0.$$

今回は $\omega = 3.0$ とすると $\gamma(\omega) = -\frac{24}{21} < 0$ より $dx < 0$ が保たれるべきことが分かる。つまり、

$dx = -\sqrt{\dfrac{\Delta}{1+\gamma^2(x)}}$, $dt = \gamma(x)\,dx$ として粗い近似を得る。

そして、(27.4)を満たす $(\tilde{x}, \tilde{t})$ と、$\begin{cases} \tilde{x} = x + dx + d\tilde{x} \\ \tilde{t} = t + dt + d\tilde{t} \end{cases}$ と書くと、

$$(27.4) \Leftrightarrow \begin{cases} f(x+dx+d\tilde{x}) + (t+dt+d\tilde{t}-1)f(\omega) = 0 \\ dx\cdot d\tilde{x} + dt\cdot d\tilde{t} = 0 \end{cases}$$

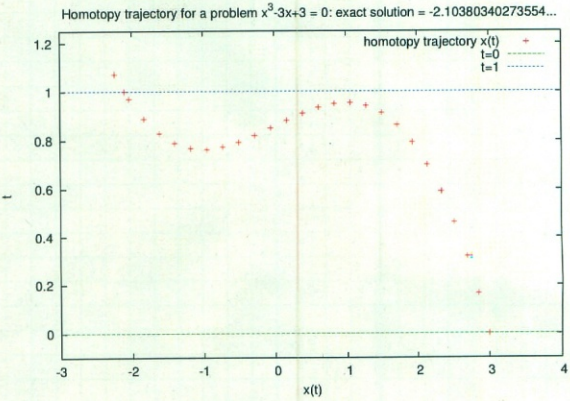
$$\Leftrightarrow \begin{cases} d\tilde{x} = \zeta\, d\tilde{t}, \quad \zeta := -\dfrac{dt}{dx} \quad (dx<0 \text{ より } dx \neq 0) \\ g(d\tilde{t})\Big|_{(x,t,dx,dt)} := f(x+dx+\zeta d\tilde{t}) + (t+dt+d\tilde{t}-1)f(\omega) = 0. \end{cases}$$

つまり、g に対する Newton 法で $d\tilde{t}$ を求め、$d\tilde{x}$ も求められる。

ちなみに、$\dfrac{dg}{d\tilde{t}} = f'(x+dx+\zeta d\tilde{t})\cdot\zeta + f(\omega)$.

あとは原理通り.

p.27のQを実際にやってみた結果.



きちんと軌跡を追跡できていることに着目せよ.

Daisuke Furihata

§2.4 f が多項式のケース.

一般の f に対する方法ももちろん使えるが…

以下、n 次多項式を対象 ($f(x)=P_n(x)$) として考える.

連立法

n 個の根を 1 度に求める方法 (もちろん $\mathbb{C}$ 中で).

(適当な初期値 $\boldsymbol{z}^{(0)}=\{z_1^{(0)}, z_2^{(0)}, \cdots, z_n^{(0)}\}$ を用意し、
$\boldsymbol{z}^{(m+1)}=\boldsymbol{z}^{(m)}+\Delta\boldsymbol{z}^{(m)}$
と更新していく.)

§2.4.1 Durand-Kerner (二次法)

$\boldsymbol{z}^{(m)}$ に対する Newton 法 + 多項式の特性.

まず、普通に Newton 法を考えると、 $\Delta z_i \simeq -\dfrac{P_n(z_i^{(m)})}{P_n'(z_i^{(m)})}$ (30.1) であるが、 $(i=1\sim n)$

P_n の真の根を $\alpha_1, \alpha_2, \cdots, \alpha_n$ とすると

$P_n(z)=a_0\prod_{i=1}^{n}(z-\alpha_i)$, $P_n'(z)=a_0\sum_{i=1}^{n}\prod_{j\neq i}(z-\alpha_j)$ であることと、

$z_j^{(m)}\sim\alpha_j$ であることより、

$$P_n'(z_i^{(m)}) \simeq a_0\prod_{j\neq i}(z_i^{(m)}-\alpha_j) \simeq a_0\prod_{j\neq i}(z_i^{(m)}-z_j^{(m)}) \quad (30.2)$$

とできるので、

$$\Delta z_i^{(m)} = -\frac{P_n(z_i^{(m)})}{a_0\prod_{j\neq i}(z_i^{(m)}-z_j^{(m)})} \quad (i=1\sim n) \quad \text{とする!}$$

注) P_n' を計算しなくて済む. $\Longrightarrow$ さらに 多項式の係数の $a_1\sim a_n$ を知らなくても良い!

これが役立つケース.

n 次行列 A の特性多項式 $P_n(z)=|zI-A|$ については、
$a_0=1$ は分かるが、$a_1\sim a_n$ は計算しないと分からない.
($a_1\sim a_n$ を知らなくても $P_n(z)$ は計算できる).
よって、こうした問題には とても うれしい.

特徴.

- $\mathbb{C}$ 中で計算しないといけない.
- P_n' の計算が不要.
- Newton 法もどきなので、二次収束する. (証明略)

Q. $f(z)=z^3-3z+3$ の根を、Durand-Kerner 法で求めよ. 初期値は p.32 のいずれかを用いよ.

§2.4.2 三次法

$\varphi(z) := \dfrac{P_n(z)}{P_n'(z)}$ として、

$$\Delta z_i^{(m)} = -\frac{\varphi(z_i^{(m)})}{1-\varphi(z_i^{(m)})\sum_{j\neq i}\left(\frac{1}{z_i^{(m)}-z_j^{(m)}}\right)}$$ として z を更新する方法。 (31.1)

この式はどこから？

Newton法の Taylor展開をもう1つ上の order まで打ってみる。まず、

$$0 \simeq P_n(z+\Delta z) = P_n(z) + P_n'(z)\Delta z + P_n''(z)\frac{\Delta z^2}{2} + O(\Delta z^3)$$
$$= P_n(z)\left(1 + \frac{P_n'}{P_n}\Delta z + \frac{P_n''}{2P_n}\Delta z^2\right) + O(\Delta z^3) \quad (31.2)$$

で、Newton法の更新 $\Delta z = -\dfrac{P_n}{P_n'}$ に対し、これを改善するということで

$\Delta z = \dfrac{1}{1+\zeta}\left(-\dfrac{P_n}{P_n'}\right)$, $|\zeta| \ll 1$ として ζ を求めよう。 (31.3)

(31.2) $\Leftrightarrow$ $1 + \dfrac{P_n'}{P_n}\Delta z + \dfrac{P_n''}{2P_n}\Delta z^2 \simeq 0$ に (31.3) を代入すると、

ζ^2 以上を ムシして

$\zeta \simeq -\dfrac{1}{2}\varphi\cdot\dfrac{P_n''}{P_n'}$ (31.4) を得る。

で、$P_n(z) = a_0\prod_{i=1}^{n}(z-\alpha_i)$ に対して

$P_n'(z) = a_0\sum_{i=1}^{n}\prod_{j\neq i}(z-\alpha_j)$, $P_n''(z) = a_0\sum_{\substack{i\neq j\\(i,j)}}\prod_{\substack{k\neq i\\k\neq j}}(z-\alpha_k)$ であるので、

$z_i \simeq \alpha_i$ と考えて

$P_n'(z_i) \simeq a_0\prod_{j\neq i}(z_i-z_j)$, $P_n''(z_i) = 2a_0\sum_{j\neq i}\prod_{\substack{k\neq i\\k\neq i}}(z_i-z_k)$ となり、

$\dfrac{P_n''(z_i)}{P_n'(z_i)} \simeq 2\sum_{j\neq i}\left(\dfrac{1}{z_i-z_j}\right)$ となるので、(31.4) に代入して (31.1) となる。

三次収束

Th. (31.1) は error が三次収束する。

Proof). 計算すると、$\dfrac{P'}{P} = -\dfrac{1}{\Delta z} + \sum\dfrac{1}{z_i-z_j}$

$$\simeq -\frac{1}{\alpha_i - z_i} + \sum\frac{1}{z_i-z_j} + O(\varepsilon), \quad \varepsilon := \alpha_i - z_i$$

$\Leftrightarrow \Delta z = \left(\dfrac{1}{\varepsilon} + O(\varepsilon)\right)^{-1} = \varepsilon + O(\varepsilon^2)$

$\Rightarrow \varepsilon_{New} = \alpha - z_{New} = \alpha - (z+\Delta z) = \varepsilon - (\varepsilon + O(\varepsilon^2)) = O(\varepsilon^3)$. ∎

Q. p.30 の Q を、上の三次法で行え。

Daisuke Furihata

§2.4.3. 初期値の決め方.

Abirth法.

古典的で有名. そんなに良くない?

解の重心 $\left(= -\frac{a_1}{a_0 \cdot n}\right.$, 解と係数の関係より $\left.\right)$ を中心として,
全ての解を中に含むような半径 r_A の円周上に等間隔に初期値をとる.

r_A は? $Q_n(z) := |a_0| z^n - |a_1| z^{n-1} - |a_2| z^{n-2} - \cdots - |a_n|$ の実根. (1つの正かつ正であることがわかっている)

Why? $P_n(z)$ の根 α_i に対し, $|\alpha_i| \leq r_A$ for $\forall i$ がわかっている為.

How to obtain? $r_* := \max_{0 \leq k \leq n} \left| S \cdot \frac{a_k}{a_0} \right|^{1/k}$ (ただし S は非零な a_k の個数) とすると,

$r_* \geq r_A$ がわかっているので, r_* を初期値として Q_n に対する Newton法で r_A が求まる.

Ozawa法 [Jsiam, 1993]

秀れているが, 証明 etc はない.

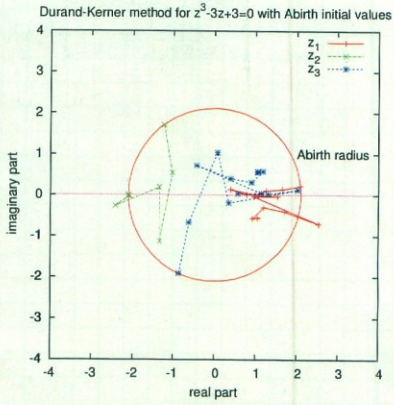
解の重心 $\left(= -\frac{a_1}{a_0 n}\right.$ $\left.\right)$ を中心とし, 重心から各解までの幾何平均 r_{gm} を半径とする円周上に (以下同文).

r_{gm} は? $r_{gm} = \left(\prod_{i=1}^{n} |\alpha_i - \beta| \right)^{1/n} = \left| \frac{P_n(\beta)}{a_0} \right|^{1/n}$, $\beta = -\frac{a_1}{a_0 n}$.

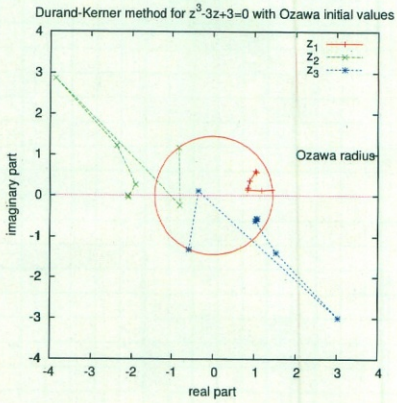
→ 簡単に求められる.

(しかも, 初期値としての性能も良い)

p.30 のQに対し、Abirthの初期値を用いた計算結果。



同じく、Ozawaの初期値を用いたもの。



```
# f(x) = exp(-x)-x の根を求める ホモトピー法

include Math

# f(x), h(x,t) の絶対値がこれより小さければ満足するとする.
Epsilon = 1.0e-15

# 初期値
W = 100.0

# t =[0,1] をどれくらい細かく刻むか
Nt = 100
Dt = 1.0/Nt

# 関数 f
def f(x)
  return exp(-x)-x
end

# f の導関数
def df(x)
  return -exp(-x)-1
end

# f(W)
Fw = f(W)

# 関数 h(x,t)
def h(x,t)
  return f(x)+(t-1.0)*Fw
end

# 関数 h(x,t) の x に関する導関数
def dh(x,t)
  return df(x)
end

# Newton 反復一回分. x も t も最新値.
def newton_one_iteration(x,t)
  return x-h(x,t)/dh(x,t)
end

# 満足いくまで Newton 反復. x も t も最新値.
def newton(x,t)
  y = x
  while h(y,t).abs >Epsilon do
    y = newton_one_iteration(y,t)
  end
  return y
end

# ホモトピー法の初期値.
t = 0.0
x = W

print "t,x¥n"
print t," ",x,"¥n"

# (安直)ホモトピー法で t = 0 から t = 1 まで x を求めていく. 最後の x が求めたかったもの.
for i in (1..Nt)
  t += Dt
  x = newton(x,t)
  print t," ",x,"¥n"
end
```

```ruby
# f(x) = x^3-3x+3 の根を求める ホモトピー法

include Math

# f(x)等の絶対値がこれより小さければ満足するとする.
Epsilon = 1.0e-8

# t が 1 にこれぐらいの距離で近くなるか, t > 1 で普通の Newton 法に切り替える.
DS = 0.01

# 初期値
W = 3.0

# 軌跡をおおよそどれくらいの長さで移動していくか
Dl = 0.05

# 関数 f
def f(x)
  return x**3 -3.0*x +3.0
end

# f の導関数
def df(x)
  return 3.0*(x**2) -3.0
end

# f に関する Newton 反復一回分.
def newton_one_iteration_f(x)
  return x - f(x)/df(x)
end

# f に関して満足いくまで Newton 反復.
def newton_f(x)
  y = x
  while f(y).abs >Epsilon do
    y = newton_one_iteration_f(y)
  end
  return y
end


# f(W)
Fw = f(W)

# 粗い移動方向
def d(x,t)
  gamma = - df(x)/Fw

  dx = - sqrt(Dl/(1.0+(gamma**2)))
  dt = gamma * dx

  return [dx,dt]
end

# 細かい調整時にゼロにする関数 q
def q(dt_tilde,x,t,dx,dt)
  zeta = - dt/dx
  return f(x+dx+zeta*dt_tilde)+(t+dt+dt_tilde - 1.0)*Fw
end

# 細かい調整時にゼロにする関数 q の導関数
def dq(dt_tilde,x,t,dx,dt)
  zeta = - dt/dx
  return df(x+dx+zeta*dt_tilde)*zeta + Fw
end

# q に関する Newton 反復一回分.
def newton_one_iteration_q(dt_tilde,x,t,dx,dt)
  return dt_tilde - q(dt_tilde,x,t,dx,dt)/dq(dt_tilde,x,t,dx,dt)
end

# q に関して満足いくまで Newton 反復.
def newton_q(dt_tilde,x,t,dx,dt)
  y = dt_tilde
  while q(y,x,t,dx,dt).abs >Epsilon do
    y = newton_one_iteration_q(y,x,t,dx,dt)
  end
  return y
end

# ホモトピー法の初期値.
t = 0.0
x = W

print "# t,x¥n"
print t," ",x,"¥n"

# ホモトピー法で近づく
while (1.0-t) > DS do
  dxdt = d(x,t)
  dx = dxdt[0]
  dt = dxdt[1]
  zeta = - dt/dx
  dt_tilde = newton_q(0.0, x,t,dx,dt)
  dx_tilde = zeta*dt_tilde
  x += dx + dx_tilde
  t += dt + dt_tilde
  print t," ",x,"¥n"
end

# 充分近いか t > 1 なので普通の Newton 法を行う(本当は, 最後の二点から初期値近似した方がよい).
x = newton_f(x)
print "1.00000000000000 ",x,"¥n"
# print "# Last Newton result: ",x,"  f(x): ",f(x),"¥n"

```

```ruby
# f(x) = x^3-3x+3 の根を求める Durand-Kerner 法

include Math
require 'complex'

# 虚数単位
I = Complex::I

# f の係数
A = [1.0,0,-3.0,3.0]

# f(x)等の絶対値がこれより小さければ満足するとする.
Epsilon = 1.0e-8

# 関数 f
def f(x)
  return x**3 -3.0*x +3.0
end

# f のゼロ判定
def f_zero(z)
  r = ""
  z.each{ |a|
    if f(a).abs > Epsilon then
      r = false
    end
    }
  return r
end


# Abirth の初期値
def q(z)
  return z**3 - 3.0*z - 3.0
end

def dq(z)
  return 3.0*(z**2)-3.0
end

def newton_one_iteration_q(x)
  return x - q(x)/dq(x)
end

def newton_q(x)
  y = x
  while q(y).abs >Epsilon do
    y = newton_one_iteration_q(y)
  end
  return y
end

def abirth_initial
  b = []
  A.each_with_index { |a,i|
    if a != 0.0 then
      b.push(((2.0*a/A[0]).abs)**(1.0/(i+1)))
    end
    }

  r_star = b.max
  r = newton_q(r_star)
  print "# Abirth radius = ",r,"¥n"

  z = []
  theta = 0.1
  for i in (0..2) do
    z.push(r * exp(I * theta))
    theta += 2.0*PI/3
  end

  return z
end

# Ozawa の初期値
def ozawa
  r = (f(0.0).abs)**(1.0/3)
  print "# Ozawa rarius = ",r,"¥n"
  z = []
  theta = 0.1
  for i in (0..2) do
    z.push(r * exp(I * theta))
    theta += 2.0*PI/3
  end

  return z
end

# Duran-Kerner による反復の分母
def d_dk(z,i)
  dn = 1.0
  for j in (1..2) do
    dn *= z[i] - z[i-j]
  end
  return dn
end

# Duran-Kerner による反復一回分.
def dk_iteration(z)
  dz = []
  for i in (1..z.size) do
    dz[i-1] = - f(z[i-1])/(A[0]*d_dk(z,i-1))
  end
  y = []
  z.each_with_index { |a,i|
    y.push((a + dz[i]))
    }

  return y
end

# f に関して満足いくまで Durand-Kerner 反復.
def durand_kerner(z)
  y = z
  while not(f_zero(y)) do
    print y[0].real," ",y[0].image," ",y[1].real," ",y[1].image," ",y[2].real," ",y[2].image,"¥n"
    y = dk_iteration(y)
```

```
  end
  return y
end

###
print "# Using Abirth initial values¥n"
z = abirth_initial

# Duran-Kerner
z = durand_kerner(z)
print z[0].real," ",z[0].image," ",z[1].real," ",z[1].image," ",z
[2].real," ",z[2].image,"¥n"

print "¥n¥n"
###
print "# Using Ozawa initial values¥n"
z = ozawa

# Duran-Kerner
z = durand_kerner(z)
print z[0].real," ",z[0].image," ",z[1].real," ",z[1].image," ",z
[2].real," ",z[2].image,"¥n"


```

§3. 線形代数

- コンピュータで線形代数って簡単では？

  → 問題が小さければその通り（手計算と同じでよい）
  しかし、実際には「大規模な問題」に取り組まねばならず、きちんと考える必要アリ。

- がんばる価値はあるのか？

  → 世の中の数値計算のコスト（≒計算時間）のほとんどは実は連立一次方程式の求解である。

  （つまり、NAの世界においては、連立一次eqの解法に貢献する奴はヒーローである。）

何を学ぶ？

- 線形代数について一部復習。（ノルム etc.）
- 連立一次eq.の解法
- 固有値問題。

参考文献

- 「数値解析」（第2版）森正武，共立出版，§1，§3あたり
- 「プログラミングのための線形代数」平岡&垣，オーム社

Point

線形代数の教科書を手元に用意しておこう。

§3.1 ノルム etc.

ベクトルのノルム. $x \in \mathbb{C}^n$ に対し.

$$\|x\|_1 := \sum_{k=1}^{n} |x_k|$$

$$\|x\|_2 := \left(\sum_{k=1}^{n} |x_k|^2 \right)^{1/2}$$

$$\|x\|_\infty := \max_k |x_k| \quad \text{など.}$$

ノルムとは何だっけ？

1). $\|x\| \geq 0$ for $\forall x$, $\|x\| = 0 \Leftrightarrow x = 0$.
2) $\|\alpha x\| = |\alpha| \cdot \|x\|$
3) $\|x + y\| \leq \|x\| + \|y\|$.

を満たすもの.

(有限次元ベクトルの) ノルムの同値性

Th. 2種類のノルム $\|\cdot\|_\alpha$, $\|\cdot\|_\beta$ に対し、$\forall x$ に対して

$$m\|x\|_\alpha \leq \|x\|_\beta \leq M\|x\|_\alpha$$

を満たす $m, M > 0$ が存在する.

Proof 略.

つまり、

$\Rightarrow$ $\|x^{(k)} - a\|_\alpha \xrightarrow{k\to\infty} 0$ ならば $\|x^{(k)} - a\|_\beta \xrightarrow{k\to\infty} 0$. であり、

どのノルムで見ても 収束することに変わりはない (収束判定は、どのノルムでやっても本質は同じ)

(行列のノルム) オペレータノルム (natural norm ともいう)

オペレータノルムとは、作用素 $O_p : X \to Y$ に対し、

$$\|O_p\| := \sup_{X \ni x \neq 0} \frac{\|O_p x\|_Y}{\|x\|_X}$$ とするものをいう. これを用いて、

行列 $A \in M_n(\mathbb{C})$ (n次正方行列) のノルムを $\|A\| := \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}$ とする. (35.1)

具体的には

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & & \vdots \\ a_{n1} & \cdots & & a_{nn} \end{pmatrix}$$ として

$$\|A\|_1 = \max_{1 \leq k \leq n} \sum_{i=1}^{n} |a_{ik}|, \quad (35.2)$$

$$\|A\|_2 = \left\{ \rho(A^*A) \right\}^{1/2} \quad (\rho : \text{行列のスペクトル半径} = \text{固有値絶対値の max}) \quad (35.3)$$

$$\|A\|_\infty = \max_{1 \leq k \leq n} \sum_{i=1}^{n} |a_{ki}| \quad (35.4)$$

Q. (35.2) ~ (35.4) を証明してみよ.

行列のノルムの性質

ノルム本来の性質 1)~3) に加えて、

4) $\|Ax\| \leq \|A\| \cdot \|x\|$ for $A \in M_n(\mathbb{C}), x \in \mathbb{C}^n$,

5) $\|AB\| \leq \|A\| \cdot \|B\|$ for $A, B \in M_n(\mathbb{C})$ が成り立つ！

(オペレータノルムでない) 行列のノルム

行列のユークリッドノルム

$$\|A\|_E := \left\{ \sum_{i,j} |a_{ij}|^2 \right\}^{1/2}$$

行列のノルム間の関係

いろいろあるので 例として

Th. $\frac{1}{\sqrt{n}} \|A\|_\infty \leq \|A\|_2 \leq \sqrt{n} \|A\|_\infty$

Proof). 一般に $\|x\|_\infty \leq \|x\|_2 \leq \sqrt{n} \|x\|_\infty$ であることを用いよ.

注) 他の関係 ($\|\cdot\|_1$ と $\|\cdot\|_\infty$ とか) についても試してみよう. 例えば

$\frac{1}{\sqrt{n}} \|A\|_E \leq \|A\|_2 \leq \|A\|_E$ が言える.

スペクトル半径とノルム (オペレータノルムの場合)

A の eigen pair (λ_i, x_i) に対し $Ax_i = \lambda_i x_i$ であることから,

$$\frac{\|Ax_i\|}{\|x_i\|} = |\lambda_i|$$

$$\Rightarrow \|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|} \geq |\lambda_i| \quad \text{for } {}^\forall \lambda_i \text{ となるので},$$

$\|A\| \geq \rho(A)$. (← 固有値絶対値の上限がノルムで分かる)

行列の固有値のノルムによる評価

例1) $\rho(A) \leq \|A\|_1, \|A\|_\infty$ より上限を「計算」できる.
(上の評価式より)

例2). A がエルミート ($A^* = A$) の場合、$\|A\|_2 = \sqrt{\rho(A^*A)} = \rho(A)$ なことと、
$\|A\|_2 \leq \|A\|_E$ を用いて、

$\rho(A) \leq \|A\|_E$ として上限を計算できる.

固有値のありか

Görschgorin Th.

($\mathbb{C}$内で) 中心が a_{ii} で、半径 $\sum_{j\neq i}|a_{ij}|$ の円の内部を S_i として、
行列 $A=\{a_{ij}\}$ の固有値は全て $\bigcup_{i=1}^{n} S_i$ の内部にある。

Proof) eigen pair (λ, x) に対し、x の成分で絶対値最大のものを x_i とおく。
$Ax=\lambda x$ の第 i 行は $\sum_{j=1}^{n} a_{ij}x_j = \lambda x_i$ であるので、これより

$$a_{ii}-\lambda = -\sum_{j\neq i} a_{ij}\frac{x_j}{x_i} \Rightarrow |a_{ii}-\lambda| \leq \sum_{j\neq i}|a_{ij}|.$$

つまり、この λ は S_i の中にいれば。 ▨

↓
どう使う？

例)。固有値探索の際、$\bigcup_{i=1}^{n} S_i$ の中に収める。

例) $1\leq {}^{\forall}i\leq n$ に対し $|a_{ii}| > \sum_{j\neq i}|a_{ij}|$ である行列 (優対角行列という) は、${}^{\forall}$固有値 $\neq 0$。つまり、正則。

§3.2 連立一次方程式

§3.2.1 入力誤差の影響.

$Ax = b$ を解いて x を求める時.

$$A \to A + \Delta A$$
$$b \to b + \Delta b$$

と入力誤差があった時、x にどう影響するのか？

ほぼ恒等な写像の逆.

Lemma. $(I+H)^{-1}$ が存在し、かつ、$\|H\| < 1$ ならば

$$\|(I+H)^{-1}\| \leq \frac{1}{1-\|H\|}$$

を利用する.

Proof) $(I+H)^{-1} = I - H(I+H)^{-1}$ より、

$\|(I+H)^{-1}\| \leq 1 + \|H\| \cdot \|(I+H)^{-1}\|$ として上を得る. □

解析. いくぜ〜

本来： $Ax = b$.

実際： $(A+\Delta A)(x+\Delta x) = b + \Delta b$ として、下が上をうんで

$$\Delta A \cdot x + (A+\Delta A)\Delta x = \Delta b.$$
$$\Leftrightarrow (I + A^{-1}\Delta A)\Delta x = A^{-1}(-\Delta A \cdot x + \Delta b)$$

ここで、$(I + A^{-1}\Delta A)^{-1}$ が存在し、かつ、$\|A^{-1}\Delta A\| < 1$ と仮定して上の lemma を用いると、

$$\|\Delta x\| \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\Delta A\|}\left(\|\Delta A\| \cdot \|x\| + \|\Delta b\|\right)$$

$$\Leftrightarrow \frac{\|\Delta x\|}{\|x\|} \leq \text{〃} \left(\|\Delta A\| + \frac{\|\Delta b\|}{\|x\|}\right) \quad (Ax = b \text{ より})$$
$$\leq \text{〃} \left(\text{〃} + \frac{\|\Delta b\| \cdot \|A\|}{\|b\|}\right)$$
$$= \frac{\|A\|\|A^{-1}\|}{1 - \|A^{-1}\Delta A\|}\left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|}\right)$$

さらに $\|A^{-1}\| \cdot \|\Delta A\| < 1$ と仮定すると、

$$\leq \frac{K(A)}{1 - K(A) \cdot e_A}\left(e_A + e_b\right) \quad \text{つまり…}$$

ただし、$K(A) := \|A\| \cdot \|A^{-1}\|$ (A の条件数),

$$e_\bullet := \frac{\|\Delta \bullet\|}{\|\bullet\|}$$

Daisuke Furihata

逃がされない誤差、

Th. $A \in M_n(K)$ が正則で ΔA が充分小さく、
$\|\Delta A\| < \frac{1}{\|A^{-1}\|}$ を満たすならば、

$$e_x \leq \frac{K(A)}{1 - K(A)\, e_A} \left(e_A + e_b \right) \quad \text{を満たす.}$$

Proof). ほぼ p38 の通り.

あとは、上の仮定 $\Rightarrow$ $I + A^{-1}\Delta A$ が正則　を示せば良い.

これは、オペレータノルムに対し $\|A\| \geq \rho(A)$ なので、

$\rho(A^{-1}\Delta A) \leq \|A^{-1}\Delta A\| \leq \|A^{-1}\| \cdot \|\Delta A\| < 1$ より、

$A^{-1}\Delta A$ の固有値 λ は $|\lambda| < 1$. よって $I + A^{-1}\Delta A$ の固有値 $1+\lambda$ はゼロにならず、$I + A^{-1}\Delta A$ は正則. ▨.

上の Th の意味は？

$Ax = b$ の近似解 の相対誤差 $e_x = \frac{\|\Delta x\|}{\|x\|}$ は、

条件数 $K(A)$ が大きいと 入力誤差 e_A, e_b の影響を強くうける！

（解法に関係なく！）

そこであらためて
条件数
(Condition Number)

$A \in M_n(K)$ が正則な時、

$K_\alpha(A) := \|A\|_\alpha \cdot \|A^{-1}\|_\alpha$ を A の条件数という.

注) A がエルミート ($A^* = A$) の時、$\|A\|_2 = \sqrt{\rho(A^*A)} = \rho(A) = \max_i |\lambda_i|$
なので、

$$K_2(A) = \max_i |\lambda_i| \cdot \max_j \left|\frac{1}{\lambda_j}\right| = \frac{\max_i |\lambda_i|}{\min_j |\lambda_j|} \quad \left(= \max_{i,j} \left|\frac{\lambda_i}{\lambda_j}\right| \right)$$

条件数が大きい例.

例1) Vandermonde 行列.

$$V_n := \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & & & & \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix}$$ に対し、$x_k = k$ としてみよう.

例えば $n=5$ では

α	$\|V_5\|_\alpha$	$\|V_5^{-1}\|_\alpha$	$K_\alpha(V_5)$
1	979	45	44055
2	$\simeq 696$	$\simeq 38$	$\simeq 26170$
∞	781	56	43736

となり、わずか 5×5 のスケールの問題なのに、$V_5 x = b$ の求解の際は、出力誤差(相対) e_x は 入力の相対誤差 e_A, e_b の 10^4 倍以上になりうる!

⇒ 有効桁数が 4つ以上,残る!

例2) Hilbert 行列

$$H_n = \begin{pmatrix} 1 & 1/2 & 1/3 & \cdots & 1/n \\ 1/2 & & & & \vdots \\ \vdots & & & & \\ 1/n & \cdots & \cdots & & \frac{1}{2n-1} \end{pmatrix}$$ に対しては

$$K_\alpha(H_5) = \begin{cases} 943656 & : \alpha = 1, \infty \\ \sim 4.766 \times 10^5 & : \alpha = 2 \end{cases}$$

となり、Vandermonde 行列よりたちが悪い.

Q. V_5 の $K_\alpha(V_5)$ を自分で計算してみよう. V_6, V_7 ではどうか.

Q. H_5 〃 H_6, H_7 〃

(条件数や誤差の推定)

$\|A^{-1}\|$ の推定.

Lemma A^{-1} の近似行列 X を得ている場合、$\|I - XA\| < 1$ ならば

$$\|A^{-1}\| \leq \frac{\|X\|}{1 - \|I - XA\|}$$

Proof) $R = I - XA$ として、$A^{-1} = X + RA^{-1}$ より

$\|A^{-1}\| \leq \|X\| + \|R\| \cdot \|A^{-1}\|$ として題意を得る. ▨

条件数の推定.

上を用いて. $K(A) \leq \dfrac{\|A\| \cdot \|X\|}{1 - \|I - XA\|}$.

誤差の「事後」推定

$Ax = b$ の真の解を x, 近似解を $\hat{x}$ $(= x + \Delta x)$ とおと,

$\|x - \hat{x}\| = \|A^{-1}b - A^{-1}A\hat{x}\| \leq \|A^{-1}\| \cdot \|b - A\hat{x}\|$ より、

$$\|x - \hat{x}\| \leq \frac{\|X\|}{1 - \|I - XA\|} \cdot \|b - A\hat{x}\|$$

と、誤差を「計算後に」推定できる.

§3.2.2 解法の分類

大まく分けると2種類

- 直接法 …… 解を1回で求める。Simpleだが計算量多め。
  小規模内題向き。
  線形代数で学ぶのはこれに属する。
- 反復法 …… 近似解を少しずつ改良していく方法。
  やや複雑だが、計算量少なめ
  大規模内題向き。

§3.2.3 直接法

分類は以下のようになっている。

- $A^{-1} = \frac{\mathrm{adj}\, A}{\|A\|}$ を用いて A^{-1} を計算して $A^{-1}b$ を求める … (42-1)
- Gaussの消去法ベース
  - Gaussの消去法 … (42-2)
  - LU分解 … (42-3)

  まあ、似ているといえば似ている。
- その他

Warning!

(42-1)は使ってはいけない!

理由1. 計算量が大きすぎる。$\|A\|$ の計算だけで $n!$ かかる。
つまり、$(n+1)!$ 程度の計算量 ⇒ 遅いだけでなく、
丸め誤差も大きくなる。

理由2. そもそも A^{-1} を経由することが良くない。
Aの成分の多くがゼロ(こういうAを疎(sparse)という)でも、
A^{-1} は一般に疎ではない。

Gauss の消去法

Gauss の消去法は多少変種があるが、おおよそ以下のようになる。

$$Ax = b \Leftrightarrow \begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1n}^{(1)}x_n = b_1^{(1)} \\ a_{21}^{(1)}x_1 + \cdots = b_2^{(1)} \\ \vdots \\ a_{n1}^{(1)}x_1 + \cdots + a_{nn}^{(1)}x_n = b_n^{(1)} \end{cases} \text{ に対し、}$$

前進消去

第 k 行に ($\forall k > 1$)
$-\frac{a_{k1}^{(1)}}{a_{11}^{(1)}} \times$ (第1行) を足すと

$$\rightarrow \begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1n}^{(1)}x_n = b_1^{(1)} \\ 0 + a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\ \vdots \\ 0 + a_{n2}^{(2)}x_2 + \cdots + a_{nn}^{(2)}x_n = b_n^{(2)} \end{cases}$$

積: $(n-1)\times n$ 回, 和: $(n-1)\times n$ 回

- $a_{ss}^{(s)} = 0$ となる場合どうする？ → (Pivot 変換という操作が必要)

第 k 行 ($\forall k > 2$) に
$-\frac{a_{k2}^{(2)}}{a_{22}^{(2)}} \times$ 第2行 を足すと

$$\rightarrow \begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + a_{13}^{(1)}x_3 + \cdots + a_{1n}^{(1)}x_n = b_1^{(1)} \\ 0 + a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\ 0 + 0 + a_{33}^{(3)}x_3 + \cdots + a_{3n}^{(3)}x_n = b_3^{(3)} \\ \vdots \\ 0 + 0 + a_{n3}^{(3)}x_3 + \cdots + a_{nn}^{(3)}x_n = b_n^{(3)} \end{cases}$$

積 $(n-2)\times(n-1)$ 回　和: 〃

$\vdots$

第 n 行に ($\Leftrightarrow \forall k > n-1$)
$-\frac{a_{n,n-1}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}} \times$ 第 $(n-1)$ 行 を足すと

$$\rightarrow \begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1n}^{(1)}x_n = b_1^{(1)} \\ a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n = b_2^{(2)} \\ \qquad \ddots \\ a_{n-1,n-1}^{(n-1)}x_{n-1} + a_{n-1,n}^{(n-1)}x_n = b_{n-1}^{(n-1)} \\ a_{nn}^{(n)}x_n = b_n^{(n)} \end{cases} \sim (\star)$$

積 1×2 回　和 〃

上三角行列になる。

積の回数 (= 和 〃):

$$\sum_{k=1}^{n-1}(n-k)\cdot(n-k+1) \overset{k' := n-k}{=} \sum_{k'=1}^{n-1} k'(k'+1) = \sum_{k'=1}^{n-1}\left\{(k')^2 + k'\right\}$$

$$= \frac{(n-1)\cdot n\cdot(2n-2+1)}{6} + \frac{(n-1)\cdot n}{2} = \frac{1}{3}n(n-1)(n+1) \simeq \frac{n^3}{3}.$$

⇓

前進消去は $n^3/3$ の計算量。

後退代入

前ページの (☆) に対し、

(第k行 ($\forall k<n$) に対し、 $\left(-\frac{a_{k,n}^{(k)}}{a_{nn}^{(n)}}\right)$ × 第n行 を足す)

→ 積: $(n-1)$回, 和: 〃

$$\begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1,n-1}^{(1)}x_{n-1} + \boxed{0} = b_1^{(2)} \\ a_{22}^{(2)}x_2 + \cdots + a_{2,n-1}^{(2)}x_{n-1} + 0 = b_2^{(2)} \\ \vdots \\ a_{n-1,n-1}^{(n-1)}x_{n-1} + 0 = b_{n-1}^{(n)} \end{cases}$$

$$\boxed{a_{nn}^{(n)}x_n = b_n^{(n)}}$$

もう外せば、($\Leftrightarrow x_n = b_n^{(n)}/a_{nn}^{(n)}$)

(第k行 ($\forall k<n-1$) に対し、 $\left(-\frac{a_{k,n-1}^{(k)}}{a_{n-1,n-1}^{(n-1)}}\right)$ × 第$(n-1)$行 を足す)

→ 積: $(n-2)$回, 和: 〃

$$\begin{cases} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \cdots + a_{1,n-2}^{(1)}x_{n-2} + \boxed{0} = b_1^{(3)} \\ a_{22}^{(2)}x_2 + \cdots + a_{2,n-2}^{(2)}x_{n-2} + 0 = b_2^{(4)} \\ \vdots \\ a_{n-2,n-2}^{(n-2)}x_{n-2} + 0 = b_{n-2}^{(n)} \end{cases}$$

$$\boxed{a_{n-1,n-1}^{(n-1)}x_{n-1} = b_{n-1}^{(n)}}$$

外せば、($x_{n-1} = b_{n-1}^{(n)}/a_{n-1,n-1}^{(n-1)}$)

⋮

(第1行 ($\Leftrightarrow \forall k<2$) に

$\left(-\frac{a_{12}^{(1)}}{a_{22}^{(1)}}\right)$ × 第2行 を足す.

→ 積: 1回, 和: 1回.

$$\begin{cases} a_{11}^{(1)}x_1 + \boxed{0} = b_1^{(n)} \\ a_{22}^{(2)}x_2 = b_2^{(n)} \end{cases}$$

$\Updownarrow$

$a_{kk}^{(k)}x_k = b_k^{(n)}$ for $\forall k$.

$\Updownarrow$ (積 n回)

$x_k = \dfrac{b_k^{(n)}}{a_{kk}^{(k)}}$ 〃

積の回数: $\sum_{k=1}^{n-1} k + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$ 回. $\left(\simeq \frac{n^2}{2}\right)$

(= 和の〃)

$\Downarrow$

後退代入は $n^2/2$ の計算量

・よって Gauss の消去法は $O(n^3/3)$ となる. (逆行列を $\frac{\mathrm{adj}\,A}{|A|}$ で求めるより とても少ない)

Q. A^{-1} を $\frac{\mathrm{adj}\,A}{\|A\|}$ で求めて $A^{-1}b$ とする方法の計算量を $(n+1)!$ とし、
Gauss 消去法のそれを $n^3/3$ としよう.
この2つを比較するグラフを作れ. ($n=1,2,\cdots$ を横軸にする).

・LU分解.
…Gauss消去って少しおかしくない？

Gauss消去をよくみると、$a_{ij}^{(i)}$の計算にはb_kの値は関係ない。
つまり、Aを上三角に変形するのにbは不要だ。そこで、ここをすっきりさせよう。

$A = LU$（L：下三角、U：上三角）と分解できたとして、

$Ax = b \Leftrightarrow LUx = b \Leftrightarrow Ux = L^{-1}b \Leftrightarrow x = U^{-1}(L^{-1}b)$ であるので、
（前々ページの㊙相当）

① $L^{-1}b$を $L\cdot y = b$ を解いて（前進代入するだけ！）yとして求める。
② xを $Ux = y$ と解いて（後退代入するだけ！）求める。

という手順で xを容易に求められる。

↓

では、$A = LU$ と分解するには？ ～前々ページの㊙までをきちんと丸めこめばよい。

$$A = A^{(1)} \xrightarrow[\text{1回目の掃除}]{} A^{(2)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & & \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix}$$ で あるが、

$$\left(a_{ij}^{(2)} = a_{ij}^{(1)} - \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} a_{1j}^{(1)} \right)_{(i,j \geq 2)}$$

これは

$$\begin{cases} A^{(2)} = M_1 A^{(1)} \\ M_1 := I - \widetilde{M}_1, \quad \widetilde{M}_1 = \begin{pmatrix} 0 & & \\ m_{21} & & \\ \vdots & & 0 \\ m_{n1} & & \end{pmatrix}, \quad m_{k1} := \dfrac{a_{k1}^{(1)}}{a_{11}^{(1)}} \ (k \geq 2) \end{cases}$$

確かめておけ！ → と表わせる。

以下同様に

$$\widetilde{M}_k := \begin{pmatrix} 0 & \overset{k}{\overset{\downarrow}{\begin{matrix} 0 \\ \vdots \\ 0 \\ m_{k+1,k} \\ \vdots \\ m_{n,k} \end{matrix}}} & 0 \end{pmatrix}, \quad m_{sk} := \frac{a_{sk}^{(k)}}{a_{kk}^{(k)}} \ (s \geq k+1) \quad \text{とおくと、}$$

$A^{(k+1)} = M_k A^{(k)}$ となる。

つまり、$M_{n-1} M_{n-2} \cdots M_2 M_1 A = U$ となり、$L = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}$ となる。
（㊙相当）

・実は、$M_k^{-1} = -M_k$ となる。

実際に L を求めると、

$$L = \begin{pmatrix} 1 & & & 0 \\ m_{21} & 1 & & \\ m_{31} & m_{32} & 1 & \\ \vdots & \vdots & \ddots & \ddots \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} \quad 1 \end{pmatrix}$$ となる。

↓ そして、ゴリに良く表えると今の過程は $a_{ij} = (A)_{ij}$ に対し、次のようにまとめられる。

```
for k = 1 to n-1
  for i = k+1 to n
    a_ik ← 代入 a_ik / a_kk   ( = m_ik )
    for j = k+1 to n
      a_ij ← 代入 a_ij - (a_ik)^{m_ik} · a_kj
```

プログラミングがとても簡単！ →

計算量は簡単に見積もれる。

(☆) を作るまでと同じで、$\simeq n^3/3$ である。

という三重ループを回した後にできる行列 $\widetilde{A}$ は、

$$\widetilde{A} = \begin{pmatrix} & U & \\ L & & \end{pmatrix}$$ となる！

(☆) の行列の非ゼロ部分。

L の非対角部分。

だから、この後は

$LUx = b$

$\Leftrightarrow \begin{cases} Ly = b \text{ を解く。} \\ Ux = y \end{cases}$

つまり、$\widetilde{A}$ の a_{ij} を用いて

```
for i = 1 to n
  y_i = b_i
  for j = (i+1) to n
    b_j = b_j - a_ji · b_i
```

として y を求めて、

```
for i = n to 1
  x_i = y_i / a_ii
  for j = (i-1) to 1
    y_j = y_j - (a_ji / a_ii) y_i
```

として x を求める。

(演習)

$$A = \begin{pmatrix} 3 & 3 & -5 & -6 \\ 1 & 2 & -3 & -1 \\ 2 & 3 & -5 & -3 \\ -1 & 0 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$ として、

LU分解を経由して $Ax = b$ を解いてみよ。

手で解けたら、プログラムで解いてみよ。

$$\widetilde{A} = \begin{pmatrix} 3 & 3 & -5 & -6 \\ 1/3 & 1 & -1/3 & 1 \\ 2/3 & 1 & -1/3 & 0 \\ -1/3 & 1 & 1 & -2 \end{pmatrix}$$

となる。

ちなみに、$A^{-1}b = (-5, 0, -2, -1)^T$

注). 上のプログラムの代入部分 $a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$ より分かるように、この計算は本質的に対称性 $(i \leftrightarrow j)$ を保つ。

つまり、$A = LDL^T$, $D = \mathrm{diag}(a_{ii}^{(i)})$ $(\Leftrightarrow U = DL^T)$ と分解できる。これを

(A が対称ならば)

A の LDL^T 分解という。

LU分解の利点.

- 問題 $Ax=b$ に対し、「Aは同じでbのみ異なる」時、主要な計算を1回するのみで良い.
- Aが疎ならば、L, Uもほぼ疎である. (A^{-1}は一般に密).

例). Aが三重対角行列 $\begin{pmatrix} a_1 & b_1 & & & 0 \\ c_1 & a_2 & b_2 & & \\ & c_2 & a_3 & b_3 & \\ & & c_3 & \ddots & b_{n-1} \\ 0 & & & c_{n-1} & a_n \end{pmatrix}$ の時、

$$L=\begin{pmatrix} 1 & & & 0 \\ l_1 & 1 & & \\ & \ddots & \ddots & \\ 0 & & l_{n-1} & 1 \end{pmatrix},\quad U=\begin{pmatrix} u_1 & \tilde{u}_1 & & 0 \\ & u_2 & \ddots & \\ & & \ddots & \tilde{u}_{n-1} \\ 0 & & & u_n \end{pmatrix}$$ となる.

- Aが疎ならば、計算量を減らせる.
- プログラムが簡単.

・Pivot変換

注) 計算途中で $a_{ss}^{(s)}=0$ となって割り算に困る場合は？

(前ページのプログラムで言えば、a_{ik}/a_{kk} の部分)

これは第s～n行の処理をしている時なので、$s\le \ell \le n$ に対し、

$\max_{s\le \ell\le n}|a_{\ell s}^{(s)}|=|a_{\ell^* s}^{(s)}|$ となる ℓ を ℓ^* として、ℓ^*行とs行を交換してしまえばよい.

この操作を(行)Pivot変換という.

$\max_{s\le \ell\le n}|a_{\ell s}^{(s)}|=0$ の場合は？

第s列に意味はないので、何もしなくて良い.

(というか、その時Aは正則じゃないハズ. 停止してしまえ!)

(#32プログラムでいえば、$a_{ik}\xleftarrow{代入} a_{ik}/a_{kk}$ の代わりに $a_{ik}\xleftarrow{代入} 0$ とすれば良い).

コレスキー分解

行列 A が正定値対称行列 ($\Leftrightarrow$ 対称かつ $x^TAx > 0$ for $^\forall x \neq 0$) であれば、対称性より $A = LDL^T$ と分解できる (#32 下とみよ).

また、正定値性より $a_{ii}^{(i)} > 0$ の為 (← 示してみよ!),

$S := \mathrm{diag}(\sqrt{a_{ii}^{(i)}}) \cdot L^T$ とすると、 $A = S^T \cdot S$ と分解できる.

これを A のコレスキー分解という.

$$S = \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ & s_{22} & \cdots & s_{2n} \\ & & \ddots & \vdots \\ 0 & & & s_{nn} \end{pmatrix}$$ であり、上三角. よって.

$a_{ij} = \sum_{k=1}^{i} s_{ki} s_{kj}$ である. よって.

$(i \leq j)$ としてよい.

$i=1$: $a_{11} = s_{11}^2 \Rightarrow s_{11} = \sqrt{a_{11}}$

$a_{12} = s_{11} s_{12} \Rightarrow s_{12} = a_{12}/s_{11}$

$a_{1n} = s_{11} s_{1n} \Rightarrow s_{1n} = a_{1n}/s_{11}$

$i=2$: $a_{22} = s_{12}^2 + s_{22}^2 \Rightarrow s_{22} = \sqrt{a_{22} - s_{12}^2}$

$a_{23} = s_{12}s_{13} + s_{22}s_{23} \Rightarrow s_{23} = (a_{23} - s_{12}s_{13})/s_{22}$

$a_{2n} = s_{12}s_{1n} + s_{22}s_{2n} \Rightarrow s_{2n} = (a_{2n} - s_{12}s_{1n})/s_{22}$

一般の i $a_{ii} = s_{1i}^2 + s_{2i}^2 + \cdots + s_{ii}^2 \Rightarrow s_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} s_{ki}^2}$

$a_{ij} = s_{1i}s_{1j} + \cdots + s_{ii}s_{ij} \Rightarrow s_{ij} = (a_{ij} - \sum_{k=1}^{i-1} s_{ki}s_{kj})/s_{ii}$

$(j > i)$

として、全ての s_{ij} が求まる.

(演習)

$A = \begin{pmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 2 \end{pmatrix}$ をコレスキー分解してみよ.

$S = \begin{pmatrix} \sqrt{2} & 0 & -1/\sqrt{2} \\ 0 & 1 & 0 \\ 0 & 0 & \sqrt{3/2} \end{pmatrix}$

となる.

```
#
# LU 分解を試してみる
# a: 行列, b: ベクトル として，ax = b を解いてみる.
#

include Math

# 行列やベクトルのサイズ
n = 4

# 行列
a = [
  [3.0, 3.0, -5.0, -6.0],
  [1.0, 2.0, -3.0, -1.0],
  [2.0, 3.0, -5.0, -3.0],
  [-1.0, 0, 0, 1.0 ]
]

# 右辺ベクトル
b = [1.0, 2.0, 3.0, 4.0]

#
# LU 分解(ピボット変換してない) ... b に関係なく計算可能.
# ループが三重なので, 計算量が n^3 に比例することがすぐわかる.
#
for k in (1..(n-1))
  for i in (k+1..n)
    a[i-1][k-1] /= a[k-1][k-1]
    for j in (k+1..n)
      a[i-1][j-1] -= a[i-1][k-1]*a[k-1][j-1]
    end
  end
end


#
# 前進代入で y = L^(-1)b を求める.
# ループが二重で済み, 計算量がおおよそ n^2/2 で済むことに注意せよ.
#
y = [0.0, 0.0, 0.0, 0.0]
for i in (1..n)
  y[i-1] = b[i-1]
  if i < n
    for j in ((i+1)..n)
      b[j-1] -= a[j-1][i-1]*b[i-1]
    end
  end
end

#
# 後退代入で x = U^(-1)y を求める. 減っていく数字で for 文を回すのは厄介なので, downto を使っている.
# ループが二重で済み, 計算量がおおよそ n^2/2 で済むことに注意せよ.
#
x = [0.0, 0.0, 0.0, 0.0]
n.downto(1) { |i|
  x[i-1] = y[i-1]/a[i-1][i-1]
  if i > 2
    (i-1).downto(1){ |j|
    y[j-1] -= (a[j-1][i-1]/a[i-1][i-1])*y[i-1]
    }
  end
}

# 結果出力
p x


```

Daisuke Furihata

§3.2.4 反復法

近似解をうしろから改善していく方法

大まかに 旧世代、新世代 に分けられる。

- 旧世代 … Jacobi法, Gauss-Seidel法, SOR法 etc.
- 新世代 … CG法系統

§3.2.4.1 旧世代

§3.2.4.1.1 Jacobi法 (1845, Jacobi)

$A = L + D + R$ （左下三角, 対角, 右上三角）

として、$Ax = b \Leftrightarrow Dx = b - (L+R)x$ より、

$$x^{(m+1)} := D^{-1}\left\{ b - (L+R)x^{(m)} \right\}$$ と反復する方法.

Point: $D^{-1} = \mathrm{diag}\{d^{-1}\}$ なので、計算量はごくわずか.

§3.2.4.1.2 Gauss-Seidel法 (1822, Gauss)

Jacobi法の途中で、既にまだ得ている最新のデータを使うようにしたもの. つまり、

（上の $x^+ = D^{-1}\{b - (L+R)x\}$ と対し、

$(x^+)_i$ の $i: 1 \to n$ と更新していくならば、$(L+R)x$ の Lx の部分は Lx^+ と用いることができる. よって、）

$$x^{(m+1)} := D^{-1}\left\{ b - Lx^{(m+1)} - Rx^{(m)} \right\}$$

$$\Leftrightarrow\ x^{(m+1)} := (D+L)^{-1}\left\{ b - Rx^{(m)} \right\}$$ と反復する方法.

Point: $D+L$ は左下三角行列なので、$(D+L)^{-1}a$ は $(D+L)x = a$ に対して前進代入するだけで求められる.

§3.2.4.1.3 SOR法 Successive Over Relaxation (1950, Young)

GS法の修正を過剰にするもの. $w \geq 1$ として、

$$\begin{cases} a^{(m)} := D^{-1}\left\{ b - Lx^{(m+1)} - Rx^{(m)} \right\} \\ x^{(m+1)} := x^{(m)} + w\left(a^{(m)} - x^{(m)} \right) \end{cases}$$ とする.

$$\Leftrightarrow\ x^{(m+1)} := (D+wL)^{-1}\left[wb - \left\{ wR + (w-1)D \right\} x^{(m)} \right]$$ と反復する.

（計算の手間は GS法と同じ）

Daisuke Furihata

§3.2.4.2 1回世代反復法の性質

Jacobi法, GS法, SOR法について調べてみる.

3つとも $x^{(m+1)} = Mx^{(m)} + Nb$ の形としてはのて,

$f(x) := Mx + Nb$ として, f の挙動を調べてみる.

どんな行列も, ノルムを固有値で評価できるよ.

lemma. 行列 M, $\forall \varepsilon > 0$ に対し,

$\|M\|_\alpha \le \rho(M) + \varepsilon$

を満たす自然なノルム $\|\cdot\|_\alpha$ が存在する.

Proof) M を正則行列 P で Jordan 標準形にしよう. すると.

$$M_1 := P^{-1}MP = \begin{pmatrix} \mu_1 & \alpha_1 & & 0 \\ & \mu_2 & \alpha_2 & \\ & & \ddots & \alpha_{n-1} \\ 0 & & & \mu_n \end{pmatrix}, \quad \alpha_i = 0 \text{ or } 1 \text{ とする.}$$

μ_i : M の固有値

そこで,

$$S := \begin{pmatrix} 1 & & & 0 \\ & \delta & & \\ & & \ddots & \\ 0 & & & \delta^{n-1} \end{pmatrix}$$ として, $M_2 := S^{-1}M_1S$ とすると,

このSを見つけた人はすごい!

$$M_2 = \begin{pmatrix} \mu_1 & \delta\alpha_1 & & 0 \\ & \mu_2 & \delta\alpha_2 & \\ & & \ddots & \delta\alpha_{n-1} \\ 0 & & & \mu_n \end{pmatrix} \quad \text{となる.}$$

そして, $\|x\|_\alpha := \|(PS)^{-1}x\|_2$ とおく. すると,

$$\|Mx\|_\alpha = \|(PS)^{-1}Mx\|_2 = \|\underbrace{S^{-1}P^{-1}MPS}_{M_2}\underbrace{S^{-1}P^{-1}x}_{=:y}\|_2$$

$$= \|M_2 y\|_2 = \sqrt{y^* M_2^* M_2 y} \quad \text{であるが.}$$

$M_2^* M_2 = (D+\delta G)^*(D+\delta G) = D^*D + H$,

$D := \mathrm{diag}(\mu_i)$, $G := \begin{pmatrix} 0 & \alpha_1 & & 0 \\ & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1} \\ 0 & & & 0 \end{pmatrix}$, $H = O(\delta)$ より,

$$\frac{\|Mx\|_\alpha}{\|x\|_\alpha} = \frac{\sqrt{y^*D^*Dy + y^*Hy}}{\sqrt{y^*y}} \le \sqrt{\max_i |\mu_i|^2 + O(\delta)}$$

$$\le \max_i |\mu_i| + \varepsilon \quad \text{for } \forall \varepsilon > 0 \text{ となるように } \delta \text{ をとれる.}$$

∎

Daisuke Furihata

旧世代反復法の本質.

Th.

反復法 $x^{(m+1)} = Mx^{(m)} + Nb$ が真の解 x に収束する

$\Leftrightarrow \rho(M) < 1$.

Proof). $\Rightarrow$ の proof.

$x^{(k)}$ の誤差を $e^{(k)}$ として、 $e^{(k)} = x^{(k)} - x$ とすると、

$e^{(k)} = (Mx^{(k-1)} + Nb) - (Mx + Nb) = Me^{(k-1)} = M^k e^{(0)}$.

で、 $e^{(0)} := u_i$ (固有値 μ_i に対応する固有ベクトル) とすると、

$\|e^{(k)}\| = |\mu_i|^k \|u_i\|$ より、 $\|e^{(k)}\| \xrightarrow{k\to\infty} 0 \Rightarrow |\mu_i| < 1$.

(逆は、固有ベクトルによるベクトル分解の可能性により、ちゃんと内緒)

$\Leftarrow$ の proof.

$\rho(M) < 1$ なので、 $0 < \varepsilon < 1 - \rho(M)$ とれ、前ページの lemma より、

$\|M\|_\alpha \le \rho(M) + \varepsilon < 1$ となる自然なノルム $\|\cdot\|_\alpha$ が存在する. よって、

$\|f(x) - f(y)\|_\alpha = \|M(x-y)\|_\alpha \le \|M\|_\alpha \cdot \|x-y\|_\alpha$

$< \|x-y\|_\alpha$ となるため,

f は縮小写像.

よって、 f によって $x^{(k)}$ は $\|\cdot\|_\alpha$ の意味で収束し、その収束先の一意性より、それは x である. (#14 を見よ) ▨

以下、各方法の $\rho(M)$ を見てみよう.

Jacobi 法の $\rho(M)$.

$M = -D^{-1}(L+R)$. これについては、下の Th. より、 A が優対角ならば良い.

Th. A が優対角行列 $\Rightarrow \rho(D^{-1}(L+R)) < 1$.

proof) $M = -D^{-1}(L+R) = \{m_{ij}\}$ とすると、 $m_{ij} = \begin{cases} 0 & : i=j \\ -\frac{a_{ij}}{a_{ii}} & : i \neq j \end{cases}$ で、

M の固有値 μ に対し、

Görshgorin's Th. より $|\mu - \overset{0}{m_{ii}}| \le \sum_{j\neq i} |m_{ij}|$ を満たすものが存在する.

で、 $\sum_{j\neq i} |m_{ij}| = \frac{1}{|a_{ii}|} \sum_{j\neq i} |a_{ij}| < 1$ より、 $|\mu| < 1$.

(優対角性による) ▨

Daisuke Furihata

Gauss-Seidel の $\rho(M)$.

$M = -(D+L)^{-1}R$ で、下の Th. をみよう.

Th A が優対角なら、$\rho((D+L)^{-1}R) < 1$.

Proof) $M = -(D+L)^{-1}R$ の固有値対を (μ, x) として、x の成分の絶対値で最大のものを x_k とする. $(x_k \neq 0)$

$Mx = \mu x \Leftrightarrow Rx = -\mu(D+L)x$ の第 k 成分をみると、

$$\sum_{j>k} a_{kj}x_j = -\mu\left(a_{kk}x_k + \sum_{j<k} a_{kj}x_j\right)$$

$$\Leftrightarrow \mu a_{kk} = -\sum_{j>k} a_{kj}\frac{x_j}{x_k} - \mu\sum_{j<k} a_{kj}\frac{x_j}{x_k}$$

$$\Rightarrow |\mu|\cdot|a_{kk}| \leq \sum_{j>k}|a_{kj}| + |\mu|\cdot\sum_{j<k}|a_{kj}| \quad \sim(*).$$

仮に $|\mu| \geq 1$ とおくと

$$(*) \Rightarrow |\mu|\cdot|a_{kk}| \leq |\mu|\cdot\sum_{j\neq k}|a_{kj}| \Leftrightarrow |a_{kk}| \leq \sum_{j\neq k}|a_{kj}|$$

となり、A が優対角であることに矛盾する. よって $|\mu| < 1$. ▨

Th A, D が実正値対称 $\Rightarrow \rho((D+L)^{-1}R) < 1$.

Proof) この条件は、$A = L+D+R$ に対し、$L^T = R$ $(\Leftrightarrow L = R^T)$ で、(μ, x) を M の固有値対として、$G := L+D$ として $A = G+R$ とおく.

$Mx = \mu x \Leftrightarrow Rx = -\mu Gx$ より、

$Ax = (G+R)x = (1-\mu)Gx$ で、A の正定値性より

$1-\mu \neq 0$. よって、$Gx = \frac{1}{1-\mu}Ax$ とできる. そして、

$$x^*Dx = x^*(G-L)x = x^*Gx - x^*Lx = x^*Gx - x^*R^Tx$$

$$= x^*Gx - (Rx)^*x = x^*Gx + \overline{\mu}(Gx)^*x$$

$$= \frac{1}{1-\mu}x^*Ax + \overline{\mu}\frac{1}{1-\mu}\cdot x^*Ax = \frac{1-|\mu|^2}{|1-\mu|^2}x^*Ax$$

で、A と D の正定値性より、$1-|\mu|^2 > 0 \Leftrightarrow |\mu| < 1$. ▨

SOR法の $\rho(M)$.

$M = -(D+\omega L)^{-1}\{\omega R + (\omega-1)D\}$ に対して.

Th

A, Dが正定値対称 & $(D+\omega L)$が正則 & $0<\omega<2$ $\Rightarrow \rho(M)<1$.

Proof). Mの固有対を(μ, x)とすると,

$$\{\omega R+(\omega-1)D\}x = -\mu(D+\omega L)x$$

$$\{\omega(D+R)-D\}x \qquad -\mu\{D+\omega(A-D-R)\}x$$

$$\{\omega(A-L)-D\}x \qquad -\mu\{(1-\omega)D-\omega R\}x - \mu\omega Ax$$

$$\omega Ax - (D+\omega L)x \qquad \Updownarrow$$

$$\Updownarrow \qquad (1-\mu)\{\omega R+(\omega-1)D\}x = -\mu\omega Ax \quad \sim ②$$

$$(1-\mu)(D+\omega L)x = \omega Ax \quad \sim ①$$

$$①^T x = (1-\mu)\, x^T(D+\omega R)x = \omega x^T Ax$$

$$-)\ x^T ② = (1-\mu)\, x^T\{\omega R+(\omega-1)D\}x = -\mu\omega\, x^T Ax$$

$$(1-\mu)(2-\omega)\, x^T Dx = (1+\mu)\,\omega x^T Ax$$

$1-\bar{\mu}$をかけて

$$\Rightarrow \underbrace{|1-\mu|^2(2-\omega)}_{\text{実・正}}\underbrace{x^T Dx}_{\text{実・正}} = \underbrace{(1-|\mu|^2)}_{\text{実}}\underbrace{\omega x^T Ax}_{\text{実・正}} + \underbrace{(\mu-\bar{\mu})}_{\text{純虚数}}\underbrace{\omega x^T Ax}_{\text{実・正}}$$

$$\Rightarrow 1-|\mu|^2>0 \ \&\ \mu\in\mathbb{R} \iff -1<\mu<1. \qquad \square$$

条件をつけないと？

Th. $\rho(M) \geq |\omega-1|$ としかいえない.

Proof) Mの固有値を $\lambda_1, \lambda_2, \cdots \lambda_n$ とすると,

$|M| = \prod_i \lambda_i$. であるが,

$$|M| = \left|-(D+\omega L)^{-1}\{\omega R+(\omega-1)D\}\right| = \left|(D+\omega L)^{-1}\right| \cdot \left|(1-\omega)D-\omega R\right|$$

$$= |D^{-1}|\cdot|(1-\omega)D-\omega R| = |(1-\omega)I-\omega D^{-1}R| = |(1-\omega)I| = (1-\omega)^n \text{ より},$$

$$\rho(M) = \max_i |\lambda_i| \geq \{(1-\omega)^n\}^{1/n} = |1-\omega|. \qquad \square$$

Point. ところで, SOR法の ω はいくつにすべきなのか？ … 実は難しい. 1より「ちょっと」大きくするのが通例ではあるが…

Daisuke Furihata

知っていると得するオマケ.

得た近似解を、もう少し改善する.

$Ax = b$ に対し、近似解 $\tilde{x}$ を得た、とする. 誤差は $e = \tilde{x} - x$.

この時、$A\tilde{x} = A(x+e) = b + Ae \Leftrightarrow Ae = A\tilde{x} - b$ なので、

① $Ae = \underbrace{A\tilde{x} - b}_{\text{既知}}$ を解いて e を得て、

② $\tilde{\tilde{x}} := \tilde{x} - e$ とすると、$\tilde{\tilde{x}}$ は $\tilde{x}$ よりも「マシ」な可能性が高い

Point. $A\tilde{x} - b \simeq 0$ の時は、もう充分に $\tilde{x}$ の精度が良い.

Point2 LU分解を使っていれば、①の計算量は $O(n^2)$ で済む.

(演習)

1. $A = \begin{pmatrix} 7 & 1 & 2 & 3 \\ 1 & 10 & 3 & 4 \\ 2 & 3 & 13 & 6 \\ 3 & 4 & 6 & 16 \end{pmatrix}$, $b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$ として、$Ax = b$ を Jacobi法、GS法、SOR法で解いてみよ. ($\omega = 1.2$ としておく)

2. 上の A に対し、$\rho(M)$ を求めよ for Jacobi法、GS法、SOR法. ただし、$\rho(M)$ の計算法はまだ知らないので、Mathematica etc. を用いてよい.

```ruby
#
# 連立一次方程式に対し, 様々な(旧世代)反復法を試してみる
# a: 行列, b: ベクトル として,  ax = b を解いてみる.
#

include Math
require 'pp'

# 許せる誤差(ノルムで)
$eps = 1.0e-08

# 行列やベクトルのサイズ
$n = 4

# 行列
$a = [
  [7.0, 1.0, 2.0, 3.0],
  [1.0, 10.0, 3.0, 4.0],
  [2.0, 3.0, 13.0, 6.0],
  [3.0, 4.0, 6.0, 16.0 ]
]


# 右辺ベクトル
$b = [1.0, 2.0, 3.0, 4.0]

# l, d, r を作る. 一旦, 0 だけが並んでいるものを作ってから, 修正しよう.
$l = Array.new($n).collect{ Array.new($n).collect{0.0} }
$d = Array.new($n).collect{ Array.new($n).collect{0.0} }
$r = Array.new($n).collect{ Array.new($n).collect{0.0} }

for i in (1..$n)
  for j in (1..$n)
    v = $a[i-1][j-1]
    if (i==j) then
       $d[i-1][j-1] = v
    elsif (i<j) then
       $r[i-1][j-1] = v
    else
       $l[i-1][j-1] = v
    end
  end
end

# 残差 b - Ax を計算する
def residual(x)
  y = Array.new($n).collect{0.0}
  for i in (1..$n)
    y[i-1] = $b[i-1]
    for j in (1..$n)
      y[i-1] -= $a[i-1][j-1]*x[j-1]
    end
  end

  return y
end


# 1-ノルム
def norm_one(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1].abs
  end
  return v
end

# 2-ノルム
def norm_two(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1]**2
  end
  return sqrt(v)
end

# sup ノルム
def norm_sup(x)
  v = 0.0
  for i in (1..$n)
    if (x[i-1].abs > v) then
      v = x[i-1].abs
    end
  end
  return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
  return norm_two(x)
end

# 収束判定
def is_convergence(x)
  if ( norm(residual(x)) < $eps) then
     return true
    else
     return false
  end
end

# 見易く出力
def print_result(x)
  print "x = ["
  for i in (1..$n-1)
    printf("%.6f",x[i-1])
    print ", "
  end
  printf("%.6f",x[$n-1])
  print "], "
  print "residual= ",norm(residual(x)),"¥n"
end


######################
# Jacobi iteration
def jacobi(x)
```

```
117
118   # y= (L+R)x を計算
119   y = Array.new($n).collect{0.0}
120   for i in (1..$n)
121     for j in (1..$n)
122     y[i-1] += ($l[i-1][j-1] + $r[i-1][j-1])*x[j-1]
123     end
124   end
125
126   # z = D^(-1){ b - y } を計算
127   z = Array.new($n).collect{0.0}
128   for i in (1..$n)
129     z[i-1] = ($b[i-1] - y[i-1])/$d[i-1][i-1]
130   end
131
132   # 結果出力
133   return z
134
135 end
136
137 ##########################
138 # 実際に計算!
139
140 # 近似解の初期値. ゼロはまずいので, 1としておく.
141 x = Array.new($n).collect{ 1.0 }
142 print_result(x)
143
144
145 # ダミー変数
146 y = Array.new($n).collect{ 0.0 }
147
148 # 収束するまでループ
149 loop_times = 0
150 loop do
151   if is_convergence(x) then
152       break
153     else
154       y = jacobi(x)
155       print_result(y)
156       x = y
157       loop_times += 1
158     end
159 end
160
161 # 結果出力
162 print "¥n¥nYou obtain a numerical solution after ",loop_ti
    mes," loops.¥n¥n"
163 print_result(x)
164 print "¥n"
```

```
#
# 連立一次方程式に対し, 様々な(旧世代)反復法を試してみる
# a: 行列, b: ベクトル として,  ax = b を解いてみる.
#

include Math
require 'pp'

# 許せる誤差(ノルムで)
$eps = 1.0e-08

# 行列やベクトルのサイズ
$n = 4

# 行列
$a = [
  [7.0, 1.0, 2.0, 3.0],
  [1.0, 10.0, 3.0, 4.0],
  [2.0, 3.0, 13.0, 6.0],
  [3.0, 4.0, 6.0, 16.0 ]
]


# 右辺ベクトル
$b = [1.0, 2.0, 3.0, 4.0]

# l, d, r を作る. 一旦, 0 だけが並んでいるものを作ってから, 修正しよう.
$l = Array.new($n).collect{ Array.new($n).collect{0.0} }
$d = Array.new($n).collect{ Array.new($n).collect{0.0} }
$r = Array.new($n).collect{ Array.new($n).collect{0.0} }

for i in (1..$n)
  for j in (1..$n)
    v = $a[i-1][j-1]
    if (i==j) then
       $d[i-1][j-1] = v
    elsif (i<j) then
       $r[i-1][j-1] = v
    else
       $l[i-1][j-1] = v
    end
  end
end

# 残差 b - Ax を計算する
def residual(x)
  y = Array.new($n).collect{0.0}
  for i in (1..$n)
    y[i-1] = $b[i-1]
    for j in (1..$n)
      y[i-1] -= $a[i-1][j-1]*x[j-1]
    end
  end

  return y
end


# 1-ノルム
def norm_one(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1].abs
  end
  return v
end

# 2-ノルム
def norm_two(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1]**2
  end
  return sqrt(v)
end

# sup ノルム
def norm_sup(x)
  v = 0.0
  for i in (1..$n)
    if (x[i-1].abs > v) then
      v = x[i-1].abs
    end
  end
  return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
  return norm_two(x)
end

# 収束判定
def is_convergence(x)
  if ( norm(residual(x)) < $eps) then
     return true
    else
     return false
  end
end

# 見易く出力
def print_result(x)
  print "x = ["
  for i in (1..$n-1)
    printf("%.6f",x[i-1])
    print ", "
  end
  printf("%.6f",x[$n-1])
  print "], "
  print "residual= ",norm(residual(x)),"¥n"
end


######################
# Gauss-Seidel iteration
def gs(x)
```

```
117
118   # y= b-Rx を計算
119   y = Array.new($n).collect{0.0}
120   for i in (1..$n)
121     y[i-1] = $b[i-1]
122     for j in (1..$n)
123     y[i-1] -= $r[i-1][j-1]*x[j-1]
124     end
125   end
126
127   # 前進代入で z = (D+L)^(-1) y を計算
128   z = Array.new($n).collect{0.0}
129   for i in (1..$n)
130     z[i-1] = y[i-1]/$d[i-1][i-1]
131     if i < $n
132       for j in ((i+1)..$n)
133         y[j-1] -= ($l[j-1][i-1]/$d[i-1][i-1])*y[i-1]
134       end
135     end
136   end
137
138   # 結果出力
139   return z
140
141 end
142
143 ##########################
144 # 実際に計算!
145
146 # 近似解の初期値. ゼロはまずいので, 1としておく.
147 x = Array.new($n).collect{ 1.0 }
148 print_result(x)
149
150
151 # ダミー変数
152 y = Array.new($n).collect{ 0.0 }
153
154 # 収束するまでループ
155 loop_times = 0
156 loop do
157   if is_convergence(x) then
158       break
159     else
160       y = gs(x)
161       print_result(y)
162       x = y
163       loop_times += 1
164     end
165 end
166
167 # 結果出力
168 print "¥n¥nYou obtain a numerical solution after ",loop_ti
    mes," loops.¥n¥n"
169 print_result(x)
170 print "¥n"
```

```ruby
#
# 連立一次方程式に対し, 様々な(旧世代)反復法を試してみる
# a: 行列, b: ベクトル として,  ax = b を解いてみる.
#

include Math
require 'pp'

# 許せる誤差(ノルムで)
$eps = 1.0e-08

# 行列やベクトルのサイズ
$n = 4

# 行列
$a = [
  [7.0, 1.0, 2.0, 3.0],
  [1.0, 10.0, 3.0, 4.0],
  [2.0, 3.0, 13.0, 6.0],
  [3.0, 4.0, 6.0, 16.0 ]
]


# 右辺ベクトル
$b = [1.0, 2.0, 3.0, 4.0]

# l, d, r を作る. 一旦, 0 だけが並んでいるものを作ってから, 修正しよう.
$l = Array.new($n).collect{ Array.new($n).collect{0.0} }
$d = Array.new($n).collect{ Array.new($n).collect{0.0} }
$r = Array.new($n).collect{ Array.new($n).collect{0.0} }

for i in (1..$n)
  for j in (1..$n)
    v = $a[i-1][j-1]
    if (i==j) then
       $d[i-1][j-1] = v
    elsif (i<j) then
       $r[i-1][j-1] = v
    else
       $l[i-1][j-1] = v
    end
  end
end

# 残差 b - Ax を計算する
def residual(x)
  y = Array.new($n).collect{0.0}
  for i in (1..$n)
    y[i-1] = $b[i-1]
    for j in (1..$n)
      y[i-1] -= $a[i-1][j-1]*x[j-1]
    end
  end

  return y
end


# 1-ノルム
def norm_one(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1].abs
  end
  return v
end

# 2-ノルム
def norm_two(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1]**2
  end
  return sqrt(v)
end

# sup ノルム
def norm_sup(x)
  v = 0.0
  for i in (1..$n)
    if (x[i-1].abs > v) then
      v = x[i-1].abs
    end
  end
  return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
  return norm_two(x)
end

# 収束判定
def is_convergence(x)
  if ( norm(residual(x)) < $eps) then
     return true
    else
     return false
  end
end

# 見易く出力
def print_result(x)
  print "x = ["
  for i in (1..$n-1)
    printf("%.6f",x[i-1])
    print ", "
  end
  printf("%.6f",x[$n-1])
  print "], "
  print "residual= ",norm(residual(x)),"¥n"
end


######################
# SOR 法
$w = 1.2
```

```
117
118 def sor(x)
119
120   # y= wb-(wR+(w-1)D)x を計算
121   y = Array.new($n).collect{0.0}
122   for i in (1..$n)
123     y[i-1] = $w * $b[i-1]
124     for j in (1..$n)
125     y[i-1] -= ($w*$r[i-1][j-1] +($w-1.0)*$d[i-1][j-1])*x[j-
    1]
126     end
127   end
128
129   # 前進代入で z = (D+wL)^(-1) y を計算
130   z = Array.new($n).collect{0.0}
131   for i in (1..$n)
132     z[i-1] = y[i-1]/$d[i-1][i-1]
133     if i < $n
134       for j in ((i+1)..$n)
135         y[j-1] -= ($w*$l[j-1][i-1]/$d[i-1][i-1])*y[i-1]
136       end
137     end
138   end
139
140   # 結果出力
141   return z
142
143 end
144
145 ########################
146 # 実際に計算!
147
148 # 近似解の初期値. ゼロはまずいので, 1としておく.
149 x = Array.new($n).collect{ 1.0 }
150 print_result(x)
151
152
153 # ダミー変数
154 y = Array.new($n).collect{ 0.0 }
155
156 # 収束するまでループ
157 loop_times = 0
158 loop do
159   if is_convergence(x) then
160       break
161     else
162       y = sor(x)
163       print_result(y)
164       x = y
165       loop_times += 1
166     end
167 end
168
169 # 結果出力
170 print "¥n¥nYou obtain a numerical solution after ",loop_ti
    mes," loops.¥n¥n"
171 print_result(x)
172 print "¥n"
```

§3.2.4.2
逐次最小化法

・新世代反復法 の 祖先. 後述の CG法 を含む
(CG法の「古い説明」といっても良いか)

向 $Ax=b$ に対し、A が正定値対称であるとする. この時、

$S(y) := \frac{1}{2}(x-y)^T A(x-y)$ とすると、A の正定値性より

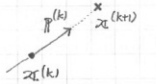
$$S(y)=\begin{cases} \text{正} & : y \neq x \\ 0 & : y = x \end{cases}$$

であり、また当然ながら $y \to x$ で S は小さくなる.

よって、$S(y)$ を小さくするように y を改善していく反復法 (最終的に $S=0$ を目指す) が考えられ、これを一般に逐次最小化法 (総称) という.

逐次最小化法のアルゴリズム

$x^{(0)} \to x^{(1)} \to x^{(2)} \to \cdots \to x^{(k)} \to$ と近似解を改善していくとして、



・$x^{(k)}$ からの修正方向 $p^{(k)}$ を決める.
(どうやって? → 後々)

・$\min_{\omega} S(x^{(k)}+\omega p^{(k)})$ を満たすと思われる ω を用いて、
$x^{(k+1)} = x^{(k)} + \omega p^{(k)}$ とする.

この $p^{(k)}$ の決め方が実は Point になる.
これによって個々の解法の性質が決まる.

というアルゴリズムで修正していく.
この ω は次のように決められる. $\min_{\omega} S$ を満たすのだから $\frac{\partial S}{\partial \omega}=0$ であるので、

$$0 = \frac{\partial}{\partial \omega}\left\{\frac{1}{2}(x-x^{(k)}-\omega p^{(k)})^T A(x-x^{(k)}-\omega p^{(k)})\right\} = \omega p^T A p - p^T A(x-x^{(k)})$$

$$\Leftrightarrow \omega = \frac{p^T(b-Ax^{(k)})}{p^T A p} = \frac{p^{(k)} \cdot r^{(k)}}{p^{(k)} \cdot A p^{(k)}} \quad \text{where } r^{(k)} := b - Ax^{(k)}.$$

(A は正定値なので、$p^{(k)} \neq 0$ ならば 分母 $\neq 0$)

・この $r^{(k)}$ を残差ベクトルという.

S は小さくなっていくか?

$$\begin{aligned} S(x^{(k+1)}) &= S(x^{(k)}+\omega p^{(k)}) = \frac{1}{2}(x-x^{(k)}-\omega p^{(k)})^T A(x-x^{(k)}-\omega p^{(k)}) \\ &= \frac{1}{2}(x-x^{(k)})^T A(x-x^{(k)}) - \frac{1}{2}\cdot 2\omega p^T A(x-x^{(k)}) + \frac{1}{2}\omega^2 p^T A p \\ &= S(x^{(k)}) + \frac{1}{2}\omega(\omega p^T A p - 2 p\cdot r) = S(x^{(k)}) + \frac{1}{2}\omega(-p\cdot r) \\ &= S(x^{(k)}) - \frac{(p\cdot r)^2}{2 p^T A p} \end{aligned}$$

となるので、S は必ず小さくなっていく.

この性質が保証されているのは大きい.

注). $S(y)$ に対して $\frac{\partial S}{\partial y_k} = (A(y-x))_k = -(b-Ay)_k$ であるので、勾配 $\nabla S = -(b-Ay)$ である. つまり

$r^{(k)} = -\nabla S(x^{(k)})$ となる. つまり、残差ベクトル $r^{(k)}$ は $S(x^{(k)})$ の最急降下方向になっている.

注2). (当然) 探索方向 $p^{(k)}$ と、新しい点 $x^{(k+1)}$ での勾配 $\nabla S(x^{(k+1)})$ は直交する. つまり、$p^{(k)} \perp r^{(k+1)}$.
(↑ $p^{(k)} \cdot r^{(k+1)}$ を計算してみよ)
(宿題)

§3.2.4.2.1
(最急降下法)

逐次最小化法の最急降下法

(#55 下欄の注意より) $r^{(k)}$ は $S(x^{(k)})$ の最急降下方向であるのでこれを $p^{(k)}$ と採用して、

$p^{(k)} = b - Ax^{(k)}$ とする方法をいう。

利点) ・(もともとだが) S は減少していく。
・簡単

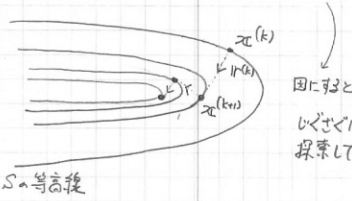
欠点) ・$x^{(k)} \simeq x$ となると、$Ax^{(k)} \simeq b$ の為、$b - Ax^{(k)}$ で桁落ちが起きて、方向 $p^{(k)}$ が「アヤシク」なる。→ 当然、S の減少性もアヤシクなる。

Q. この場合、$p^{(k+1)} \perp p^{(k)}$ となることを示しておけ。

・$p^{(k+1)} \perp p^{(k)}$ は成り立つが、一般に $p^{(k+2)} \perp p^{(k)}$ は成り立たない。
(つまり、探索方向とムダはいくらでも生じうる)

$$x^{(k+1)} = x^{(k)} + \omega_k p^{(k)}, \quad x^{(k+2)} = x^{(k)} + \omega_k p^{(k)} + \omega_{k+1} p^{(k+1)},$$
$$x^{(k+3)} = x^{(k)} + \omega_k p^{(k)} + \omega_{k+1} p^{(k+1)} + \omega_{k+2} p^{(k+2)}, \quad \cdots$$

となるので、$p^{(k)} \propto p^{(k+2)}$ となったらすごいムダ。



図にするとこんな感じ。
じぐざぐに同じ方向へ何度も探索していく。

注) 一般に、最急降下法/最急勾配法 とよばれるものの効率はよくない。
(彼ばる解えは 必ずしもベストではない、ということ)
局所的に　大域的には

Daisuke Furihata

§3.2.4.2.2
Conjugate Gradient
共役勾配法

(1952, Hestenes & Stiefel)

- NBS (National Bureau of Standards) による プロジェクトの成果.

(まずは 古いタイプの説明から) 以降, A は $n\times n$ 正定値対称, としておく.

(準備) A共役 or A直交 とは, $p\cdot Aq=0$ の時の p と q の関係をいう.

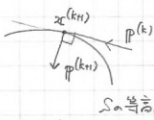
Lemma $\{p_i\neq 0\}_{i=1}^{n}$ が 互いに A直交 ならば, 一次独立である.

proof) p_i の一次結合を $x=\sum_{i=1}^{n} c_i p_i$ と書く.

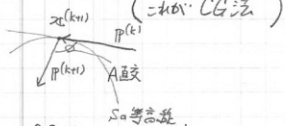
$x\cdot Ap_k = c_k \underbrace{p_k^T A p_k}_{\text{正}}$ であるので, $x=0 \Leftrightarrow \forall c_k=0$ ▨

(つまり, 通常の直交性の他に, A直交性で考えてもベクトル空間の張り方は 1つは大丈夫そうだ.)

最急降下法では $p^{(k+1)}$ と $p^{(k)}$ は直交していたけれども, これを A直交に替えてみては?



最急降下法



CG法

↓

これが, とんでもなく良い性質をもっていることが判明! (後述)

(さて, ナレ良, 7) **CG法のアルゴリズム** … 逐次最小化法の一種として,

探索方向 p を定めるのに,

$p^{(k)} = r^{(k)} + \beta_{k-1} p^{(k-1)}$ として, $p^{(k)}$ と $p^{(k-1)}$ が A直交であるように β を決める.

(ここまでだけなら 最急降下法と同じ)

つまり, $(r^{(k)}+\beta_{k-1}p^{(k-1)})\cdot Ap^{(k-1)}=0 \Leftrightarrow \beta_{k-1} = -\dfrac{r^{(k)}\cdot Ap^{(k-1)}}{p^{(k-1)T}Ap^{(k-1)}}$

- p をどう決めても, S は減っていくことに注意.

後で使う. →

Lemma. $\{p^{(i)}\}_{i=1}^{k}$ が 互いに A直交 ならば, $\forall p^{(i)} \perp r^{(k+1)}$.

proof). $x^{(k+1)} = x^{(i+1)} + \sum_{j=i+1}^{k} \omega_j p^{(j)}$ より, $r^{(k+1)} = b - Ax^{(k+1)} = r^{(i+1)} - \sum_{j=i+1}^{k} \omega_j Ap^{(j)}$.

よって, $p^{(i)}\cdot r^{(k+1)} = \underbrace{p^{(i)}\cdot r^{(i+1)}}_{\#\text{§3.3 下欄の注2 より,}} - \sum_{j=i+1}^{k} \omega_j \cdot p^{(i)}\cdot Ap^{(j)} = 0$. ▨

Daisuke Furihata

定理 CG法において、$i \neq j$ に対し

$$\begin{cases} r^{(i)} \cdot r^{(j)} = 0, \\ p^{(i)} \cdot Ap^{(j)} = 0 \end{cases}$$

が成り立つ！

すごい定理である！

proof) 帰納法を使う。 $i<j \leq k$ に対し、上の定理が成立していると仮定する。

#43の2つめのlemmaより、 $0 = p^{(i)} \cdot r^{(k+1)}$ for $\forall i \leq k$ であるが、

$p^{(i)} \cdot r^{(k+1)} = \left(r^{(i)} + \beta_{i-1} p^{(i-1)}\right) \cdot r^{(k+1)} = r^{(i)} \cdot r^{(k+1)}$ であるので、

$r^{(i)} \cdot r^{(j)} = 0$ が $i<j \leq k+1$ に対して成り立つ。

次に、 $i \leq k$ に対し、

$$p^{(i)} \cdot Ap^{(k+1)} = \begin{cases} 0 & : i = k \leftarrow \text{CG法のアルゴリズムそのもの。} \\ (\text{以下計算}) & : i \leq k-1 \end{cases} \text{について、}$$

$$p^{(i)} \cdot Ap^{(k+1)} = p^{(i)} \cdot A\left(r^{(k+1)} + \beta_k p^{(k)}\right) = p^{(i)} \cdot Ar^{(k+1)}$$

$$= \frac{1}{\omega_i}\left(x^{(i+1)} - x^{(i)}\right) \cdot Ar^{(k+1)} = \frac{-1}{\omega_i}\left(r^{(i+1)} - r^{(i)}\right) \cdot r^{(k+1)}$$

$$= 0 \quad \text{となる} \left(\omega_i \neq 0 \text{の場合}\right)$$

ただし、 $\omega_i = 0$ の場合は、#45より $p^{(i)} \cdot r^{(i)} = 0$ 。この時、

$p^{(i)} \cdot r^{(i)} = \left(r^{(i)} + \beta_{i-1} p^{(i-1)}\right) \cdot r^{(i)} = r^{(i)} \cdot r^{(i)}$ より、これは $r^{(i)} = 0$ と意味する（つまり、 $x^{(i)}$ は真の解 x に一致している！）。この時、 $\beta_{i-1} = 0$ となので、

$p^{(i)} = r^{(i)} + \beta_{i-1} p^{(i-1)} = 0 + 0 \cdot p^{(i-1)} = 0$ となり、反復は停止するので、

$\omega_i = 0$ の場合は $x^{(i+1)}$ 以降のギロンは不要である。 ▨

↓

この定理がどうすごいのか？

- 局所的なA直交性を $p^{(i)}$ に課しただけなのに、$\{p^{(i)}\}$ は（i 全体にわたる）大域的なA直交性をもつ。

  $\Rightarrow \{p^{(i)}\}_{i=0}^{n-1}$ は互いに一次独立。つまり、$\mathbb{R}^n$ を張れてしまう。

  $\Rightarrow$ 少なくとも $p^{(n)} = 0$.

  $\Leftrightarrow$ 〃 $r^{(n)} = 0$

  $\Leftrightarrow$ 〃 $Ax^{(n)} = b$. であるので、

  （こうならないのは、それより前に $p = 0$ となる時、つまり、解が「早めに」求まってしまった場合。）

とんでもない結果である！

（理論的には）CG法は n 回反復以内に近似解が真の解に一致する！

おそるべし。

注) 実際には、丸めerr.の影響があるので こんなに正確にはいかないが。

Daisuke Furihata

・CG法アルゴリズムの書き直し

CG法アルゴリズム中の ω_i, β_i をもう少し丁寧にみると、
まず #58 の 1h の proof からわかるように $p^{(i)} \cdot r^{(i)} = r^{(i)} \cdot r^{(i)}$ なので、

$$\omega_i = \frac{p^{(i)} \cdot r^{(i)}}{p^{(i)} \cdot A p^{(i)}} = \frac{r^{(i)} \cdot r^{(i)}}{p^{(i)} \cdot A p^{(i)}} \quad \text{と書ける.}$$

次に、$r^{(i+1)} = b - A x^{(i+1)} = b - A(x^{(i)} + \omega_i p^{(i)}) = r^{(i)} - \omega_i A p^{(i)}$

$\Leftrightarrow \quad A p^{(i)} = -\frac{1}{\omega_i}(r^{(i+1)} - r^{(i)})$ を用いて、

$$\beta_i = -\frac{r^{(i+1)} \cdot A p^{(i)}}{p^{(i)} \cdot A p^{(i)}} = \frac{1}{\omega_i} \frac{r^{(i+1)} \cdot r^{(i+1)}}{p^{(i)} \cdot A p^{(i)}} = \frac{r^{(i+1)} \cdot r^{(i+1)}}{r^{(i)} \cdot r^{(i)}} \quad \text{と書ける.}$$

つまり、$R_i := r^{(i)} \cdot r^{(i)}$, $Q_i := p^{(i)} \cdot A p^{(i)}$ とすると $\omega_i = \frac{R_i}{Q_i}$, $\beta_i = \frac{R_{i+1}}{R_i}$
と書ける.

まとめると？

CG法のアルゴリズム（まとめ版）

初期化

① 適当に $x^{(0)} \neq 0$ ととる.
② $r^{(0)} = b - A x^{(0)}$ とする.
③ $p^{(0)} = r^{(0)}$ 〃
④ $R_0 = r^{(0)} \cdot r^{(0)}$ 〃
⑤ $q^{(0)} = A p^{(0)}$ 〃

ループ計算 ： $k = 0, 1, 2, 3, \cdots$ と増やしつつ、以下の計算を行う.

① $Q_k = p^{(k)} \cdot q^{(k)}$ を求める.
② $\omega_k = R_k / Q_k$ 〃
③ $x^{(k+1)} = x^{(k)} + \omega_k p^{(k)}$ 〃
④ $r^{(k+1)} = r^{(k)} - \omega_k q^{(k)}$ 〃
⑤ $\| r^{(k+1)} \| < \varepsilon \| b \|$ ならば終了する. ← $\varepsilon > 0$ は小さくとる.
⑥ $R_{k+1} = r^{(k+1)} \cdot r^{(k+1)}$ を求める.
⑦ $\beta_k = R_{k+1} / R_k$ 〃
⑧ $p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}$ 〃
⑨ $q^{(k+1)} = A p^{(k+1)}$ 〃

などとすれば良い.

計算量をみつもってみよう. 面倒なので積の回数のみみるとして、ループ計算では
① n ② 1 ③ n ④ n ⑤ n ⑥ n ⑦ 1 ⑧ n ⑨ n^2 → 計 $n^2 + 5n + 2 \simeq n^2$ 回/ループ で済み、
（理論的には）反復は n 回以下なので、全部で n^3 回以下 程度ですむかもしれない.
しかも、⑨の n^2 は、A が疎ならば n 程度で済むことも多い. その時は 全部で n^2 程度で済んでしまう.

Daisuke Furihata

CG法のもう1つの解釈

準備

現代的解法の発展はココから始まった。

- クリロフ Krylov 空間 $K_s(A;u)$ ($s \le n$) : $\mathrm{span}\{u, Au, A^2u, \cdots, A^{s-1}u\}$ が s 次元をもつ時、これを s 次元クリロフ部分空間 $K_s(A;u)$ とよぶ。

クリロフ部分空間反復法 : $Ax = b$ の求解問題に対し、

$$x_k := x_0 + z_k, \quad z_k \in K_k(A;r_0) \qquad (k \ge 1)$$
$$(r_0 := b - Ax_0)$$

という条件のもとに近似解 x_k を改善していく解法を一般にこう言う。

↓ どうやって z_k を決める？

「理想的な状態」とよばれる次の2条件を考える。これかあれば良い。

条件1) ・最小条件 minimal residual cond. : $r_k = \min_{x \in x_0 + K_k(A;r_0)} \| b - Ax \|_\alpha$

・直交条件 orthogonal 〃 : $r_k \perp_\alpha K_k(A;r_0)$

} のどちらかが成立する。

条件2) 解の更新時に、過去の情報全てを参照しなくて良い。

- $K_n(A;r_0)$ で n 次元空間そのものになってしまうので、この条件1の中の2つの条件はいずれも「n 回以内で反復が終了する」ことを保証する

↓ しかし

この2条件を同時に満たせるのは (実質的に) A = エルミート行列 の時のみ。

Q. (実質的に)、ってどういうこと？

A. 実はエルミートでなくとも良い。本当は、
$A = \alpha T + \beta I$, $\alpha, \beta \in \mathbb{C}$,
$\exists$ 正値エルミート B に対し
$TB = (TB)^*$.
であればよいことがわかっている。
(Ashby, Manteuffel, Saylor 1990).

⇔ 一般の A に対しては、条件1 or 2を「あきらめる」

↙ 結局、(現状は) 以下のようになっている (1995年頃)

条件1 \ 条件2	○	×	
○	(A = エルミート) CG法.	(最小条件) GMRES (→GCR, orthomin)	(直交条件) orthores
×	(条件1を 双直交性 $r_k \perp_\alpha K_k(A^*, r_0^*)$ で代用) Bi-CG法系統		

・右図のうち、CG系しか授業では扱わないわけだが。

(準備)

Arnoldi Process
・アーノルディ過程 : クリロフ部分空間の(正規)直交基底 $\{u_1, \cdots, u_s\}$ を作る方法.
(要するに Gram-Schmidt)

$K_s(A;u)$ に対し、

$$\begin{cases} u_1 := \mu_1 u, \ \mu_1 \neq 0 \\ \text{for } k=1 \text{ to } (s-1) \\ \quad u_{k+1} := \mu_{k+1}\left\{ Au_k - \sum_{j=1}^{k} \frac{(Au_k, u_j)}{(u_j, u_j)} \cdot u_j \right\}, \ \mu_{k+1} \neq 0 \end{cases}$$

とする.

Lanczos
・ランチョス過程 : A がエルミートならば、アーノルディ過程が3項漸化式に「短縮」される.
それを示そう.

Au_ℓ は $u_1, \cdots, u_{\ell+1}$ に分解されるので、$(Au_\ell, u_m) = 0$ for $\forall m \geq \ell+2$.

で、$(Au_\ell, u_m) = (u_\ell, A^* u_m) = (u_\ell, Au_m)$ であるので、結局、A がエルミートなら、
(A がエルミート)
$(Au_\ell, u_m) = 0$ for $\forall |m-\ell| \geq 2$. となる. よって、アーノルディ過程は結局、

$K_s(A;u)$ に対し、A がエルミートなら

$$\begin{cases} u_1 := \mu_1 u, \ \mu_1 \neq 0 \\ \text{for } k=1 \text{ to } (s-1) \\ \quad u_{k+1} = \mu_{k+1}\left\{ Au_k - \frac{(Au_k, u_k)}{(u_k, u_k)} u_k - \frac{(Au_k, u_{k-1})}{(u_{k-1}, u_{k-1})} u_{k-1} \right\} \end{cases}$$

となる.

・なにか直交性をもたすのに過去の情報全てを必要としない、ということ.
便利そうだぞ?

⇩

CG法とは(この文脈で説明すると)

- クリロフ部分空間反復法である.
- 直交条件を採用 $\Leftrightarrow \{r_0, r_1, \cdots r_{s-1}\}$ が $K_s(A;r_0)$ の直交基底
  $\Leftarrow$ $r_k (k \geq 1)$ をランチョス過程で生成することとする
  $\Rightarrow$ #44 の条件2も満たす
  $\Rightarrow$ 「理想的な特徴」をもつ.

という方法といえる. (上の ~~ が Point)

Q. これが #59 の CG法と同じものになるのは本当?

A. 本当だ. まじめに計算してみると #59 と同じものになる. やってみるとよいだろう.

・ランチョス過程の自由度 μ_{k+1} がクリロフ部分空間反復法である為の条件を満たすように決まる.

注) 最適化問題の解法としてのCG法も存在する. 発想はCG法そのものだ…

Daisuke Furihata

(オマケ) 最適化問題の解法の1つの CG法.

(向) $f:\mathbb{R}^n \to \mathbb{R}$ を最小にする x^* を知りたい. に対し、反復法を考える.

$x^{(k)}$ に対し、$x^{(k+1)}$ ではうまく解に近づいたと期待すると $0 \simeq \nabla f(x^{(k+1)})$. よって、
$0 \simeq \nabla f(x^{(k+1)}) \simeq \nabla f(x^{(k)}) + \underbrace{H(x^{(k)})}_{\text{Hesse行列}}(x^{(k+1)} - x^{(k)})$ とTaylor展開して、

$$x^{(k+1)} \simeq x^{(k)} - H^{-1}\cdot\nabla f \quad \text{と考えられる.} \qquad (62.1)$$

・勾配 ∇f に対し、$H^{-1}\nabla f$ を共役勾配という.

しかし、$H^{-1}\nabla f$ の計算コストは高い. そこで、次のように考える.
(∇f を求めるだけでも大変なため)

$$\begin{cases} -H^{-1}\nabla f = w_k p^{(k)}, & (62.2) \\ p^{(k)} = -\nabla f(x^{(k)}) + \beta_{k-1} p^{(k-1)} & (62.3) \\ p^{(k)} \perp H(x^{(k-1)})\cdot p^{(k-1)} & (62.4) \end{cases}$$

(62.3) が成り立っている とする.

実はこれが CG法のミソなので.

どこかで見たような....
↓
CG法の枠組みを $H = A$ として採用していることに相当する.

すると、(62.3 & 4) より $\beta_{k-1} = \dfrac{(\nabla f(x^{(k)}), H(x^{(k-1)})p^{(k-1)})}{(p^{(k-1)}, H(x^{(k-1)})p^{(k-1)})}$, (62.5)

(62.2) より $w_k = -\dfrac{(p^{(k)}, \nabla f(x^{(k)}))}{(p^{(k)}, H(x^{(k)})p^{(k)})}$ (62.6) とまるまるで、結局、

$$\begin{cases} \beta_{k-1} = (62.5) \\ p^{(k)} = (62.3) \\ w_k = (62.6) \\ x^{(k+1)} = x^{(k)} + w_k p^{(k)} \end{cases}$$

として反復していけばよい ということになる.

注) もしも H が定数行列 かつ 正定値対称ならば、#57 の lemma より
$\{p^{(k)}\}_k$ は全て互いに一次独立となる. よって、この反復は n 回以内に必ず終了する.

Daisuke Furihata

(演習)

1. $A = \begin{pmatrix} 7 & 1 & 2 & 3 \\ 1 & 10 & 3 & 4 \\ 2 & 3 & 13 & 6 \\ 3 & 4 & 6 & 16 \end{pmatrix}$, $b = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$ に対し、

$Ax = b$ を #56 の最急降下法で解いてみよ.

2. 上の $Ax = b$ を、#59 の CG 法で解いてみよ.

```ruby
#
# 連立一次方程式に対し, 逐次最小化法のなかの最急降下法を試してみる
# a: 行列, b: ベクトル として,  ax = b を解いてみる.
#

include Math
require 'pp'

# 許せる誤差(ノルムで)
$eps = 1.0e-08

# 行列やベクトルのサイズ
$n = 4

# 行列
$a = [
  [7.0, 1.0, 2.0, 3.0],
  [1.0, 10.0, 3.0, 4.0],
  [2.0, 3.0, 13.0, 6.0],
  [3.0, 4.0, 6.0, 16.0 ]
]


# 右辺ベクトル
$b = [1.0, 2.0, 3.0, 4.0]

#

# ベクトルの内積を計算する
def inner_product(x,y)
  z = 0.0
  for i in (1..$n)
    z += x[i-1]*y[i-1]
  end
  return z
end

# ベクトルにスカラー倍する
def vc_coeff(c,b)
  z = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    z[i-1] = c * b[i-1]
  end
  return z
end

# ベクトルの和を計算する
def vc_sum(a,b)
  c = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    c[i-1] = a[i-1]+b[i-1]
  end
  return c
end

# 行列 かける ベクトル を計算する
def mtrx_vc_product(a,b)
  c = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    for j in (1..$n)
      c[i-1] += a[i-1][j-1]*b[j-1]
    end
  end

  return c
end

# 行列 A かける ベクトル を計算する
def a_vc(b)
  return mtrx_vc_product($a,b)
end

# 残差 b - Ax を計算する
def residual(x)
  return vc_sum($b, vc_coeff(-1.0, a_vc(x)))
end

# 最急降下法での一反復
def steepest_descent_iteration(x)
  r = residual(x)
  w = inner_product(r,r) / inner_product(r, a_vc(r))

  return vc_sum(x, vc_coeff(w,r))
end

# 1-ノルム
def norm_one(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1].abs
  end
  return v
end

# 2-ノルム
def norm_two(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1]**2
  end
  return sqrt(v)
end

# sup ノルム
def norm_sup(x)
  v = 0.0
  for i in (1..$n)
    if (x[i-1].abs > v) then
      v = x[i-1].abs
    end
  end
  return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
  return norm_two(x)
```

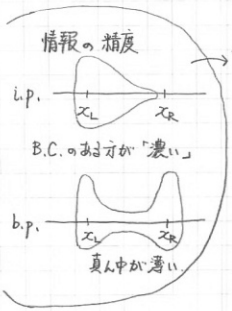
```
118 end
119
120 # 収束判定
121 def is_convergence(x)
122   if ( norm(residual(x)) < $eps) then
123       return true
124     else
125       return false
126   end
127 end
128
129 # 見易く出力
130 def print_result(x)
131   print "x = ["
132   for i in (1..$n-1)
133     printf("%.6f",x[i-1])
134     print ", "
135   end
136   printf("%.6f",x[$n-1])
137   print "], "
138   print "residual= ",norm(residual(x)),"¥n"
139 end
140
141
142 ########################
143 # 実際に計算!
144
145 # 近似解の初期値, ゼロはまずいので, 1としておく.
146 x = Array.new($n).collect{ 1.0 }
147 print_result(x)
148
149 # # 収束するまでループ
150 loop_times = 0
151 loop do
152   if is_convergence(x) then
153       break
154     else
155       x = steepest_descent_iteration(x)
156       print_result(x)
157       loop_times += 1
158     end
159 end
160
161 # # 結果出力
162 print "¥n¥nYou obtain a numerical solution after ",loop_ti
    mes," loops.¥n¥n"
163 print_result(x)
164 print "¥n"
165
166
```

```ruby
#
# 連立一次方程式に対し, 逐次最小化法のなかの CG法を試してみる
# a: 行列, b: ベクトル として,  ax = b を解いてみる.
#

include Math
require 'pp'

# 許せる誤差(ノルムで)
$eps = 1.0e-08

# 行列やベクトルのサイズ
$n = 4

# 行列
$a = [
  [7.0, 1.0, 2.0, 3.0],
  [1.0, 10.0, 3.0, 4.0],
  [2.0, 3.0, 13.0, 6.0],
  [3.0, 4.0, 6.0, 16.0 ]
]


# 右辺ベクトル
$b = [1.0, 2.0, 3.0, 4.0]

#

# ベクトルの内積を計算する
def inner_product(x,y)
  z = 0.0
  for i in (1..$n)
    z += x[i-1]*y[i-1]
  end
  return z
end

# ベクトルにスカラー倍する
def vc_coeff(c,b)
  z = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    z[i-1] = c * b[i-1]
  end
  return z
end

# ベクトルの和を計算する
def vc_sum(a,b)
  c = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    c[i-1] = a[i-1]+b[i-1]
  end
  return c
end

# 行列 かける ベクトル を計算する
def mtrx_vc_product(a,b)
  c = Array.new($n).collect{ 0.0 }
  for i in (1..$n)
    for j in (1..$n)
      c[i-1] += a[i-1][j-1]*b[j-1]
    end
  end

  return c
end

# 行列 A かける ベクトル を計算する
def a_vc(b)
  return mtrx_vc_product($a,b)
end

# 残差 b - Ax を計算する
def residual(x)
  return vc_sum($b, vc_coeff(-1.0, a_vc(x)))
end

#####
# CG 法での一反復(本当はこの中で収束判定をした方がよいのだが, まあいいか)
def cg_iteration(x,r,p,r_R,q)
  q_Q = inner_product(p,q)
  w = r_R/q_Q
  x_new = vc_sum(x, vc_coeff(w,p))
  r_new = vc_sum(r, vc_coeff((-1.0*w),q))

  r_R_new = inner_product(r_new, r_new)
  beta = r_R_new / r_R
  p_new = vc_sum(r_new, vc_coeff(beta,p))
  q_new = a_vc(p_new)

  return [x_new, r_new, p_new, r_R_new, q_new]
end
#####

# 1-ノルム
def norm_one(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1].abs
  end
  return v
end

# 2-ノルム
def norm_two(x)
  v = 0.0
  for i in (1..$n)
    v += x[i-1]**2
  end
  return sqrt(v)
end

# sup ノルム
def norm_sup(x)
  v = 0.0
  for i in (1..$n)
```

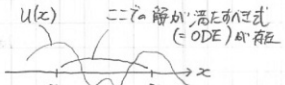
```
    if (x[i-1].abs > v) then
      v = x[i-1].abs
    end
  end
  return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
  return norm_two(x)
end

# 収束判定
def is_convergence(x)
  if ( norm(residual(x)) < $eps) then
      return true
    else
      return false
  end
end

# 見易く出力
def print_result(x)
  print "x = ["
  for i in (1..$n-1)
    printf("%.6f",x[i-1])
    print ", "
  end
  printf("%.6f",x[$n-1])
  print "], "
  print "residual= ",norm(residual(x)),"¥n"
end


########################
# 実際に計算!

# 初期値作成
x = Array.new($n).collect{ 1.0 }
r = residual(x)
p = vc_coeff(1.0, r)
r_R = inner_product(r,r)
q = a_vc(p)

print_result(x)


###
# 収束するまでループ
loop_times = 0
loop do
  if is_convergence(x) then
      break
    else
      x,r,p,r_R,q = cg_iteration(x,r,p,r_R,q)
      print_result(x)
      loop_times += 1
    end
end

# # 結果出力
print "¥n¥nYou obtain a numerical solution after ",loop_times," loops.¥n¥n"
print_result(x)
print "¥n"
```

§4. 常微分方程式の数値的求解

日本語ならば、三井先生、小藤先生が参考文献として鉄板、森先生の本(共立)も参考となる。



問題：



Ordinary Differential Equation 常微分方程式 が「解ける」には、適切な境界条件 Boundary Condition が必要で、通常は

$$\begin{cases} f(u, u_x, u_{xx}, \cdots) = 0 & (\text{ODE}) \\ u(x_L) = a, \\ \vdots \end{cases} \quad \left.\begin{matrix} \\ \\ \end{matrix}\right\} (\text{B.C.})$$

と記述される。

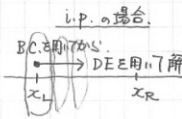
分類
- initial problem 初期値問題タイプ：x_L もしくは x_R のみで B.C. が与えられる
- boundary problem 境界値 〃：x_L および x_R での 〃

ex). i.p. の例、$\frac{du}{dx} = 3u$, $u(x_L) = 2$ (x_L の B.C. のみ)

b.p. の例、$\frac{d^2u}{dx^2} = 0$, $u(x_L) = 1$, $u(x_R) = 2$ (x_L, x_R の B.C.)

適した解法

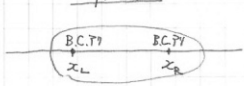
i.p. の場合：



近似解を新たに計算/追加する手法がいくつもある。

例) Euler法,
Runge-Kutta法,
複形多段階法,
⋮

b.p. の場合：



全体をまとめて1つの系としてとらえて、つじつまのあう解を一気に求める。

問題全体をそもそもどう近似的にとらえるか、が大きなPoint。

例) 差分近似,
スペクトル法,
⋮

- Newton力学は常微分方程式で記述される代表的な例。
- B.P. に対して I.P. の計算手法を用いる Shooting Method というものもある。これは、B.C. のうちいくつかを「達成目標」とし、代わりに適当な B.C. を「調整パラメータ」として I.P. を解き、目標が達成されるまでパラメータを調整するものである。

§4.1. 境界値問題.

まず、代表的な「差分法」を見てみよう.
これは、微分を差分で近似する方法である.

差分近似の例

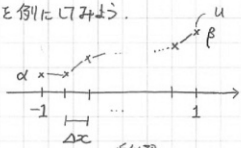
微分	差分近似
$\frac{du}{dx}$	$\frac{u(x+h)-u(x)}{h}$
	$\frac{u(x+h)-u(x-h)}{2h}$
$\frac{d^2u}{dx^2}$	$\frac{u(x+h)-2u(x)+u(x-h)}{h^2}$

Strum-Liouville 型問題

$$\begin{cases} \frac{d}{dx}\left(a(x)\frac{d}{dx}u(x)\right) + b(x)u(x) = 0 \quad (-1<x<1) & \sim(1) \\ u(-1)=\alpha, \\ u(1)=\beta \end{cases} \quad (a(x), b(x) \text{は既知}) \quad \sim(2)$$

・よくあるパターンでは、
$a(x) \le 0 \le b(x)$
である.

を例にしてみよう.



x と x を分割し、$x_k := -1 + k\Delta x \quad (0 \le k \le N)$
where $N\Delta x = 2$ としておいて、

近似解 $u_k \simeq u(x_k)$ とする.

また、$a_k := a(x_k)$, $b_k = b(x_k)$ としておく.
すると、(1) & (2) は差分近似によって例えば

$$\begin{cases} \frac{a_{k+1/2}\left(\frac{u_{k+1}-u_k}{\Delta x}\right) - a_{k-1/2}\left(\frac{u_k-u_{k-1}}{\Delta x}\right)}{\Delta x} + b_k u_k = 0, \quad (1 \le k \le N-1) & \sim(3) \\ u_0 = \alpha, \\ u_N = \beta \end{cases}$$

(1)相当 → 上段
(2)相当 → 下段
と近似される. そしてこれは、

$$\Leftrightarrow \begin{cases} \left(\frac{a_{k+1/2}}{\Delta x^2}\right)u_{k+1} + \left(\frac{-a_{k+1/2}-a_{k-1/2}}{\Delta x^2} + b_k\right)u_k + \left(\frac{a_{k-1/2}}{\Delta x^2}\right)u_{k-1} = 0 \quad (1 \le k \le N-1) \\ u_0 = \alpha, \quad u_N = \beta \end{cases}$$

$$\Leftrightarrow \begin{pmatrix} 1 & 0 & & & 0 \\ \frac{a_{1/2}}{\Delta x^2} & -\frac{a_{1/2}-a_{3/2}}{\Delta x^2}+b_1 & \frac{a_{3/2}}{\Delta x^2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{a_{N-3/2}}{\Delta x^2} & -\frac{a_{N-3/2}-a_{N-1/2}}{\Delta x^2}+b_{N-1} & \frac{a_{N-1/2}}{\Delta x^2} \\ 0 & & & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \\ \vdots \\ 0 \\ \beta \end{pmatrix} \quad \sim(4)$$

・連立一次式(4)
計算量は、係数行列が
疎(ゼロでない N 個程度しか
非ゼロ成分がない)であるので、
N^3 程度で済む.

となるので、連立一次方程式(4)を解けば、近似解 u_k $(k=0 \sim N)$ が求まる.

注). (4) の係数行列は、$a(x) \le 0 \le b(x)$ の時、優対角であることに注意せよ.
(よくあるパターン.)

注). (4) による全体の打切 err $\propto \Delta x^2$ であるが、同時に丸め err. $\propto 1/\Delta x^2$ となる (森.共立出版. p258~259 参照のこと)
結局、err. $\propto C_1\Delta x^2 + C_2/\Delta x^2$ の形になる為、Δx をただ小さくしても誤差が小さくなるわけでない.

$\{\phi_n\}_n$ が正規直交基底
$\Leftrightarrow$
- 内積 $(\ ,\ )$ に対し、
- $\|\phi_n\| = \sqrt{(\phi_n, \phi_n)} = 1$
- $(\phi_n, \phi_m) = 0$ for $n \neq m$
- (函数空間を $\{\phi_n\}_n$ で完備)

スペクトル法 も見ておこう．これは、与えられた領域上に適当な(正規直交)基底 $\{\phi_n(x)\}_n$ を用意しておいて、

$u(x) \simeq \sum_n c_n \phi_n(x)$ と近似して c_n を求める という手法である．

$(\Leftrightarrow c \cdot g$, where $(g)_n = \phi_n)$

Point. $\phi_n(x)$ を知っているので、$\frac{d}{dx}\phi_n(x) = \sum_m \lambda_{nm}\phi_m(x)$ とできる．

$(\Leftrightarrow \frac{d}{dx} g = D g$, $(D)_{nm} = \lambda_{nm})$

前ページの例に対し、

前ページの
$$(1) \simeq \frac{d}{dx}\left\{a(x)\frac{d}{dx}(C\cdot g)\right\} + b(x) C\cdot g$$
$$= \frac{d}{dx}\left(a(x)\cdot C^T D g\right) + b(x) C^T g$$
$$= a(x) C^T D^2 g + a'(x) C^T D g + b(x) C^T g$$
$$= C^T\left(a(x)D^2 + a'(x)D + b(x)\right) g \quad \text{for } -1 < x < 1 \qquad \sim(1)$$

前ページの
$$(2) \quad \sum_n c_n \phi_n(-1) = \alpha, \quad \sum_n c_n \phi_n(1) = \beta. \qquad \sim(2)$$

↓

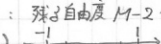
But (1) はまだ計算に使えない．(えが自由すぎる)．

↓

いろいろな方法で、(nの個数-2) まで自由度を減らす．例えば、基底を $\{\phi_n\}_{n=1}^M$ としておいて、

2つは既にB.C.として(2)の形で式になっている．

選点法 (Collocation)：残る自由度 $M-2$ 個の式をつくる為に、立向 $[-1,1]$ を $(M-1)$ 分割して



$\Delta x := \frac{2}{M-1}$, $x_k := -1 + k\Delta x$ $(0 \le k \le M-1)$

- ϕ の性質を生かしきれないのが弱点．

に対し、$\{x_k\}_{k=1}^{M-2}$ (これが選点) 上で (1) が厳密に 0 となることを要求する．

(つまり、 $\rightarrow M-2$ 本の式ができる)

- 均等に選点をとるよいとはいけないわけではない．これはあくまで一つの例．

$$\left\{ C^T\left(a(x_k)D^2 + a'(x_k)D + b(x_k)\right) g \Big|_{x_k} = 0 \quad , \ k = 1 \sim M-2, \right. \qquad \sim(3)$$
$$\left. \& \ (2) \right.$$

- $\{\phi_n\}_n$ と内積の相性の良し悪しに左右される．

Galerkin ガレルキン法：ϕ_n のうち、重要な方から $M-2$ 本を選んで $\{\hat{\phi}_\ell\}_{\ell=1}^{M-2}$ として、

$$\int_{-1}^{1} (1) \times \hat{\phi}_\ell(x)\, dx = 0 \quad , \ \ell = 1 \sim M-2 \qquad \sim(4)$$

を要求する方法．

注) 多項式近似 $u(x) \simeq c_0 + c_1 x + c_2 x^2 + \cdots + c_\ell x^\ell$ もスペクトル法の一種である．

この時、$\phi_n(x) = x^n$ とおいても悪くはないが、直交多項式 (Legendre 多項式 など) を ϕ_n にしておくと便利である．

（演習）

境界値問題

$$\begin{cases} u'' - u' - 2u + 2 = 0 & (0 < x < 1) \\ u(0) = u(1) = 0 \end{cases}$$

（高橋，岩波，p11）

の数値解を求めてみよう．

1)．#65 のように、差分法で解いてみよう．

2) #66 のスペクトル法（選点 / Galerkin のいずれでもよい）で解いてみよう．

ただし、Δx は $1/50$ 程度にしておいて良い（細かい方が近似度は高いが、計算に時間がかかる）．

（解答例）

1) #65 同様に計算してみると（面倒な分母をはらって），$N\Delta x = 1$ として

$$\begin{cases} (2-\Delta x)u_{k+1} + (-4-4\Delta x^2)u_k + (2+\Delta x)u_{k-1} = -4\Delta x^2 & (1 \le k \le N-1) \\ u_0 = 0 \\ u_N = 0 \end{cases}$$

となる．

#65 の(4)に相当する式を書くと，$\gamma_+ := 2-\Delta x$, $\gamma := -4(1+\Delta x^2)$, $\gamma_- := 2+\Delta x$ として

$$\begin{pmatrix} 1 & 0 & & & & \\ \gamma_- & \gamma & \gamma_+ & & 0 & \\ & \gamma_- & \gamma & \gamma_+ & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \gamma_- & \gamma & \gamma_+ \\ & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ -4\Delta x^2 \\ -4\Delta x^2 \\ \vdots \\ -4\Delta x^2 \\ 0 \end{pmatrix}$$

となる．

$u_0 = 0$, $u_N = 0$ を利用してもっとすっきりさせると，

$$\begin{pmatrix} \gamma & \gamma_+ & & 0 \\ \gamma_- & \ddots & \ddots & \\ & \ddots & \ddots & \gamma_+ \\ 0 & & \gamma_- & \gamma \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \\ u_{N-1} \end{pmatrix} = -4\Delta x^2 \begin{pmatrix} 1 \\ \vdots \\ \\ 1 \end{pmatrix}$$

となる．

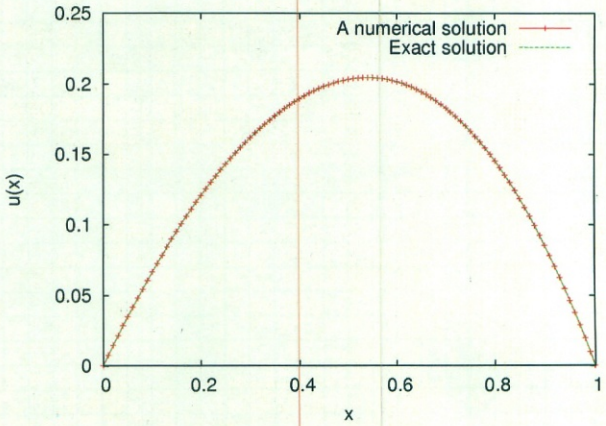
あとは これを 何らかの方法で解けば良い．

注) 上の問題は線形微分方程式なので、手で解けて

$$u(x) = -\frac{1}{e^2+e+1} e^{2x} - \frac{e+1}{e^2+e+1} e^{1-x} + 1$$

となる．自分でも計算できるか、やってみよ．

ちなみに、$N = 100 \left(\Leftrightarrow \Delta x = 1/100\right)$ として LU 分解で
前ページの連立一次eq. を解くと、下のような結果が得られる。



Daisuke Furihata

2) #66 にあわせて、$(\mathcal{G})_n = \phi_n(x)$, $\frac{d}{dx}\mathcal{G} \overset{\text{行列}}{\cong} D\mathcal{G}$ と近似するとして

$u(x) := \sum_n C_n \phi_n(x) = C^T \cdot \mathcal{G}$ と書くと、

$u'(x) \cong C^T D\mathcal{G}$, $u''(x) \cong C^T D^2 \mathcal{G}$ であるので

問題の方程式 $\Leftrightarrow$ $C^T(D^2 - D - 2I)\mathcal{G} = -2$. と書ける.

次にこの C を求める方法として、今回は 選点法 を用いてみよう. すると、

$$\begin{cases} C^T(D^2-D-2I)\mathcal{G}\big|_{x=k\Delta x} = -2, & 1 \le k \le M-2 \\ C^T\mathcal{G}\big|_{x=0} = 0 \\ C^T\mathcal{G}\big|_{x=1} = 0 \end{cases}$$

とおくので、

↘ ただし、$(M-1)\Delta x = 1$. となる. (今回は)

これを「C を求める式」に書き直すと、

$$\begin{pmatrix} (\mathcal{G}|_{x=0})^T \\ ((D^2-2D-2I)\mathcal{G}|_{x=\Delta x})^T \\ \vdots \\ ((D^2-2D-2I)\mathcal{G}|_{x=(M-2)\Delta x})^T \\ (\mathcal{G}|_{x=1})^T \end{pmatrix} \cdot C = \begin{pmatrix} 0 \\ -2 \\ \vdots \\ -2 \\ 0 \end{pmatrix} \quad \sim (*)$$

を解くということになる.

で、あとは具体的に ϕ_n を選んで、上に代入して解けば良い.

今回は 境界条件 と a matching もさえて、

$$\phi_n(x) = \begin{cases} \cos(n-1)\pi x & : n \text{ が奇} \\ \sin n\pi x & : n \text{ が偶} \end{cases}, \quad 1 \le n \le M \quad \text{としてみる.}$$

(楽をする為に、M を奇数としておく). すると、

$$\mathcal{G} = \begin{pmatrix} 1 \\ \sin 2\pi x \\ \cos 2\pi x \\ \sin 4\pi x \\ \cos 4\pi x \\ \vdots \\ \cos(M-1)\pi x \end{pmatrix} \text{より、} \quad \frac{d}{dx}\mathcal{G} = \begin{pmatrix} 0 & & & & \\ & \begin{matrix} 0 & 1 \\ -1 & 0 \end{matrix} & & & \\ & & \begin{matrix} 0 & 2 \\ -2 & 0 \end{matrix} & & \\ & & & \ddots & \\ & & & & \begin{matrix} 0 & \frac{M-1}{2} \\ -\frac{M-1}{2} & 0 \end{matrix} \end{pmatrix} \cdot 2\pi \cdot \mathcal{G}$$

D.

とおり、

$D^2 = -4k^2 \begin{pmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}$ はすぐ分かるので、

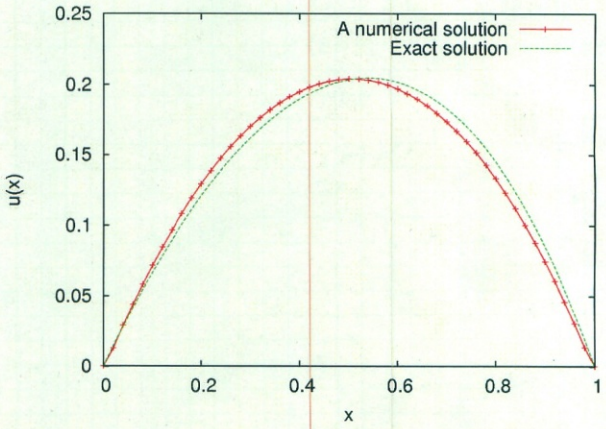
$$D^2 - D - 2I = \begin{pmatrix} -2 & & & \\ & -2-4k^2-1 & & \\ & & -2-4k^2-2 & \\ & & & \ddots \\ & & & -2-4k^2\left(\frac{M-1}{2}\right)^2 - \frac{M-1}{2} \\ & & & \frac{M-1}{2} \quad -2-4k^2\left(\frac{M-1}{2}\right)^2 \end{pmatrix}$$

となる。

よって あとは 前ページの ㊋ に代入して、㊌ を解くだけである。

Point：上を見ると、この形では ϕ_n の n の大きさの影響が不公平に強い。
その為、$n \to$ 大 でおかしくなることが想定される。
（n が小さい ϕ_n に大きな誤差が入ってしまう）

どうしたら良いか、考えてみよう。 $M=51 \left(\Leftrightarrow \Delta x = \frac{1}{50}\right)$ として計算してみたもの。
ずいぶん誤差がある……



```ruby
# -*- coding: euc-jp -*-
#
# 境界値問題を差分法で解いてみる．
# 問題が線形な場合は，単なる「連立一次方程式を解く問題」に帰着する．
# よって，あとは連立一次方程式が解ければ OK.
# $a: 行列，$b: ベクトル として， $ax = $b を解く．
# とりあえずLU分解で解く．

include Math
require 'pp'

# 領域を何分割するか
N = 100

# DeltaX
$dx = 1.0/N

# 行列やベクトルのサイズ
# 両端の値は与えられているので今回は不要としている．
$n = (N + 1) - 2

##
# 結果を出力するルーチン
def print_result(x)
 print "# x, u(x) \n"

 printf("%.6f  %.6f\n", 0.0, 0.0)
 for i in (1..$n)
   printf("%.6f  %.6f\n", i*$dx, x[i-1])
 end
 printf("%.6f  %.6f\n", 1.0, 0.0)

end

#####
# 係数行列作成
# 一旦 0 が並んでいるものを作ってから，必要なところを書き換える．
$a = Array.new($n).collect{ Array.new($n).collect{0.0} }

# 右上副対角成分
x = 2.0-$dx
for i in (1..$n-1)
 $a[i-1][i] = x
end

# 対角成分
x = -4.0*(1.0+$dx**2)
for i in (1..$n)
 $a[i-1][i-1] = x
end

# 左下副対角成分
x = 2.0+$dx
for i in (1..$n-1)
 $a[i][i-1] = x
end


#####
# 右辺ベクトル
x = -4.0*($dx**2)
$b = Array.new($n).collect{ x }

########################
# 実際に計算!

#
# LU 分解(ピボット変換してない) ... b に関係なく計算可能.
# ループが三重なので，計算量が n^3 に比例することがすぐわかる．
#
for k in (1..($n-1))
 for i in (k+1..$n)
   $a[i-1][k-1] /= $a[k-1][k-1]
   for j in (k+1..$n)
     $a[i-1][j-1] -= $a[i-1][k-1]*$a[k-1][j-1]
   end
 end
end

#
# 前進代入で y = L^(-1)b を求める．
# ループが二重で済み，計算量がおおよそ n^2/2 で済むことに注意せよ．
#
y = Array.new($n).collect{ 0.0 }
for i in (1..$n)
 y[i-1] = $b[i-1]
 if i < $n
   for j in ((i+1)..$n)
     $b[j-1] -= $a[j-1][i-1]*$b[i-1]
   end
 end
end

#
# 後退代入で x = U^(-1)y を求める．減っていく数字で for 文を回すのは厄介なので，downto を使っている．
# ループが二重で済み，計算量がおおよそ n^2/2 で済むことに注意せよ．
#
x = Array.new($n).collect{ 0.0 }
$n.downto(1) { |i|
 x[i-1] = y[i-1]/$a[i-1][i-1]
 if i > 1
   (i-1).downto(1){ |j|
   y[j-1] -= ($a[j-1][i-1]/$a[i-1][i-1])*y[i-1]
   }
 end
}


# # 結果出力
print "\n\n# You obtain a numerical solution.\n\n"
print_result(x)
print "\n"
```

```ruby
# -*- coding: euc-jp -*-
#
# 境界値問題をスペクトル法(選点)で解いてみる.
# 問題が線形な場合は，単なる「連立一次方程式を解く問題」に帰着する.
# よって，あとは連立一次方程式が解ければ OK.
# $a: 行列，$b: ベクトル として， $ax = $b を解く.
#

include Math
require 'pp'

# 基底関数をいくつとるか. この安直なやり方だとあまり大きく取れない.
M = 51

# DeltaX
$dx = 1.0/(M-1)

# 行列やベクトルのサイズ
$n = M

# 偶奇判定(古い ruby だと偶奇判定機能がないので念の為に作っておく)
def is_even(n)
 return (n%2 == 0)
end

# 基底関数
def phi(n,x)
 if is_even(n)
  return sin(n*PI*x)
 else
  return cos((n-1)*PI*x)
 end
end

# 基底関数を並べたベクトル q
def vec_q(x)
 v = Array.new(M).collect{0.0}
 for i in (1..M)
  v[i-1] = phi(i,x)
 end
 return v
end

#
# ベクトルの内積を計算する
def inner_product(x,y)
 z = 0.0
 for i in (1..$n)
  z += x[i-1]*y[i-1]
 end
 return z
end

# ベクトルにスカラー倍する
def vc_coeff(c,b)
 z = Array.new($n).collect{ 0.0 }
 for i in (1..$n)
  z[i-1] = c * b[i-1]
 end
 return z
end

# ベクトルの和を計算する
def vc_sum(a,b)
 c = Array.new($n).collect{ 0.0 }
 for i in (1..$n)
  c[i-1] = a[i-1]+b[i-1]
 end
 return c
end

# 行列 かける ベクトル を計算する
def mtrx_vc_product(a,b)
 c = Array.new($n).collect{ 0.0 }
 for i in (1..$n)
  for j in (1..$n)
   c[i-1] += a[i-1][j-1]*b[j-1]
  end
 end

 return c
end

# 行列 A かける ベクトル を計算する
def a_vc(b)
 return mtrx_vc_product($a,b)
end

# 残差 b - Ax を計算する
def residual(a,b,x)
 return vc_sum(b, vc_coeff(-1.0, mtrx_vc_product(a,x)))
end

# 1-ノルム
def norm_one(x)
 v = 0.0
 for i in (1..$n)
  v += x[i-1].abs
 end
 return v
end

# 2-ノルム
def norm_two(x)
 v = 0.0
 for i in (1..$n)
  v += x[i-1]**2
 end
 return sqrt(v)
end

# sup ノルム
def norm_sup(x)
 v = 0.0
 for i in (1..$n)
  if (x[i-1].abs > v) then
   v = x[i-1].abs
```

```
  end
 end
 return v
end

# 実際に使うノルム. 好きなものに取り替えよう.
def norm(x)
 return norm_sup(x)
end

### 出力ルーチン
#
# 結果を出力
def print_result(x)
 print "# x, u(x) \n"
 for i in (1..M)
  printf("%.6f  %.6f\n", (i-1)*$dx, x[i-1])
 end

end

# 残差を出力
def print_residual(a,b,x)
 print "# residual= ",norm(residual(a,b,x)),"\n"
end

##############################
# 行列 D^2 - D - 2I
# 一旦 0 が並んでいるものを作ってから，必要なところを書き換える.
$d = Array.new($n).collect{ Array.new($n).collect{0.0} }

# 右上副対角成分
for i in (1..(M-1)/2)
 $d[2*i-1][2*i] = (- i).to_f
end

# 左下副対角成分
for i in (1..(M-1)/2)
 $d[2*i][2*i-1] = i.to_f
end

# 対角成分
w = 4.0*(PI**2)
for i in (1..M)
 $d[i-1][i-1] = -2.0 - w * (((i/2).floor)**2)
end

##############################
# 問題全体の係数行列
# 一旦 0 が並んでいるものを作ってから，必要なところを書き換える.
$a = Array.new($n).collect{ Array.new($n).collect{0.0} }

# 1行目
x = 0.0
v = vec_q(x)
for j in (1..M)
 $a[0][j-1] = v[j-1]
end

# 2〜(M-1)行目
for i in (2..M-1)
 x = (i-1)*$dx
 v = vec_q(x)
 w = mtrx_vc_product($d, v)
 for j in (1..M)
  $a[i-1][j-1] = w[j-1]
 end
end

# M行目
x = 1.0
v = vec_q(x)
for j in (1..M)
 $a[M-1][j-1] = v[j-1]
end

# 後で残差を計算できるよう，行列のオリジナルをとっておく.
$a_org = Array.new($n).collect{ Array.new($n).collect{0.0} }
for i in (1..M)
 for j in (1..M)
  $a_org[i-1][j-1] = $a[i-1][j-1]
 end
end


###
# 右辺ベクトル
x = -2.0
$b = Array.new($n).collect{ x }
$b[0] = 0.0
$b[M-1] = 0.0

# 後で残差を計算できるよう，右辺ベクトルのオリジナルをとっておく.
$b_org = Array.new($n).collect{ 0.0 }
for i in (1..M)
 $b_org[i-1] = $b[i-1]
end


##########################
# 実際に計算! LU 分解を用いる.

#
# LU 分解(ピボット変換してない) ... b に関係なく計算可能.
# ループが三重なので，計算量が n^3 に比例することがすぐわかる.

for k in (1..($n-1))
 for i in (k+1..$n)
  $a[i-1][k-1] /= $a[k-1][k-1]
  for j in (k+1..$n)
   $a[i-1][j-1] -= $a[i-1][k-1]*$a[k-1][j-1]
  end
 end
end

```

```
#
# 前進代入で y = L^(-1)b を求める.
# ループが二重で済み，計算量がおおよそ n^2/2 で済むことに注意せよ.
#
y = Array.new($n).collect{ 0.0 }
for i in (1..$n)
 y[i-1] = $b[i-1]
 if i < $n
  for j in ((i+1)..$n)
   $b[j-1] -= $a[j-1][i-1]*$b[i-1]
  end
 end
end

#
# 後退代入で x = U^(-1)y を求める. 減っていく数字で for 文を回すのは厄介なので，downto を使っている.
# ループが二重で済み，計算量がおおよそ n^2/2 で済むことに注意せよ.
#
c = Array.new($n).collect{ 0.0 }
$n.downto(1) { |i|
 c[i-1] = y[i-1]/$a[i-1][i-1]
 if i > 1
  (i-1).downto(1){ |j|
  y[j-1] -= ($a[j-1][i-1]/$a[i-1][i-1])*y[i-1]
  }
 end
}

# 得られた結果は係数なので，関数値に直す.
$u = Array.new($n).collect{ 0.0 }
for i in (1..M)
 x = (i-1)*$dx
 v = vec_q(x)
 $u[i-1] = inner_product(v,c)
end

# # 結果出力
print "\n\n# You obtain a numerical solution.\n\n"
print_result($u)
print "\n"
print_residual($a_org, $b_org, $u)

```

§4.2 初期値問題

初期値問題は、B.C. のある点を start 地点として、DE を離散化した式に従って近似解を次々に構成していくことで解くのが一般的である。以下、それを見ていこう。

典型的問題

$$\begin{cases} \dfrac{du}{dt} = f(u,t) \quad , \quad u = u(t) : \mathbb{R} \to \mathbb{R}^m \quad , \quad t > 0 \qquad \sim (1) \\ \text{B.C.} \quad u(0) = u_{ini} \end{cases}$$

を考える（高階の ODE も、例えば $u_1 := u$, $u_2 := \frac{du}{dt}$, $u_3 := \frac{du_2}{dt}$, … とすれば、新しい変数を導入 一階の ODE に簡単に変形できる）。

↓ どうやって N.A. で解く？

大きく分けて以下の2通り。

① 一段法

$u_k \overset{\text{近似}}{\simeq} u(k\Delta t)$. $k = 0, 1, 2, \cdots$ に対し、

$u_{k+1} = \overset{\text{解法}}{F}(u_k, \Delta t)$ として $u_k \to u_{k+1}$ と "一段で" 近似解を構成していく解法という。

例) Euler 法, Taylor 法, 差分法, Runge-Kutta 法

② 多段法

$u_{k+1} = F(u_k, u_{k-1}, \cdots, u_{k-\ell}, \Delta t)$ として、

$(u_{k-\ell}, \cdots, u_k) \to u_{k+1}$ と "多段で" 近似解を構成していく解法。

例) 線形多段階法

所与の
・ODE に解が存在するか？
は別の問題だ。
例えば (1) はリプシッツ条件

$\|f(u_1,t) - f(u_2,t)\| \le L \|u_1 - u_2\|$

$(L < \infty)$

を満たすも、u の範囲で一意に解をもつ。

Q. 結局どれがいいの？

A. ・簡単な問題 & 粗い解でよい
→ Euler 法
・速度も精度も欲しい
→ 線形多段階法
・そこそこの精度 & 安定性
→ Runge-Kutta 法
・迷ったとりあえず？
→ Runge-Kutta 法

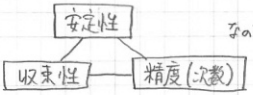
(準備)・**局所誤差** : $\overset{\text{近似}}{u_0} = \overset{\text{真}}{u}(t_0)$, $u_1 = u(t_1)$, …, $u_k = u(t_k)$ $(t_j = j\Delta t)$ が成立している時に、u_{k+1} がどれだけ err. をもつか。
この err. を局所 err. という。

・**大域的誤差** : 純粋に $\|u_k - u(t_k)\|$ のこと。

・**次数** : $\left(\text{局所誤差}/\Delta t\right)$ の Δt のオーダー。大域 err. のオーダーと一致することが多い。

・**収束性** : $\Delta t \to 0$ の時 大域 err. $\to 0$ となる時、その解法は収束性をもつと言う。(or 近似解は収束するとも言う)

実は N.A. の3つの柱は



なのだ。安定性については #54 で。

§4.2.1. Euler法.

もっとも Simple & 高速な解法. 精度その他の性質は期待してはいけない.

前ページの問題に対し、右辺を「前進差分近似」して、

$$\begin{cases} \dfrac{u_{k+1}-u_k}{\Delta t} = f(u_k, t_k) & (k=0,1,2,\cdots) \\ u_0 = u_{\text{ini}} \end{cases} \quad \left(\Leftrightarrow u_{k+1} = u_k + \Delta t f(u_k,t_k)\right) \quad \sim(1)$$

とする方法. $u_0 \to u_1 \to u_2 \to \cdots$ と、(1)に代入していくだけで次々求まる.

(以下、面倒なので1次元 $u(t)$ で話を進める.)

局所誤差: 近似 u_k = 真 $u(t_k)$ が成り立っていると仮定して、(あと、$u(t) \in C^2$ くらいとして)

$$\begin{aligned} u_{k+1} - u(t_{k+1}) &\overset{(1)}{=} \left(u_k + \Delta t f(u_k,t_k)\right) - u(t_k+\Delta t) \quad \text{Taylor展開} \\ &= \left(\quad 〃 \quad\right) - \left(u(t_k) + \frac{du}{dt}\Big|_{t_k}\Delta t + \frac{d^2u}{dt^2}\Big|_{t_k+\theta\Delta t}\cdot\frac{\Delta t^2}{2}\right) \quad (0\le\theta\le 1) \\ &= \left(\quad 〃 \quad\right) - \left(u_k + f(u_k,t_k)\Delta t + \frac{d^2u}{dt^2}\Big|_{t_k+\theta\Delta t}\frac{\Delta t^2}{2}\right) \\ &= -\frac{1}{2}\frac{d^2u}{dt^2}\Big|_{t_k+\theta\Delta t}\cdot\Delta t^2 \quad \text{が局所 err.} \end{aligned}$$

次数: $\left(\text{局所 err}/\Delta t\right)$ の Δt のオーダー $= 1$ より、1次.

大域誤差: まあ、次の仮定をしておく. ① f はリプシッツ連続. $|f(u,t)-f(v,t)| \le L|u-v|$. ② u は C^2 級. ($\Rightarrow |u_{tt}| < {}^\exists M$).

この時、

$e_k := u_k - u(t_k)$ に対し、

$$\begin{aligned} |e_{k+1}| &= |u_{k+1} - u(t_{k+1})| = \left|\left(u_k + f(u_k,t_k)\Delta t\right) - \left(u(t_k) + f(u(t_k),t_k)\Delta t + \frac{1}{2}u_{tt}\Big|_{t_k+\theta\Delta t}\Delta t^2\right)\right| \\ &\le |e_k| + |f(u_k,t_k) - f(u(t_k),t_k)|\Delta t + \frac{1}{2}M\Delta t^2 \\ &\le |e_k| + L|e_k|\Delta t + \frac{1}{2}M\Delta t^2 = (1+L\Delta t)|e_k| + \frac{1}{2}M\Delta t^2 \end{aligned}$$

$$\Leftrightarrow \left(|e_{k+1}| + \zeta\right) \le \left(1+L\Delta t\right)\left(|e_k| + \zeta\right), \quad \zeta := \frac{M\Delta t}{2L}.$$

よって、$|e_k| + \zeta \le (1+L\Delta t)^k(|e_0| + \zeta) = (1+L\Delta t)^k\zeta$ であるので、

$$|e_k| \le \frac{M\Delta t}{2L}\left\{(1+L\Delta t)^k - 1\right\}$$

で、$(1+L\Delta t) \le e^{L\Delta t}$ より $(1+L\Delta t)^k \le e^{L\Delta t\cdot k} = e^{Lt_k}$ の為、

$$|e_k| \le \frac{M}{2L}\left(e^{Lt_k} - 1\right)\cdot\Delta t \quad \text{と評価できる.} \quad \sim(2)$$

収束性: (2)は、$t = t_k$ を保ったまま $\Delta t \to 0$ とすると $e(t) \to 0$ となるので収束を意味する.

$e_0 = 0$ は前提としても良いよね.

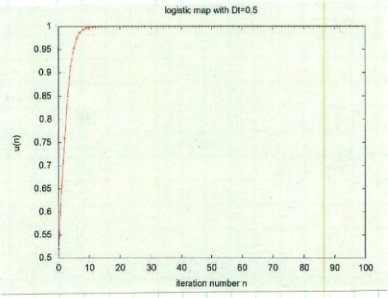
$t = t_k$ を保つというのは、$\Delta t \to 0$ の時 $k = t/\Delta t$ としておくという意味だね

§4.2.2. 安定性について.

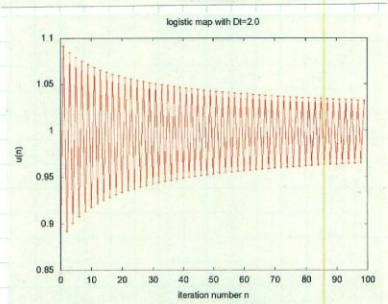
実は、ODEの解法で最も重要なfactorは「安定性」である. これは何か？というと、(大ざっぱにいって)「数値解が暴走しないこと」である.
↑いろんな種類の暴走がある.

Q. じゃあ厳密に言うと？
A.「○○安定性」という概念がいくつかあって、各々 def. が異なる.

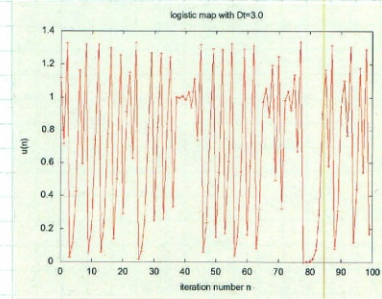
例えば、$\frac{du}{dt} = u(1-u)$ をEuler法で解いてみると…… $u_0 = 0.4$ として、u_1 以降は、



← $\Delta t = 0.5$ の場合.
妥当そうな近似解が得られている.



← $\Delta t = 2.0$ の場合.
振動する近似解.
とても信用できそうにない.



← $\Delta t = 3.0$ の場合.
カオスな近似解.
もう何がなんだか

などとあり、$\Delta t > 2.0$ ではまっとうな近似解を得られていない.
(他にも、「発散」なども暴走の一種)
↓
というか、発散こそ代表例なのだが.

・というか、このEuler法による写像(Logistic map)はカオスの有名な例なのだ.
(が発生する)

注). $u_t = u(1-u)$ は厳密解が容易に求まる. $1 = \frac{u_t}{u(1-u)} = \frac{u_t}{u} + \frac{u_t}{1-u} \Rightarrow \frac{du}{u} + \frac{du}{1-u} = dt$ として、
後は分かるね？

§4.2.3. Runge-Kutta 法.

一段法の中で、最も良く知られている方法、
まず一般形を書く.

s段 Runge-Kutta

ODE: $\frac{du}{dt} = f(u,t)$ に対し、$u_n \simeq u(t_n)$, $t_n = t_0 + n\Delta t$, として

$$\begin{cases} u_{n+1} = u_n + \Delta t \sum_{i=1}^{s} b_i f_i \\ f_i := f\left(u_n + \Delta t \sum_{j=1}^{s} a_{ij} f_j ,\ t_n + c_i \Delta t \right) \quad (i = 1 \sim s) \end{cases} \tag{1}$$

とするものを言う. (a_{ij}, b_i, c_i は公式を定めるパラメータ)

注). 通常は、$\sum_{j=1}^{s} a_{ij} = c_i$ が成立する.

(1)は使いやすい？ …… ケースbyケース.

(パターン1) $j \geq i$ の時 $a_{ij} = 0$ の場合.
よくみると、$f_1, f_2, f_3, \cdots, f_s$ を順次計算できる. (陽的 explicit という)
扱いやすい.

(パターン2) $j > i$ の時 $a_{ij} = 0$ の場合.
$f_i = f(u_n + \Delta t(a_{i1}f_1 + a_{i2}f_2 + \cdots + a_{ii}f_i),\ t_n + c_i\Delta t)$
を解いて f_i を求めねばならない (この段階の前に $f_1 \sim f_{i-1}$ は求まっている). 半陰的 semi-implicit ともいう、やや大変.

(パターン3) 上のいずれでもない場合.
f_i を求める為に、$f_1 \sim f_s$ とからんだ連立非線形方程式を解かねばならない、陰的 implicit という、大変.

利点.

- 一段法なのでシンプル (特に陽的な場合は).
- わかりやすい.
- Δt を自由に変えられる.

欠点

- 計算量が多め.
- 精度を上げるのがすし大変 (そういうパラメータをどうやって見つけるか)

Daisuke Furihata

有名な Runge-Kutta 公式

陽的なもののみ

いくつか有名なものを挙げておこう。

(Euler 法の Runge-Kutta 公式の1つである。$s=1$ の陽的なものがそうである)

2段：修正 Euler 法 (2次)

$$\begin{cases} u_{n+1} = u_n + \Delta t \cdot f_2, \\ f_1 := f(u_n, t_n), \quad f_2 := f(u_n + \frac{1}{2}\Delta t f_1, t_n + \frac{1}{2}\Delta t) \end{cases}$$

2段：改良 Euler 法 (2次)

$$\begin{cases} u_{n+1} = u_n + \frac{1}{2}\Delta t (f_1 + f_2), \\ f_1 := f(u_n, t_n), \quad f_2 := f(u_n + \Delta t f_1, t_n + \Delta t) \end{cases}$$

3段：Heun 法 (3次)

$$\begin{cases} u_{n+1} = u_n + \Delta t (\frac{1}{4} f_1 + \frac{3}{4} f_3), \\ f_1 := f(u_n, t_n), \ f_2 := f(u_n + \frac{1}{3}\Delta t f_1, t_n + \frac{1}{3}\Delta t), \\ f_3 := f(u_n + \frac{2}{3}\Delta t f_2, t_n + \frac{2}{3}\Delta t) \end{cases}$$

3段：Kutta 法 (3次)

$$\begin{cases} u_{n+1} = u_n + \Delta t (\frac{1}{6} f_1 + \frac{2}{3} f_2 + \frac{1}{6} f_3) \\ f_1 := f(u_n, t_n), \ f_2 := f(u_n + \frac{1}{2}\Delta t f_1, t_n + \frac{1}{2}\Delta t), \\ f_3 := f(u_n + \Delta t(-f_1 + 2f_2), t_n + \Delta t) \end{cases}$$

4段：古典 Runge-Kutta 法 (4次) ← とてもよく使われる。

$$\begin{cases} u_{n+1} = u_n + \Delta t (\frac{1}{6} f_1 + \frac{1}{3} f_2 + \frac{1}{3} f_3 + \frac{1}{6} f_4), \\ f_1 := f(u_n, t_n), \ f_2 := f(u_n + \frac{1}{2}\Delta t f_1, t_n + \frac{1}{2}\Delta t), \\ f_3 := f(u_n + \frac{1}{2}\Delta t f_2, t_n + \frac{1}{2}\Delta t), \\ f_4 := f(u_n + \Delta t f_3, t_n + \Delta t) \end{cases}$$

⋮

Runge-Kutta の次数

実は、段数と(到達可能な)次数は、今のところ次のように分かっている。

段数 s	1	2	3	4	5	6	7	8	9	10	10以上
到達可能な次数 $p(s)$	1	2	3	4	4	5	6	6	7	7	($p(s) \leq s-2$ ぐらいかも…)

- 古典 Runge-Kutta が「おいしい」ことが分かる。

(演習)

IVP
初期値問題を解いてみよう.

IVP 1) $\frac{du}{dt} = u(1-u)$, $u(t=0) = 0.4$

IVP 2) $\begin{cases} \frac{du_1}{dt} = (2-u_2)u_1 \\ \frac{du_2}{dt} = (2u_1-3)u_2 \end{cases}$ (高橋, 岩波書店, p.83)

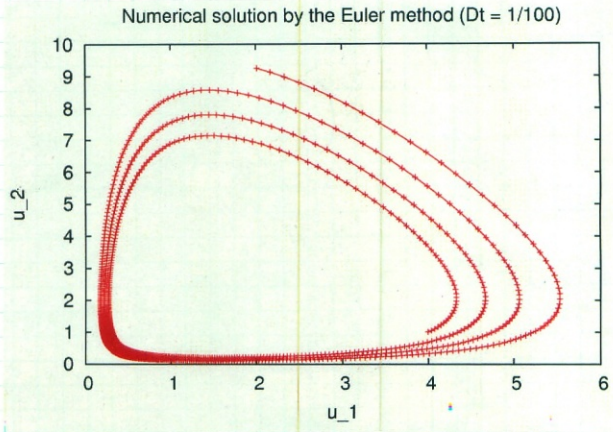
$u_1(0) = 4$, $u_2(0) = 1$.

に対し,

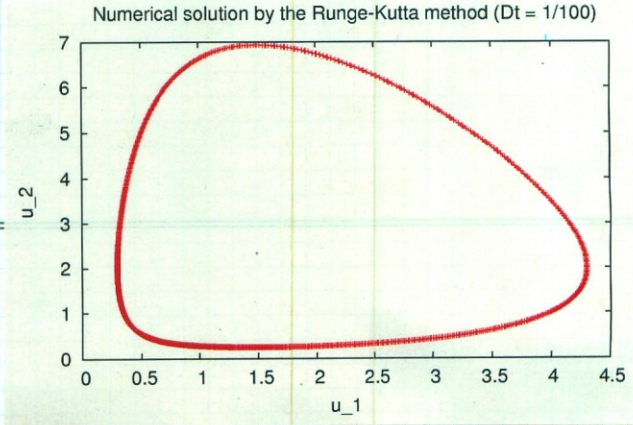
a) Euler法で
b) 古典的 Runge-Kutta法で

} 各々数値解を求めよ.
ただし, Δt をいろいろ変えてみよ.

IVP 2) に対し
Eulerによる結果 →



Runge-Kuttaによる結果 →



```
# -*- coding: euc-jp -*-
#
# 初期値問題 1) を Euler法で解いてみる.
#

include Math
require 'pp'

# DeltaX
$dt = 2.0

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = 0.4

# 微分方程式の右辺
def f(u)
 return u*(1.0-u)
end

# Euler法による関数更新
def new_by_euler(u)
 return u + $dt * f(u)
end

##
# 結果を出力するルーチン
def print_result(u,t)
 printf("%.6f  %.6f\n", t, u)
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Euler scheme.\n
"
print "# Delta t = ", $dt,", u(0) = ",$u0,"\n"
print "# t, u(t) \n\n"

# 初期値設定
u = $u0
t = 0.0
print_result(u,t)

N = ($t_max/$dt).ceil
for i in (1..N)
 u = new_by_euler(u)
 t = i*$dt
 print_result(u,t)
end


```

```ruby
# -*- coding: euc-jp -*-
#
# 初期値問題 2) を Euler法で解いてみる.
#
# このプログラムが出力した結果をファイルに記録して(例えば ddd.dat として),
# gnuplot で
#   plot "ddd.dat" using 2:3
# とすると，u1 vs u2 のグラフが描ける.
#
include Math
require 'pp'

# DeltaX
$dt = 1.0/100

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = [4.0, 1.0]

# 微分方程式の右辺
def f(u)
 a = u[0]
 b = u[1]
 return [ (2.0-b)*a, (2.0*a-3.0)*b ]
end

# Euler法による関数更新
def new_by_euler(u)
 v = f(u)
 a = v[0]
 b = v[1]
 return [ u[0] + $dt*a, u[1] + $dt*b ]
end

##
# 結果を出力するルーチン
def print_result(u,t)
 printf("%.4f  %.4f  %.4f\n", t, u[0], u[1])
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Euler scheme.\n"
print "# Delta t = ", $dt,", u(0) = (",$u0[0],", ",$u0[1],")\n"
print "# t, u1(t), u2(t) \n\n"

# 初期値設定
u = $u0
t = 0.0
print_result(u,t)

N = ($t_max/$dt).ceil
for i in (1..N)
 u = new_by_euler(u)
 t = i*$dt
 print_result(u,t)
end
```

```
# -*- coding: euc-jp -*-
#
# 初期値問題 1) を古典 Runge-Kutta 法で解いてみる.
#

include Math
require 'pp'

# DeltaX
$dt = 1.0/10

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = 0.4

# 微分方程式の右辺
def f(u)
 return u*(1.0-u)
end

# Runge-Kutta法による関数更新
def new_by_rk(u)
 f1 = f(u)
 f2 = f(u + 0.5*$dt*f1)
 f3 = f(u + 0.5*$dt*f2)
 f4 = f(u + $dt*f3)
 return u + $dt*(f1/6.0 + f2/3.0 + f3/3.0 + f4/6.0)
end

##
# 結果を出力するルーチン
def print_result(u,t)
 printf("%.6f  %.6f\n", t, u)
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Runge-Kutta sc
heme.\n"
print "# Delta t = ", $dt,", u(0) = ",$u0,"\n"
print "# t, u(t) \n\n"

# 初期値設定
u = $u0
t = 0.0
print_result(u,t)

N = ($t_max/$dt).ceil
for i in (1..N)
 u = new_by_rk(u)
 t = i*$dt
 print_result(u,t)
end
```

```ruby
# -*- coding: euc-jp -*-
#
# 初期値問題 2) を古典 Runge-Kutta 法で解いてみる.
#
# このプログラムが出力した結果をファイルに記録して(例えば ddd.dat として),
# gnuplot で
#   plot "ddd.dat" using 2:3
# とすると，u1 vs u2 のグラフが描ける.
#
include Math
require 'pp'

# DeltaX
$dt = 1.0/100

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = [4.0, 1.0]

# 微分方程式の右辺
def f(u)
 a = u[0]
 b = u[1]
 return [ (2.0-b)*a, (2.0*a-3.0)*b ]
end

# Runge-Kutta 法による関数更新
def new_by_rk(u)
 f1 = f(u)
 f2 = f([ u[0]+0.5*$dt*f1[0], u[1]+0.5*$dt*f1[1] ])
 f3 = f([ u[0]+0.5*$dt*f2[0], u[1]+0.5*$dt*f2[1] ])
 f4 = f([ u[0]+$dt*f3[0], u[1]+$dt*f3[1] ])

 a = u[0] + $dt * (f1[0]/6.0 + f2[0]/3.0 + f3[0]/3.0 + f4[0]/6.0)

 b = u[1] + $dt * (f1[1]/6.0 + f2[1]/3.0 + f3[1]/3.0 + f4[1]/6.0)

 return [ a,b ]
end

##
# 結果を出力するルーチン
def print_result(u,t)
 printf("%.4f  %.4f  %.4f\n", t, u[0], u[1])
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Runge-Kutta scheme.\n"
print "# Delta t = ", $dt,", u(0) = (",$u0[0],", ",$u0[1],")\n"
print "# t, u1(t), u2(t) \n\n"

# 初期値設定
u = $u0
t = 0.0
print_result(u,t)

N = ($t_max/$dt).ceil
for i in (1..N)
 u = new_by_rk(u)
 t = i*$dt
 print_result(u,t)
end
```

§4.2.4 線形多段階法 Linear Multistep method

u, f がベクトルでも大丈夫！

過去のデータを「線形に」組み合わせて新しい点での近似値を得る方法。

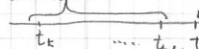
問題の ODE : $\frac{du}{dt} = f(u,t)$ に対し、 $u_k \overset{\text{近似}}{\simeq} u(t_k)$, $f_k := f(u_k, t_k)$ として

$$\frac{\alpha_0 u_k + \alpha_1 u_{k+1} + \cdots + \alpha_\ell u_{k+\ell}}{\Delta t} = \beta_0 f_k + \beta_1 f_{k+1} + \cdots + \beta_\ell f_{k+\ell}$$

$$\Leftrightarrow \sum_{j=0}^{\ell} \alpha_j u_{k+j} = \Delta t \cdot \sum_{j=0}^{\ell} \beta_j f_{k+j}, \quad \alpha_0 \neq 0, \alpha_\ell \neq 0 \quad \sim \star$$

- $\beta_\ell = 0$ の時は $u_{k+\ell}$ が陽に求まるので「陽的な」解法であると言う。
  逆に $\beta_\ell \neq 0$ の時は「陰的な」解法と言う。

ℓ 段 LM 法という（過去のデータを ℓ 個使う）　新しく求めるデータ。



利点
- 計算量が小さい。過去のデータは線形和として利用するだけなので計算量にはほとんど影響しない → 段数 ℓ が増えても計算量はほとんど変わらない。
- 計算がシンプル。特に陽的な時は。　つまり？「速い！」
- （後述するが）一般に $\ell \to$ 大で精度 → 向上なので、精度を上げても計算量がほとんど増えずに済む

欠点
- 「初期データ」$u_0, u_1, \cdots, u_{k+\ell-2}$ (ℓ-1個) は何らかの別の手法で得る必要がある
- 独立変数 t の刻み幅 Δt を一定にしないといけない。
- 日本語の資料が少ない（^^

・そのせいか、NA の専門家以外では、LM は Runge-Kutta ほど知られていないね。

注）☆は非常に汎用性の高い書き方なので、様々なものを含む。
例えば（ある程度知られている）アダムス { バッシュフォース / ムルトン } 公式や、Euler 解法、BDF（後退微分公式）などが含まれる。

Point.

陽的
大雑把には、ℓ 段 LM は ℓ 次にできる（陰的なら $\ell+1$ 次も可能）。にもかかわらず、f の新たな計算はわずか1回で済む！

m段 Runge-Kutta 法は u の更新のために f を m個計算しないといけないことを思い出そう。
つまり、単純比較で、ℓ 次のスキームとしては、LM 法は RK 法より ℓ 倍以上速い！

§4.2.4.1 LMの例.

アダムス型公式
→ 予測子・修正子法

アダムス型公式　$\frac{du}{dt} = f(u,t)$ に対し、右辺を多段にするタイプ.

(上のODE) $\Leftrightarrow$ $u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u,t)\,dt$ に対し、

(上式) 右辺中の f を 過去の t_ℓ 上での値を用いた補間多項式で近似してしまう.

おまけ　補間多項式.

$t = t_n, t_{n+1}, \cdots, t_{n+\ell}$ での f の値 $f_n, f_{n+1}, \cdots, f_{n+\ell}$ を用いた補間多項式とは

ℓ次
$P_\ell(t) = f_n \cdot Q_n(t) + f_{n+1} \cdot Q_{n+1}(t) + \cdots + f_{n+\ell} \cdot Q_{n+\ell}(t)$ のコト.

ただし
$Q_{n+k}(t) := \dfrac{\prod_{j\neq k} (t - t_{n+j})}{\prod_{j\neq k} (t_{n+k} - t_{n+j})}$ とする.

注: $Q_{n+k}(t_{n+j}) = \begin{cases} 1 & : j = k \\ 0 & : j \neq k \end{cases}$ であることに気づけ! (せめて確かめてみよう)

これを正しくは
ラグランジュ補間公式
という.

より細かくは,

- $t_{n-p+1}, t_{n-p+2}, \cdots, t_n$ 上での f を用いる → (P次の) アダムス-バッシュフォース公式 (AB公式) (陽的になる)
- $t_{n-p+2}, t_{n-p+3}, \cdots, t_{n+1}$ 〃 → (P次の) アダムス-ムルトン公式 (AM公式) (陰的になる)

Q. $p=3$ として、AB公式とAM公式の係数 β を各々計算してみよ.

注) AB公式では
$\beta_0 = \frac{5}{12}\Delta t$, $\beta_1 = -\frac{4}{3}\Delta t$,
$\beta_2 = \frac{23}{12}\Delta t$, $\beta_3 = 0$
AM公式では
$\beta_0 = -\frac{1}{12}\Delta t$, $\beta_1 = \frac{2}{3}\Delta t$,
$\beta_2 = \frac{5}{12}\Delta t$
となるはず.

$\Leftrightarrow$

AB公式: $P(t) := \sum_{j=0}^{p-1} f_{n-j} Q_{n-j}(t)$ として
(p段, $\ell = p-1$ ということ)
$u_{n+1} - u_n = \int_{t_n}^{t_{n+1}} P(t)\,dt = \sum_{j=0}^{p-1} f_{n-j} \underbrace{\left(\int_{t_n}^{t_{n+1}} Q_{n-j}(t)\,dt \right)}_{\beta_{p-1-j} \text{ となる.}}$ とするもの.
($\beta_3 = \beta_2 = 0$)

AM公式: $P(t) = \sum_{j=-1}^{p-2} f_{n-j} Q_{n-j}(t)$ として
($p-1$段, $\ell = p-1$ ということ)
$u_{n+1} - u_n = \sum_{j=-1}^{p-2} f_{n-j} \underbrace{\left(\int_{t_n}^{t_{n+1}} Q_{n-j}(t)\,dt \right)}_{\beta_{p-2-j} \text{ となる.}}$ とするもの.

Daisuke Furihata

アダムス型公式のパラメータを簡単にすると…

後退(差分)作用素 ∇_- ($\nabla_- f_n := f_n - f_{n-1}$, $\nabla_-^2 f_n = \nabla_-(f_n - f_{n-1})$, …) と、

数列 $\begin{cases} \gamma_0^* = 1 \\ \gamma_\ell^* + \frac{1}{2}\gamma_{\ell-1}^* + \cdots + \frac{1}{\ell+1}\gamma_0^* = 1 \end{cases}$,

数列 $\begin{cases} \gamma_1 = 1. \\ \gamma_\ell + \frac{1}{2}\gamma_{\ell-1} + \cdots + \frac{1}{\ell+1}\gamma_0 = 0 \end{cases}$ を用いて、アダムス型公式は次のようにまとめられる。

p段(p次) AB公式 $u_{n+1} - u_n = \Delta t \cdot \sum_{j=0}^{p-1} \gamma_j^* \nabla_-^j f_n$

p段(p+1次) AM公式 $u_{n+1} - u_n = \Delta t \cdot \sum_{j=0}^{p} \gamma_j \nabla_-^j f_{n+1}$

⇓ 次数をそろえて並べてみると…

1次公式. AB: $u_{n+1} - u_n = \Delta t \cdot f_n$ AM: $u_{n+1} - u_n = \Delta t \cdot f_{n+1}$

2次公式 AB: $u_{n+1} - u_n = \Delta t \left(\frac{3}{2} f_n - \frac{1}{2} f_{n-1} \right)$

AM: $u_{n+1} - u_n = \Delta t \left(\frac{1}{2} f_{n+1} + \frac{1}{2} f_n \right)$

3次公式 AB: $u_{n+1} - u_n = \Delta t \left(\frac{23}{12} f_n - \frac{16}{12} f_{n-1} + \frac{5}{12} f_{n-2} \right)$

AM: $u_{n+1} - u_n = \Delta t \left(\frac{5}{12} f_{n+1} + \frac{2}{3} f_n - \frac{1}{12} f_{n-1} \right)$

4次公式 AB: $u_{n+1} - u_n = \Delta t \left(\frac{55}{24} f_n - \frac{59}{24} f_{n-1} + \frac{37}{24} f_{n-2} - \frac{9}{24} f_{n-3} \right)$

AM: $u_{n+1} - u_n = \Delta t \left(\frac{9}{24} f_{n+1} + \frac{19}{24} f_n - \frac{5}{24} f_{n-1} + \frac{1}{24} f_{n-2} \right)$

⋮ などとなる.

Daisuke Furihata

Backward Differential Formula 後退微分公式

$\frac{du}{dt} = f(u,t)$ に対し、左辺を多段にするタイプ。

具体的には、$u(t)$ を $t_{n-p+1}, t_{n-p+2}, \cdots, t_n, t_{n+1}$ での u の近似値 $u_{n-p+1}, u_{n-p+2}, \cdots, u_n, u_{n+1}$ を用いた補間多項式 $U(t)$ (p次) で近似し、

$\frac{du}{dt} \simeq \frac{dU(t)}{dt}$ として計算式を作るものである。

(この時、$t = t_{n+1}$ とする)

(要するに?) $U(t) := \sum_{j=-1}^{p-1} u_{n-j} Q_{n-j}(t)$ として、

$$\left.\frac{dU}{dt}\right|_{t=t_{n+1}} = f(u_{n+1}, t_{n+1}) \quad \sim \bigstar$$

とするものである。

実際に計算してみよう。

$p=1$ の場合: $U(t) = \sum_{j=-1}^{0} u_{n-j} Q_{n-j}(t) = u_{n+1} Q_{n+1}(t) + u_n Q_n(t)$

で、$\begin{cases} Q_n(t) = \frac{t-t_{n+1}}{t_n - t_{n+1}} = r \\ Q_{n+1}(t) = \frac{t-t_n}{t_{n+1}-t_n} = -r+1 \end{cases}$ ただし $\begin{cases} \Delta t = t_{n+1} - t_n, \\ t = -r\Delta t + t_{n+1} \end{cases}$ となるので。

$(\Rightarrow \frac{d}{dt} = -\frac{1}{\Delta t}\frac{d}{dr})$

$$\frac{dU}{dt} = -\frac{1}{\Delta t}\frac{d}{dr}\left\{u_{n+1}(1-r) + u_n \cdot r\right\} = \frac{u_{n+1}-u_n}{\Delta t}$$

となるので、

$\bigstar$ 相当は $\frac{u_{n+1}-u_n}{\Delta t} = f(u_{n+1}, t_{n+1})$ となる。

$p=2$ の場合: $U(t) = \sum_{j=-1}^{1} u_{n-j} Q_{n-j}(t)$ に対し、

$$\begin{cases} Q_{n-1}(t) = \frac{(t-t_n)(t-t_{n+1})}{(t_{n-1}-t_n)(t_{n-1}-t_{n+1})} = \frac{(1-r)\Delta t \cdot (-r\Delta t)}{(-\Delta t)(-2\Delta t)} = \frac{r(r-1)}{2} \\ Q_n(t) = \frac{(t-t_{n-1})(t-t_{n+1})}{(t_n-t_{n-1})(t_n-t_{n+1})} = \frac{(2-r)\Delta t \cdot (-r\Delta t)}{\Delta t \cdot (-\Delta t)} = r(2-r) \\ Q_{n+1}(t) = \frac{(t-t_{n-1})(t-t_n)}{(t_{n+1}-t_{n-1})(t_{n+1}-t_n)} = \frac{(2-r)\Delta t \cdot (1-r)\Delta t}{2\Delta t \cdot \Delta t} = \frac{(r-2)(r-1)}{2} \end{cases}$$ より、

$$\left.\frac{dU}{dt}\right|_{t_{n+1}} = -\frac{1}{\Delta t}\frac{d}{dr}\left\{u_{n-1}\frac{r(r-1)}{2} + u_n r(2-r) + u_{n+1}\frac{(r-2)(r-1)}{2}\right\}\Big|_{r=0}$$

$$= \frac{\frac{3}{2}u_{n+1} - 2u_n + \frac{1}{2}u_{n-1}}{\Delta t}$$

となるので

$\bigstar$ 相当は $\frac{3u_{n+1} - 4u_n + u_{n-1}}{2\Delta t} = f(u_{n+1}, t_{n+1})$ とする。

$\frac{1}{\Delta t}\left(\tilde{\alpha}_0 u_{n+1} + \tilde{\alpha}_1 u_n + \tilde{\alpha}_2 u_{n-1} + \cdots + \tilde{\alpha}_\ell u_{n-\ell+1}\right) = f(u_{n+1}, t_{n+1})$ と表すと、

ℓ	1	2	3	4	5	6
$\tilde{\alpha}_0$	1	3/2	11/6	25/12	137/60	49/20
$\tilde{\alpha}_1$	−1	−2	−3	−4	−5	−6
$\tilde{\alpha}_2$		1/2	3/2	3	5	15/2
$\tilde{\alpha}_3$			−1/3	−4/3	−10/3	−5/3
$\tilde{\alpha}_4$				1/4	5/4	15/4
$\tilde{\alpha}_5$		0			−1/5	−5/5
$\tilde{\alpha}_6$						1/6

という式になるハズ。

予測子・修正子法

LM法の解法は
- 陽的 … 速いが解の精度が…
- 陰的 … 精度は良いが求解に困る

のいずれかであるが、この2つを組みあわせて良いとこどりをしようとする方法。

Predictor
予測子：陽的解法を用いて、(粗い)近似値を作る手順。

Corrector
修正子：得られている(粗い)近似値を用いて $f(u_{n+1}, t_{n+1})$ の近似値を求め、陰的解法の右辺に代入して u_{n+1} を新しく得て改善する手順。

↳ の2つを組みあわせるもの。

例). P：3段3次 AB公式 $u_{n+1}-u_n = \Delta t\left(\frac{23}{12}f_n - \frac{4}{3}f_{n-1} + \frac{5}{12}f_{n-2}\right)$

C：2段3次 AM公式 $u_{n+1}-u_n = \Delta t\left(\frac{5}{12}f_{n+1} + \frac{2}{3}f_n - \frac{1}{12}f_{n-1}\right)$ ～☆

として、PとCを用いて u_{n+1}(と f_{n+1})の近似を改善していく。

$u_n, f_{n-2}, f_{n-1}, f_n$ 既知

↓ Pを使う

$u_{n+1}^{(0)}$ … 粗い

↓

$f_{n+1}^{(0)} := f(u_{n+1}^{(0)}, t_{n+1})$ ← f_{n+1} の改善を"E"とよぶ。

↓ Cを使って

$u_{n+1}^{(1)} = u_n + \Delta t\left(\frac{5}{12}f_{n+1}^{(0)} + \frac{2}{3}f_n - \frac{1}{12}f_{n-1}\right)$ … $u_{n+1}^{(0)}$ を改善した！

↓

$f_{n+1}^{(1)} := f(u_{n+1}^{(1)}, t_{n+1})$

↓ Cを再び使って

$u_{n+1}^{(2)} = \cdots$ ← $u_{n+1}^{(1)}$ を改善した！

↓

$f_{n+1}^{(2)} := f(u_{n+1}^{(2)}, t_{n+1})$

↓ Cを再び使って(以下同様)

と、$u_{n+1}^{(m)}$ という近似値を得る方法である。

- これは、☆をヨコともに解くのと違って、計算量は非常に小さい。

注) 通常は、PECE もしくは PECECE だけで(改善を)おえてしまうことが多い。

fの計算はEのみで発生するので、この場合、fはせいぜい1回か2回しか計算しないで済むことになる。

Daisuke Furihata

（演習）

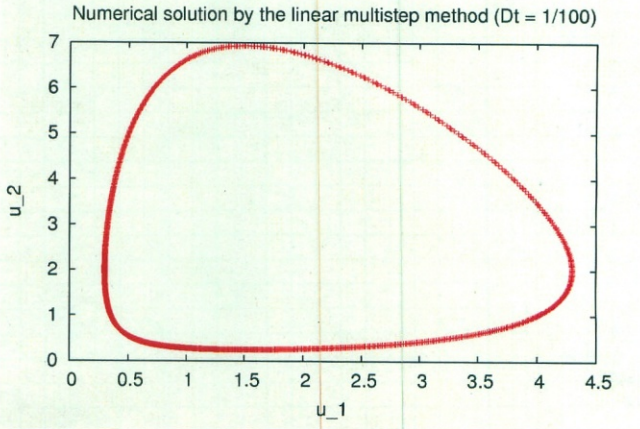
初期値問題を，予測子・修正子法で解いてみよう．
ただし，予測子を4次のAB公式，修正子を4次のAM公式とし，
初期データ u_1, u_2, u_3 については Runge-Kutta法で求めて用いよ．
（不足する つまり $u_4, u_5, u_6, u_7, \cdots$ を計算で求めよ、ということ）

1) $\dfrac{du}{dt} = u(1-u)$ ， $u(t=0) = 0.4$

2) $\begin{cases} \dfrac{du_x}{dt} = (2-u_y)u_x , \\ \dfrac{du_y}{dt} = (2u_x - 3)u_y , \end{cases}$

$u_x(0) = 4$， $u_y(0) = 1$． （ $u = \begin{pmatrix} u_x \\ u_y \end{pmatrix}$ についての問題 ）．

LM法による結果．（サンプルプログラムは，Runge-Kutta法の2倍速い！）



```ruby
# -*- coding: euc-jp -*-
#
# 初期値問題 1) を4次の線形多段階法で解いてみる.
#

include Math
require 'pp'

# DeltaT
$dt = 1.0/10

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = 0.4

# 微分方程式の右辺
def f(u)
 return u*(1.0-u)
end

# Runge-Kutta法による関数更新
def new_by_rk(u)
 f1 = f(u)
 f2 = f(u + 0.5*$dt*f1)
 f3 = f(u + 0.5*$dt*f2)
 f4 = f(u + $dt*f3)
 return u + $dt*(f1/6.0 + f2/3.0 + f3/3.0 + f4/6.0)
end

# 予測子
def predictor(old_u,old_f)
 new_u = old_u + $dt * ((55.0/24.0)*old_f[0] - (59.0/24.0)*old_f[1] + (37.0/24.0)*old_f[2] - (3.0/8.0)*old_f[3])
 return new_u
end

# 修正子
def corrector(old_u,old_f,new_f)
 new_u = old_u + $dt * ((3.0/8.0)*new_f + (19.0/24.0)*old_f[0] - (5.0/24.0)*old_f[1] + (1.0/24.0)*old_f[2])
 return new_u
end

# 実は厳密解が分かっている.
def exact_solution(t)
 c = $u0/(1.0-$u0)
 return c * exp(t) / (1.0 + c * exp(t))
end

##
# 結果を出力するルーチン. 厳密解と並べてみる.
def print_result(u,t)
 printf("%.6f %.6f %.6f\n", t, u, exact_solution(t))
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Runge-Kutta scheme.\n"
print "# Delta t = ", $dt,", u(0) = ",$u0,"\n"
print "# t, u(t) \n\n"

# 初期値設定
u = $u0
old_f = [f(u)]
t = 0.0
print_result(u,t)

# もう3つだけ，Runge-Kutta 法で作る.
for i in (1..3)
 u = new_by_rk(u)
 old_f.unshift(f(u))
 t = i*$dt
 print_result(u,t)
end

# あとは線形多段階法で計算する．繰り返し部分を PECE に決めうちしてみる.
N = ($t_max/$dt).ceil
for i in (4..N)

 # P: 予測子
 new_u = predictor(u,old_f)

 # E: 評価
 new_f = f(new_u)

 # C: 修正子
 new_u = corrector(u,old_f,new_f)

 # E: 評価
 new_f = f(new_u)

 # 値の更新
 u = new_u
 old_f.unshift(new_f)
 old_f.delete_at(4)

 # 表示
 t = i*$dt
 print_result(u,t)
end
```

```ruby
# -*- coding: euc-jp -*-
#
# 初期値問題 2) を4次の線形多段階法で解いてみる.
#
# このプログラムが出力した結果をファイルに記録して(例えば ddd.dat として),
# gnuplot で
#   plot "ddd.dat" using 2:3
# とすると，u1 vs u2 のグラフが描ける.
#
include Math
require 'pp'

# DeltaT
$dt = 1.0/100

# 計算を独立変数 t のどこまで行うか.
$t_max = 10.0

# 初期値
$u0 = [4.0, 1.0]

# 微分方程式の右辺
def f(u)
 a = u[0]
 b = u[1]
 return [ (2.0-b)*a, (2.0*a-3.0)*b ]
end

# Runge-Kutta 法による関数更新
def new_by_rk(u)
 f1 = f(u)
 f2 = f([ u[0]+0.5*$dt*f1[0], u[1]+0.5*$dt*f1[1] ])
 f3 = f([ u[0]+0.5*$dt*f2[0], u[1]+0.5*$dt*f2[1] ])
 f4 = f([ u[0]+$dt*f3[0], u[1]+$dt*f3[1] ])

 a = u[0] + $dt * (f1[0]/6.0 + f2[0]/3.0 + f3[0]/3.0 + f4[0]/6.0)

 b = u[1] + $dt * (f1[1]/6.0 + f2[1]/3.0 + f3[1]/3.0 + f4[1]/6.0)

 return [ a,b ]
end

# 予測子
def predictor(old_u,old_f)
 new_u = [0.0, 0.0]
 new_u[0] = old_u[0] + $dt * ((55.0/24.0)*old_f[0][0] - (59.0/24.0)*old_f[1][0] + (37.0/24.0)*old_f[2][0] - (3.0/8.0)*old_f[3][0])
 new_u[1] = old_u[1] + $dt * ((55.0/24.0)*old_f[0][1] - (59.0/24.0)*old_f[1][1] + (37.0/24.0)*old_f[2][1] - (3.0/8.0)*old_f[3][1])
 return new_u
end

# 修正子
def corrector(old_u,old_f,new_f)
 new_u = [0.0, 0.0]
 new_u[0] = old_u[0] + $dt * ((3.0/8.0)*new_f[0] + (19.0/24.0)*old_f[0][0] - (5.0/24.0)*old_f[1][0] + (1.0/24.0)*old_f[2][0])
 new_u[1] = old_u[1] + $dt * ((3.0/8.0)*new_f[1] + (19.0/24.0)*old_f[0][1] - (5.0/24.0)*old_f[1][1] + (1.0/24.0)*old_f[2][1])
 return new_u
end

##
# 結果を出力するルーチン
def print_result(u,t)
 printf("%.4f  %.4f  %.4f\n", t, u[0], u[1])
end

#########################
# 実際に計算!

print "\n"
print "# Numerical solution obtained by the Runge-Kutta scheme.\n"
print "# Delta t = ", $dt,", u(0) = (",$u0[0],", ",$u0[1],")\n"
print "# t, u1(t), u2(t) \n\n"

# 初期値設定
u = $u0
old_f = [f(u)]
t = 0.0
print_result(u,t)

# もう3つだけ，Runge-Kutta 法で作る.
for i in (1..3)
 u = new_by_rk(u)
 old_f.unshift(f(u))
 t = i*$dt
 print_result(u,t)
end

# あとは線形多段階法で計算する．繰り返し部分を PECE に決めうちしてみる.
N = ($t_max/$dt).ceil
for i in (4..N)

 # P: 予測子
 new_u = predictor(u,old_f)

 # E: 評価
 new_f = f(new_u)

 # C: 修正子
 new_u = corrector(u,old_f,new_f)

 # E: 評価
 new_f = f(new_u)

 # 値の更新
 u = new_u
 old_f.unshift(new_f)
 old_f.delete_at(4)

 # 表示
 t = i*$dt
```

```
108   print_result(u,t)
109 end
110
111
112
```

Daisuke Furihata

Report 1. A. $f(x) = e^{-x} - x$ に対し、$f(x) = 0$ を満たす x を

2通りの方法で求めよ.

Computer言語は問わないが、プログラミングは自分で行え.

そして、

① プログラムのソース、及びその解説

② 計算結果, 途中のデータをグラフにしたもの.
(のエラー)

をレポートにして提出せよ.

B. $P_{15}(x) := x^{15} - \frac{5}{2}x^{13} + 2.375x^{11} - 1.08631x^{9} + 0.249628x^{7}$

$- 0.0275467x^{5} + 0.00113058x^{3} - 0.0001025063x$

として、$P_{15}(x) = 0$ の根 15ヶ 全てを求めよ.

そして、A同様にレポートにして提出せよ.

Report 2.

A. Hilbert 行列 H_n に対し、($n=5$ として) その条件数 $K_\alpha(H_5)$ を
$\alpha = 1, 2, \infty$ として 各々 求めよ.
これは手計算でも良い.

B. $A = \begin{pmatrix} 1 & 1 & 0 & 1 & -1 \\ 0 & 1 & -1 & 3 & 0 \\ -2 & -1 & 1 & 2 & 3 \\ 1 & 0 & 2 & -1 & -1 \\ -1 & -2 & -3 & -7 & 1 \end{pmatrix}$, $b = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$ に対し、

$Ax = b$ を LU 分解で解け.

レポートの作成に関しては Report 1 の注意と同じとせよ (*)

C. $A = \begin{pmatrix} 11 & 1 & 2 & 3 & 4 \\ 1 & 15 & 3 & 4 & 5 \\ 2 & 3 & 19 & 5 & 6 \\ 3 & 4 & 5 & 23 & 7 \\ 4 & 5 & 6 & 7 & 27 \end{pmatrix}$, $b = \begin{pmatrix} 5 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix}$ に対し、

$Ax = b$ を Jacobi 法, Gauss-Seidel 法, SOR 法で解け.
(*) は上と同じ.

D. C と同じ A, b に対し、$Ax = b$ を CG 法で解け.
(*) は上と同じ.

Daisuke Furihata

Report 3.

常微分方程式の数値解を求めてみよう.

IVP: $\frac{d^2u}{dt^2} = -u$,

$u(0) = 10$, $\frac{du(0)}{dt} = -5$ に対し、数値解をもとめ10程度まで2通りの方法で求めよ.

A. (解く前の準備)

上のIVPは2階常微分方程式の形をしている. これに対し、

$v := \frac{du}{dt}$

という新しい変数を導入することで問題を1階常微分方程式の形に直せ.

B1. 上で直したIVPを、Runge-Kutta法で解いて数値解を求めよ.

B2. 〃 線形多段階法(ただし、3次の予測子・修正子法とする)を用いて数値解を求めよ.

注. コンピュータ言語, プログラム, 計算結果に関する注意はReport 1と同じとする.

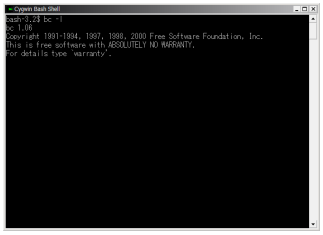
unix で使える簡易計算コマンド bc

ver. 1.1, April 26, 2009, 21:46:29+09, 降旗 大介 (大阪大学)

Abstract

Unix 系 (cygwin 含む) では，bc というコマンドがあり，これを用いると簡単な計算 (四則演算，三角関数, 指数，対数関数ぐらい) ができる．

起動方法 まずは Unix 系のターミナルを呼び出そう (cygwin の場合は cygwin を起動する)．そして，コマンドとして，`bc -l` と打ち込む． ちなみに，`-l` (ハイフン エル) を付けないと，各種の関数計算ができない．



このようになれば OK だ．ちなみに，プロンプトは出ない．

終了方法 bc 中で，`quit` とするか， CTRL-d (CTRL キーを押しながら d を押す) とする．

使い方 例えば，bc 中で 10/3 を計算したければ，そう入力するだけ．

```
10/3 ← 入力
3.3333333333333333333
```

などとなる．

各種関数 bc は `-l` をつけて起動した状態で，次の関数や計算などが使える．

`+-*/`	四則演算
`x % y`	x を y で割った余り
`x ^ y`	x の y 乗
`s(x), c(x)`	$\sin(x), \cos(x)$ (ラジアンで)
`a(x)`	$\arctan(x)$
`l(x)`	$\log(x)$ (自然対数)
`e(x)`	$\exp(x)$
`j(n,x)`	n 次ベッセル関数

より詳しくは，bc の外で `man bc` などとしてマニュアルを参照すべし．`if` や `for` など，場合わけや反復に便利なコマンドが多数ある．

知っていると便利

- 最後の計算値は .(ピリオド) か `last` で引用できる．つまり，cos(sin(0.5)) を計算するのに，

  ```
  0.5  ← 入力
  .5
  s(.) ← 入力
  0.47942553860420300027
  c(.) ← 入力
  .88726005071765262797
  ```

  などとできるので，計算過程が長いときも途中の結果を一生懸命入力しなくてよい．
- π を入力したいときは `4*a(1)` とすればよい．何回も使うならば，

  ```
  pi = 4*a(1) ← 入力
  ```

  と定義しておくと楽．
- `define` を使うと自分で関数を定義できる． 例えば，

  ```
  define f(x) { return 3*(x^2)+1 } ← 入力
  ```

  とすると $f(x) = 3x^2 + 1$ と定義したことになり，

  ```
  f(3) ← 入力
  28
  ```

  などとして使える．

gnuplot について超簡単なガイド

降旗 大介

Ver.1.2, 2009-06-08 00:10:14+09

1 Introduction

15 年以上前から存在する由緒正しいグラフソフトである. 理系学部の学生の基本ツールの一つ. グラフについては, こいつが使えればかなりの場面でなんとかなる. windows 用も存在する. 文字列からなるコマンドを用いて動作を指定する, という方式のため, 最初は取っつきにくいと思う人もいるかな. しかし, 扱える出力ファイル形式が多い, 動作の履歴が残る, 膨大かつ複雑な処理を一括して自動的に行なうことが可能である, 等の理由により, 理系必須のソフトウェアと言うべき地位を築いているソフトウェアである. 是非使いこなせるようになっておきたい.

1.1 参考資料等

gnuplot に関しては, 次の参考文献が役に立つだろう.

- 「gnuplot パーフェクト・マニュアル」… 川原 稔 著, ソフトバンク パブリッシング, 1999, ISBN 4-7973-1145-2, 1,800 円.
- 「使いこなす gnuplot」(改訂新版) … 矢吹 道郎 監修, 大竹 敢 著, テクノプレス, 2000, ISBN 4-924998-38-9, 2,500 円.
- `http://www.gnuplot.info/` … 本家
- 「GIMP/GNUPLOT/Tgif で学ぶグラフィック処理」… 皆本氏, 坂上氏著, サイエンス社, 1999, ISBN4-7819-0934-5, 2,200 円. … bk1 による書評
- Gnuplot のこれだけは! `http://phys.miyazaki-u.ac.jp/math-l/shige/html/ta.html`… 宮崎大学 矢崎氏の web
- GNUPLOT - not so Frequently Asked Questions - `http://t16web.lanl.gov/Kawano/gnuplot/index.html` … 河野氏@lanl の web
- グラフは Gnuplot にお任せ `http://ayapin.film.s.dendai.ac.jp/~matuda/Gnuplot/gnuplot.html` … 東京電機大学 松田氏の web
- gnuplot `http://oku.edu.mie-u.ac.jp/~okumura/linux/?gnuplot` … 松阪大学 奥村晴彦氏の web. 欠損データの扱いなども載っている. さすが.
- gnuplot のページ (Takeno Lab) `http://takeno.iee.niit.ac.jp/%7Efoo/gp-jman/gp-jman.html` … 新潟工科大学の 竹野研メンバーによる gnuplot 3.7.X, 3.8X, 4.X のマニュアルの日本語訳など. とても役に立つので目を通そう.

理系学生としては特に gnuplot の習熟は必須であるので, 少しでも良い参考資料を入手し手元に置いておくようにしよう. また, ネットワーク検索で,「gnuplot 書籍」「gnuplot コマンド」などと検索するのもよいだろう.

2 ごく簡単な使い方

2.1 試してみる

gnuplot を起動後，そのなかでコマンドを打つことで様々な動作を行える．以下，試してみよう．

1. `plot sin(x)`
   と入力してみよ．sine 関数のグラフが描かれるだろう．gnuplot にはこうした「関数を描く」機能がある．

2. 関数は既に与えられているもの (gnuplot で help functions と入力すればどんなものがあるか調べられる) の他に，自分で作ることもできる．例えば，
   `f(x) = 1 / (x*x)`
   として，
   `plot f(x)`
   としてみればよく分かるだろう．ちなみに，plot に使う変数はデフォルトでは x と決まっているので， 関数の定義には x を使っておけばよい．
   (注) ちなみに，二項演算は `+`, `-`, `*`, `/` で表され，巾乗 a^b は `a**b` で表される．

2.2 plot の様々なオプション

また，関数を描く機能 `plot` に対して，細かくいろいろ変えたいという場合は次のようにすればよいだろう．

- もう一度 (同じ関数等を) 描きたい．
  `replot`
  とすれば, 最も最近の `plot` か `splot` を再実行する．

- グラフの描き方を変えたい．
  gnuplot に関数を描かせると通常は線分で繋がれるが，これを点で表したい時などは次のようにすればよい．
  `plot sin(x) with points`
  `with` に続けて使えるオプションには `points` の他に `lines`, `linespoints`, `impulses`, `steps`, `dots`, `boxes` などがある．

- 描画する範囲を変えたい．
  `plot [-3:3] [-1.2:2.0] sin(x)`
  等とすればよい．一つ目の `[a:b]` が x 軸の範囲を表し，二つ目が y 軸の範囲を表す．

- 表示に使う「点」の数を増やしたい．
  「点」の数が少なすぎてグラフが粗くなって，正しく表示されないというような場合は， `plot` の前に
  `set samples` 使う点数
  を入力しておけば良い．

- 二つ以上の関数を同時に表示したい．
  `plot sin(x), cos(x)`
  などと ,(カンマ) で区切って書くだけで良い．

2.3 数値データをグラフに

gnuplot にはデータを (ファイルから読込むなどして) グラフとして描くという機能ももちろんある．試してみるには，例えば，

```
1   1.0
2   4.0
```

```
3   9.2
5   8.3
```

という内容のファイルを dummy.dat という名前で作っておいて， gnuplot 上で
`plot "dummy.dat"` とすると，(x,y) = (1, 1.0) ... の 4 つの点が打たれたグラフが得られる．このファイルだと「点だけなので見にくい」という人は，先の `with` オプションを使って，
`plot "dummy.dat" w l`
としてみるとよい．
(注) `w l` は， `with lines` の略である．このように，オプション等を区別できる範囲で略しても gnuplot はできるだけ解釈してくれる．

2.4 グラフを保存したい

グラフを保存することももちろんできる．ただ，画像ファイルの種類がいくつもあるので，明示的に指定しないといけない．
それも込みで，グラフをファイルに保存するには，

1. 保存する画像形式を指定する．
2. 保存するファイル名を指定する．
3. (もう一度) 画像を描く
4. 画像形式とファイル名を「リセット」する．リセットしないと，次に画像を描いたときに動作がわからなくなる．

という手順が必要である．今見えている画像をただちにファイルへ保存する機能は存在しないので，ここからの解説をよく読むこと!

2.4.1 画像形式の指定方法

まず，保存する画像の形式を指定する必要がある．その為には，gnuplot 上で
`set terminal` 画像形式 オプション
として指定する．使える画像形式とオプションは非常に多く，詳細はマニュアルを見てもらうとして， 実際には次のような組み合わせを知っていれば充分だろう．

表 1: 画像形式の指定例

画像形式	コマンド組み合わせ
png	`set terminal png color`
Postscript	`set terminal postscript eps 22`
リセット (unix の場合)	`set terminal x11`
リセット (windows の場合)	`set terminal windows`

2.4.2 出力するファイル名の指定方法

次に，画像を出力 (保存) するファイル名を指定しないといけない．それには，gnuplot 上で
`set output` ファイル名
として指定する．
ファイル名指定のリセットには，
`set output`
とすればよい．

2.4.3 よくわからないよ?

結局よく分からん… という者も多いだろうから，例をあげておこう．今描かれている図をファイルに保存したい，という時はとにかく次のようにすればよい．

表 2: 画像をファイル化するコマンド組み合わせ例 (ファイル名を `hoge.*` としている)

画像形式	コマンド (順番に実行する)

png	`set terminal png color`
	`set output 'hoge.png'`
	`replot`
	`set terminal win`
	`set output`

Postscript	`set terminal postscript eps 22`
	`set output 'hoge.eps'`
	`replot`
	`set terminal win`
	`set output`

Ruby について超簡単なガイド

降旗 大介

Ver.1.5, 2009-06-09 22:03:49+09

1 Introduction

Ruby とは，結構良くできているスクリプト言語，といえばいいのか．とりあえず簡単なことを簡単にできる，コンピュータ言語と思っていただけばよい．何かを数える，なんていう用途にはピッタリだ．

2 ごく簡単な使い方

Ruby では，プログラムをテキストファイルとして作成して，それを読み込むという形をとるのがまずは簡単だ．そこでそのようにガイドしよう．

(使い方の手順)

1. まずは unix 環境が必要だ．阪大教育用端末では，cygwin がその代用として使えるので，cygwin を立ち上げよう．こいつは一旦立ち上げたら授業が終わるまで使いっぱなしで構わない．

2. 次にテキストエディタを立ち上げよう．windows だとノートパッドでもよい．Emacs が使えるという人は Meadow がよいね．

3. これから作業するディレクトリ (フォルダ) を決めよう．これから作るファイルが消えたりしないところにしよう．また，フォルダ名に日本語が入っていると cygwin がとても面倒なことになるので，そうでない場所にしよう．

   そして，cygwin の作業ディレクトリもそこに一致させよう．具体的には，cygwin の中で

   ```
   cd "フル path でのフォルダ名"
   ```

   とすればよい．

4. 最初のプログラムを書こう．テキストエディタで次のように一行だけ書いてみよう．

   ```
   p 3+5
   ```

5. せっかく書いたプログラムだ，名前をつけて保存しよう．エディタの機能で「save」とか「保存」を選べば名前をつけてファイルとして保存できる．ファイル名は今回は “test.rb” としておこう．

6. プログラムを動かしてみよう．cygwin の中で，

   ```
   ruby -w test.rb
   ```

   と打ってみよう．うまくいけば

   ```
   8
   ```

   という答えが返ってくるだろう．

7. あとはこの場合，`test.rb` ファイルの中身を適当に書き換えてまた `ruby -w test.rb` としてみることを繰り返せばいろいろチャレンジできる．

3　超簡単な文法解説

名称	解説, 注意	サンプル
数学したい	数学計算を行なう準備 (1 度でよい)	include Math
-	pp を使うための準備 (1 度でよい)	require 'pp'
表示 1	画面に内容を出す	p ほにゃらら
表示 2	p より見やすい	pp ほにゃらら
表示 3	丁寧に表示したい	print ほにゃらら
代入	変数に数字などを入れる	a = 3.5
掛け算	-	a*b
べき乗	-	2**3　(8 になる)
絶対値	-	a.abs　(a の絶対値)
切り上げ	-	a.ceil　(a の切り上げ)
対数	自然対数	log(a)
指数	-	exp(a)
乱数	0 以上 1 未満	rand()
変数に足す	-	a += 1 (a に 1 足す)
変数をコピー	-	a = b (a に b の中身をコピー)
文字列をコピー	文字列はちと特殊だよ	a = b.dup(a に b の中身をコピー)
文字列のある 1 文字	数え方が 1 ずれる	str[3].chr(左から 4 文字目の場合)
文字列の一部	数え方が 1 ずれる	str[1..4](2〜5 文字目まで)
文字列一部削除	数え方が 1 ずれる	s.slice!(3..6)(4〜7 文字目を削除)
文字を整数に	-	s.to_i
文字を実数に	-	s.to_f
数字を文字に 1	便利なこともある	a.to_s
数字を文字に 2	() に 4 と入れると 4 進数表示に	a.to_s(4)
文字の数 1	全部で何文字？	s.length
文字の数 2	特定の文字の数	s.count("a")
単なる集合	-	[a,b,3] など
集合の要素	数え方が 1 ずれる	a[3] は 4 番目の要素を指す
集合に要素追加	-	group.push(a)
ペアの集合 (ハッシュ)	ペアを集めたもの	hash = { "a" => "3", "b" => "2"}
ペア集合中身の作業	ペアを一つずつ列挙して作業	hash.each { |k, v| 作業 }
最大, 最小値	集合の中身の最大, 最小値	[3,5,2].min　など
数えながら繰返す	ループだね	for i in 1..5 do ほにゃらら end
数えながら繰返す 2	上と同じ動作	1.upto(5){|i| ほにゃらら }
数えながら繰返す 3	減らす方向に	5.downto(1){|i| ほにゃらら }
もしも	条件があうなら動作	if ほにゃらら then ほい else ほい 2 end
もしもループ	条件があう間はループ動作	while ほにゃらら do ほい end
一致してる?	比較する	a == b
関数作成	できると便利	def ほにゃらら (引数) 中身 end
関数の出力は	関数定義の中で使おう	return 出力
動作時のパラメータ	ruby test.rb 3 10 とするなど	ARGV[i] で i+1 個目
コメント	メモ書きができるよ	# ほにゃらら
注意	大文字小文字は区別あるよ	
注意	変数名を大文字で書くと中身変更不可	
注意	大域変数は $ で始める	

4　簡単なサンプル

4.1　エントロピー計算

例えば，5 つの値を取る確率が

x	1	2	3	4	5
$P(X=x)$	1/20	1/5	1/2	1/5	1/20

となるような確率変数 X の情報源のエントロピー

$$H(X) = \sum_{x \in \chi} P(X=x) \log \left(\frac{1}{P(X=x)} \right)$$

を求めるには，次のようなプログラムが書ける．手で計算するよりはるかに速くて簡単．なお，底が 2 の対数関数を `lg` と表記する慣習があるようなので，覚えておくと楽かも．

```
include Math

def lg(p)
  return log(p)/log(2.0)
  end

def entropy(array)
  h = 0.0
  array.each{|p|
    h -= p * lg(p)
    }
  return h
end

X = [0.05, 0.2, 0.5, 0.2, 0.05]

Entropy_X = entropy(X)

print "H(X) = ", Entropy_X," bit\n"
```

4.2 Newton 法

$$\begin{pmatrix} x^+ \\ y^+ \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} - \frac{1}{2x + 2\sqrt{3}y} \begin{pmatrix} 1 & 2y \\ \sqrt{3} & -2x \end{pmatrix} \begin{pmatrix} x^2 + y^2 - 1 \\ \sqrt{3}x - y \end{pmatrix}$$

という反復式をつかう Newton 法を ruby で何も考えずにベタで書いてみよう．すると例えば以下のようになる (ああ，かっこわるいプログラムだね).

```
include Math

SQ = sqrt(3.0)

def f(arr)
  x = arr[0]
  y = arr[1]
  return [x**2 + y**2 -1.0, SQ*x - y]
end

def delta(arr)
  x = arr[0]
  y = arr[1]
  fx = f(arr)[0]
  fy = f(arr)[1]
  d = 2.0* (x + SQ*y)
  return [-(fx+2*y*fy)/d, (-SQ*fx+2*x*fy)/d]
end

def new(arr)
  return [arr[0]+delta(arr)[0], arr[1]+delta(arr)[1]]
end

ini = [2.0,0.0]
p ini

a = ini
for i in (1..10)
  b = new(a)
  p b
  a = b
end
```

5　ペア集合の each ってのがわかんないんだけど?

これはペア集合の中身であるペアを一つ取り出して仮の名前をつけ，指定した作業を行なったら次のペアを取り出して同様のことを繰り返すというものだ．ペアを列挙して操作するのにはピッタリの機能で，数学科の学生にはとても便利だ．
具体例で見た方がわかりやすいかな．例えば，

```
require 'pp'
a = 1.5
b = 2.5
hash = {1=>2, 2=>4, 3=>8, 4=>16, 5=>32}

hash.each {|left,right|
  print "Original pair = (", left, ", ", right, ")  : "
  print "Modified pair = (", a*left, ", ", b*right, ") "
  print "\n"
  }
```

というプログラムを考えてみよう．これは数学っぽく書けば $\{(1,2),(2,4),(3,8),(4,16),(5,32)\}$ という集合に hash という名前をつけ，その中身のペアを一つ取り出したときに仮に $(\mathrm{left},\mathrm{right})$ という名前で呼んで処理をしていることに相当する．
実際にこのプログラムを動かしてみると，

```
Original pair = (5, 32)  : Modified pair = (7.5, 80.0)
Original pair = (1, 2)  : Modified pair = (1.5, 5.0)
Original pair = (2, 4)  : Modified pair = (3.0, 10.0)
Original pair = (3, 8)  : Modified pair = (4.5, 20.0)
Original pair = (4, 16)  : Modified pair = (6.0, 40.0)
```

のような結果が出力される．だいたいこれでやっていることがわかるだろう．
そうそう，上の結果をみるとわかるが，この each は中身が「どういう順番で取り出されるかはわからない」ので，そこは気をつけておこう．
(備考) 単純な集合とかにも each は使えるので覚えておくと便利だ．

6　マニュアルや，入門コースとか無いの?

もちろんありますデスよ．ついでに，ruby についての本家 Web も示しておきましょう．

本家

http://www.ruby-lang.org/ja/

公式マニュアル

http://www.ruby-lang.org/ja/man/html/

チュートリアル (入門コース)

http://www1.tf.chiba-u.jp/~shin/tutorial/index.rb

Ruby ではじめるプログラミング (これもわかりやすいよ)

http://jp.rubyist.net/magazine/?0002-FirstProgramming

マニュアルをダウンロードしよう

http://www.ruby-lang.org/ja/documentation/