情報科学 2006　久野クラス　＃10

久野　靖*

2007.1.12

はじめに

あけましておめでとうございます。前回までで言語処理系ものを終わったので、あとはやるべき内容で取り上げていないものを「落ち穂拾い」として扱って行きます。できればそれを 2 回で済ませて、最終回は Java ならではの内容を取り上げて終わりたいところです。

今回は数値解析 (誤差、数値積分、区間 2 分法などの話題がこれに属します) で残っている話題として、連立方程式、常微分方程式、そして乱数の話題を取り上げます。

1　連立方程式の数値解法

1.1　消去法

連立方程式とは何かとかいう説明はいいですね? たとえば次に 3 元連立一次方程式の例を示す。

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ 2x_0 &+ 5x_1 &+ 4x_2 &= 4 \\ 4x_0 &+ 8x_1 &+ 8x_2 &= 7 \end{aligned}$$

これをどうやって解きますか? まず、一番上の式を 2 番目、3 番目から引く (3 番目については 2 倍してから引く) ことで x_0 の係数を消去できる。

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ &+ 2x_1 &+ 2x_2 &= 3 \\ &+ 2x_1 &+ 4x_2 &= 5 \end{aligned}$$

同様にして 2 番目の式を 3 番目から引くことで x_1 の係数を消去。

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ &+ 2x_1 &+ 2x_2 &= 3 \\ & &+ 2x_2 &= 2 \end{aligned}$$

これで $x_2 = 1$ が求まったのでそれを 2 番目、1 番目に代入できる。

$$\begin{aligned} 2x_0 &+ 3x_1 & &= -1 \\ &+ 2x_1 & &= 1 \\ & & x_2 &= 1 \end{aligned}$$

引続き $x_1 = 0.5$ が求まったのでそれを 1 番目に代入できる。

$$\begin{aligned} 2x_0 & & &= -2.5 \\ & x_1 & &= 0.5 \\ & & x_2 &= 1 \end{aligned}$$

これで $x_0 = -1.25$ と全て求まった。このように (1) 順に係数を消去していき (前進消去)、最後まで行ったら (2) 値を逆順に代入していく (後退代入) ことで連立一次方程式を解くことができる。これを **Gauss の消去法**と呼ぶ。

実際にこれをプログラムで扱う場合は、変数の名前はどうでもいいわけなので、係数だけをデータとして与える。ここではファイルに次のような形式で入れておいて読み込ませることにする。

*筑波大学大学院経営システム科学専攻

```
3          ←次数
2 3 2 1    ←各行のデータ。もちろん実数値でよい。
2 5 4 4
4 8 8 7
```

プログラムとしては、単にこれを読み込み、前進消去、後退代入を行って、最後に結果を出力するだけ。Java プログラムを次に示す。

```
import java.io.*;

public class R10Sample1 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new FileReader(args[0]));
    int n = new Integer(in.readLine()).intValue();
    double[][] a = new double[n][n+1];
    for(int i = 0; i < n; ++i) {
      String[] b = in.readLine().split(" ");
      for(int j = 0; j <= n; ++j) { a[i][j] = new Double(b[j]).doubleValue(); }
    }
    for(int i = 0; i < n; ++i) {
      for(int j = i+1; j < n; ++j) {
        double r = a[j][i] / a[i][i];
        for(int k = i; k <= n; ++k) { a[j][k] -= a[i][k]*r; }
      }
    }
    for(int i = n-1; i >= 0; --i) {
      a[i][n] = a[i][n] / a[i][i]; a[i][i] = 1.0;
      for(int j = 0; j < i; ++j) { a[j][n] -= a[j][i]*a[i][n]; }
    }
    for(int i = 0; i < n; ++i) { System.out.println(a[i][n]); }
  }
}
```

ではこれを動かしたところを見よう。

```
% cat test.data
3
2 3 2 1
2 5 4 4
4 8 8 7
% java R10Sample1 test.data
-1.25
0.5
1.0
%
```

確かに解が求まっている。ところで、つぎのデータだとどうだろうか。

```
% cat test1.data
3
2 3 2 1
2 3 4 4
4 8 8 7
% java R10Sample1 test1.data
NaN
NaN
NaN
%
```

何がいけないのだろう。この場合、2 番目の式から 1 番目の式を引くと最初の 2 つの係数がともに 0 になる。そのため、係数 0 を消去しようとして 0 で割り算してしまう。でもこの連立方程式が解けないということは全くなくて、次のように順番を入れ換えれば問題なくできる。

```
% cat test2.data
3
2 3 2 1
4 8 8 7
2 3 4 4
% java R10Sample1 test2.data
-0.25
-0.5
```

```
1.5
%
```

もちろん人間が順番を考えるようでは困るので、実際にはプログラムで次に消去しようとする係数が 0 だったら別の行と入れ換えてから消去を行う必要がある。さらに、完全に 0 でなかったとしても、非常に 0 に近い (絶対値の小さい) 値だと誤差が出やすくなるので、常に絶対値の大きい行を選んで来てそれで消去を行うのがよい。これを**ピボット選択**と呼ぶ。[1]ただし、ピボット選択を行おうとしても、すべての係数が 0 で選べないこともある。これは、方程式がもともと不定/不能の場合に起きる。

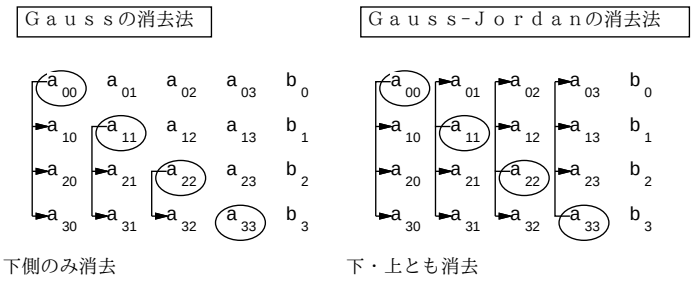

図 1: Gauss-Jordan の消去法

ところで、Gauss の消去法では処理が前進消去と後退代入の 2 つの段階に分かれていたが、消去のときにこれまでのように「自分より下の行について消去する」代わりに「自分以外のすべての行について消去」することで 1 段で解を求めることができ、プログラムがやや簡単になる (図 1)。これを **Gauss-Jordan の消去法**と呼ぶ (ただし計算量はやや不利になる)。

演習 1 上の例題をそのまま打ち込んで動かせ。動いたら、次のように改造してみよ。

a. Gauss-Jordan の消去法に直し、プログラムが短くなることを確認する。
b. ピボット選択を入れ、例題で駄目だったデータが扱えることを確認する。
c. 方程式が不能/不定のときにその旨表示する機能を追加する。

1.2 反復法

連立一次方程式の消去法とは別の原理による数値解法として、反復法と呼ばれるカテゴリのものがある。これは次の原理による。ここまでに見て来たような連立一次方程式を行列・ベクトルを使って書き直すと次のように書くことができる。

$$A\,\vec{x} \;=\; \vec{b}$$

ところで、行列 A を下三角行列 L、対角行列 D、上三角行列 U の 3 つに分解して考えると次のように変形できる。

$$(L+D+U)\,\vec{x} \;=\; \vec{b}$$

$$D\,\vec{x} \;=\; \vec{b} - (L+U)\,\vec{x}$$

$$\vec{x} \;=\; D^{-1}\{\vec{b} - (L+U)\,\vec{x}\}$$

D は対角行列だから逆行列は各要素を逆数にすればいいだけなのに注意。さて、連立一次方程式を解くということは、この式を満たす $\vec{x}$ を求めることに等しい。ところで、この式を漸化式と考え、適当な $\vec{x_0}$ から始めて次の式により値を計算していくものとする。

$$\vec{x_{i+1}} \;=\; D^{-1}\{\vec{b} - (L+U)\,\vec{x_i}\}$$

ここで $\vec{x_i}$ が収束すれば、つまり値が変化しなくなれば、その値 $\vec{x_*}$ は連立一次方程式の解となっている。これを **Jacobi 法**と呼ぶ。なお、上の漸化式は常に収束するとは限らないが、行列が対角優位 (対角成分が他の成分に比べて相対的に大きい) の場合には収束することが知られている (厳密な収束条件についてはその筋の本を調べてください)。

これを Java プログラムにしたものを示しておく。収束は $\vec{x_i}$ の各成分の前回との差の絶対値の和が 10^{-8} 以下になったことで判定し、100 回反復しても収束しない場合はあきらめるようにした。

[1]ただし、$2x+3y=1$ と $200x+300y=100$ とは同じ方程式なので、実際にはまず各行の係数を絶対値の最大が 1 になるように定数倍してそろえておき、それから最大を選択する方ががよい。これを**スケーリング**と呼ぶ。

```
import java.io.*;

public class R10Sample2 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new FileReader(args[0]));
    int n = new Integer(in.readLine()).intValue();
    double[][] a = new double[n][n+1];
    for(int i = 0; i < n; ++i) {
      String[] b = in.readLine().split(" ");
      for(int j = 0; j <= n; ++j) { a[i][j] = new Double(b[j]).doubleValue(); }
    }
    double[] x = new double[n]; double d = 1.0; int count = 0;
    while(d > 0.00000001 && count < 100) {
      double[] x1 = new double[n]; d = 0.0;
      for(int i = 0; i < n; ++i) {
        double v = a[i][n];
        for(int j = 0; j < n; ++j) { if(j != i) { v -= a[i][j]*x[j]; } }
        x1[i] = v/a[i][i]; d += Math.abs(x1[i]-x[i]); ++count;
      }
      x = x1;
    }
    if(count >= 100) { System.out.println("NO"); }
    else { for(int i = 0; i < n; ++i) { System.out.println(x[i]); } }
  }
}
```

なお、Jacobi 法では「次の」ベクトルを完全に計算し終わってから現在のものと入れ換えるが、上の例で言うと x1 を使わないで「x1[i] = …」を「x[i] = …」に直してしまい、計算し終わった成分は以後その新しい値を使う方法を **Gauss-Seidel 法**と呼び、その方が収束が速いことが知られている。

演習 2 上の例題を打ち込んでさまざまなデータで動かし、どのような場合に収束する/しないかを見てみよ。収束する場合は消去法のプログラムと結果がほぼ一致することを確認せよ。また、Gauss-Seidel 法に直した場合、収束までの計算回数がどれくらい違うかを観察せよ。

2 常微分方程式の数値解法

2.1 常微分方程式

常微分方程式とは、未知関数とその導関数から成る等式で定義される方程式 (微分方程式) のうち、未知関数が (本質的に)1 つの変数を持つ場合を言う。導関数が 2 次以上の時は高階常微分方程式、方程式が複数ある場合は連立常微分方程式と呼ぶ。ここでは一番簡単な場合として $\frac{dy}{dx} = f(x, y)$ の形を持つものを扱う。

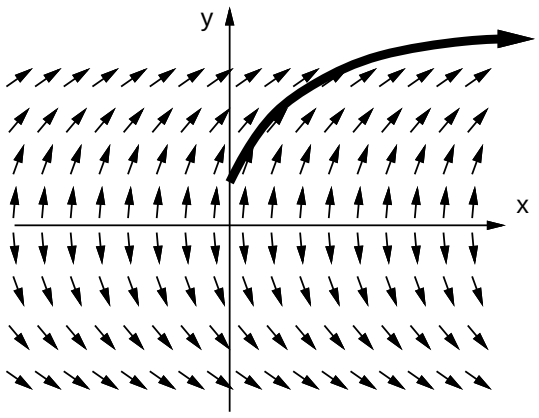

図 2: 常微分方程式と初期値問題

直観的に説明するなら、XY 平面の至るところにおいて傾きを表す矢印が定義されていて、その方向に沿った曲線を求める問題、という風に考えることができる (図 2)。もちろん、そのような曲線は無限にあるが (一般解)、任意の点 (x_0, y_0)

を与えてそこを通る曲線を求めるとすれば、その曲線は 1 つだけになる (特殊解)。この、初期値を与えて特殊解を求める問題を**初期値問題**と呼ぶ。

簡単な例として、次の常微分方程式を考える。

$$\frac{dy}{dx} = \frac{1}{2y}$$

これを次のように変形する。

$$2y\ dy = 1\ dx$$

両辺を不定積分する。

$$\int 2y\ dy = \int 1\ dx$$

両辺とも積分の結果次のようになる。

$$y^2 = x + C$$

よって次の式が求まり、これが一般解となる (もちろん $y = 0$ は除く)。

$$y = \pm\sqrt{x + C}$$

そして、たとえば点 $(1, 0)$ を通る特殊解を求めるなら、上式の y に 1、x に 0 を代入して $C = 1$ が求まるから $y = \sqrt{x + 1}$ が解となる。

2.2 Euler 法

前節のような簡単な常微分方程式の初期値問題は解析的に (数式の形で) 解が求まったが、現実の問題では解析的には解が求められないことが多い。そのような場合に、(数値積分と同様に) 数値的に解を求める方法 (数値解法) が使われる。

一番素朴な方法として、初期値 (x_0, y_0) における解曲線の接線の方向は $\frac{dy}{dx} = f(x, y)$ として分かっているわけなので、この方向に微小な値 h ぶんだけ動いた点を (x_1, y_1) とし、以下同様にして次々に曲線上の値を (近似的に) 求めて行くという方法が考えられる。これを **Euler 法**と呼ぶ。整理すると、次の式で x_i, y_i を計算していくのが Euler 法である。

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

これを使って x が指定値になるまで指定した分割数で計算して最後の y の値 ($\sqrt{x + 1}$ になるはず) を出力する Java プログラムを示す。

```
public class R10Sample3 {
  public static void main(String[] args) {
    double xmax = new Double(args[0]).doubleValue();
    int count = new Integer(args[1]).intValue();
    int count = new Integer(args[0]).intValue();
    double h = xmax/count, x = 0.0, y = 1.0;
    for(int i = 1; i <= count; ++i) {
      x = h * count;
      y = y + h * 0.5/y; // f(x,y) == 1/(2y)
    }
    System.out.println(y);
  }
}
```

実際に動かしてみよう。x として 1 を指定することで、2 の平方根を計算するようにし、刻み数を変えることで精度の変化を見ている。

```
% java R10Sample3 1 10
1.420519799291044
% java R10Sample3 1 100
1.4148279673659128
% java R10Sample3 1 1000
1.4142748458924588
%
```

まあ、あんまりよくはない。覚えていない人のため、1 桁の平方数でない数の平方根を掲げておく。

$\sqrt{2} \approx 1.41421356237309504880168872420969807856 96$
$\sqrt{3} \approx 1.73205080756887729352744634150587236694 28$
$\sqrt{5} \approx 2.23606797749978969640917366873127623544 06$
$\sqrt{6} \approx 2.44948974278317809819728407470589139196 59$
$\sqrt{7} \approx 2.64575131106459059050161575363926042571 02$
$\sqrt{8} \approx 2.82842712474619009760337744841939615713 93$

double 型の精度は 16 桁くらいなので、もうちょっと精度良く求まって欲しいわけだが… Euler 法の場合、「個々の点における」接線を延長して次の点を求めているため、微分の向き (矢線の向き) が揃っている (まっすぐな) 所ではいいけれど、カーブしているところではどんどん「外側」にずれて行ってしまう。これを補う方法を次に述べる。

2.3 Runge-Kutta 法

Euler 法では各ステップの始点における接線を延長して次の点を求めていたが、それでは解曲線がカーブしているときに誤差が大きい。そこで、始点と終点の両方で接線の向きを求め、それを平均すればより正確になると考える。しかし実際には、各ステップの終点は「これから求める」ので分からない。そこで Euler 法で終点の近似値を求め、そこの傾きを使い、これと始点での傾きとの平均を取る (図 3)。つまり次のようにするわけである。

$$x_{i+1} = x_i + h$$
$$k1 = h\ f(x_i, y_i)$$
$$k2 = h\ f(x_i + h, y_i + k1)$$
$$y_{i+1} = y_i + \frac{1}{2}(k1 + k2)$$

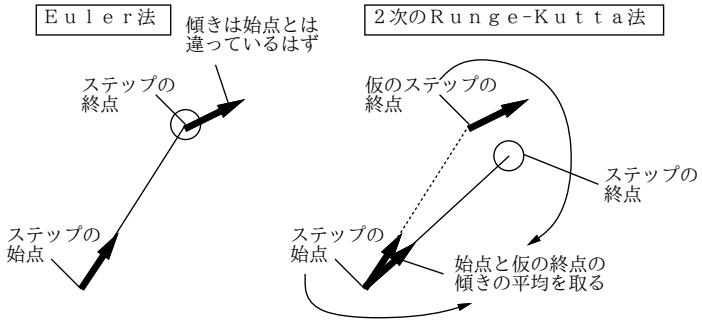

図 3: Euler 法と 2 次の Runge-Kutta 法

これを **2 次の Runge-Kutta 法**と呼ぶ。これを Java プログラムにしたものを示しておく。

```
public class R10Sample4 {
  public static void main(String[] args) {
    double xmax = new Double(args[0]).doubleValue();
    int count = new Integer(args[1]).intValue();
    double h = xmax/count, x = 0.0, y = 1.0;
    for(int i = 1; i <= count; ++i) {
      x = h * count;
      double k1 = h * 0.5/y; // f(x,y) == 1/(2y)
      double k2 = h * 0.5/(y+k1);
      y = y + 0.5 * (k1+k2);
    }
    System.out.println(y);
  }
}
```

これによる計算を示しておく。ずっと精度がよくなっていることが分かる。

```
% java R10Sample4 1 10
1.4142157109943851
% java R10Sample4 1 100
1.4142135644527205
```

```
% java R10Sample4 1 1000
1.4142135623751702
%
```

さらに正確さを増すため、h の半分だけ上記の方法で進み、そこから終点までを 2 番目の近似値で進み、始点、最初の近似値、2 番目の近似値、終点の近似値を 1:2:2:1 の比率で混ぜる方法があり、**4 次の Runge-Kutta 法**として知られている。単に Runge-Kutta 法と言った場合はこちらを指すことが多い。

$$
\begin{aligned}
x_{i+1} &= x_i + h \\
k1 &= h\ f(x_i,\ y_i) \\
k2 &= h\ f(x_i + \tfrac{h}{2},\ y_i + \tfrac{k1}{2}) \\
k3 &= h\ f(x_i + \tfrac{h}{2},\ y_i + \tfrac{k2}{2}) \\
k4 &= h\ f(x_i + h,\ y_i + k3) \\
y_{i+1} &= y_i + \tfrac{1}{6}(k1 + 2k2 + 2k3 + k4)
\end{aligned}
$$

なぜ 1:2:2:1 の比率がいいかとかはここでは説明しないが、Taylor 展開による 1 次の項のあてはめが Euler 法、2 次の項まであてはめるのが 2 次の Runge-Kutta 法、4 次の項まであてはめるのが 4 次の Runge-Kutta 法である。

ついでに誤差についてもおおまかに説明すると、Euler 法では各ステップの誤差が (1 次まであてはめた残りなので)$O(h^2)$、それを $\frac{1}{h}$ ステップ累計するため全体の誤差は $O(h)$ となる。同様に 2 次の Runge-Kutta 法では各ステップ $O(h^3)$、全体で $O(h^2)$、4 次の Runge-Kutta 法では各ステップ $O(h^5)$、全体で $O(h^4)$ となる。これを言い替えると、h を $\frac{1}{2}$ にしたとき Eular 法では全体の誤差がほぼ $\frac{1}{2}$ になるのに対し、2 次の Runge-Kutta 法では $\frac{1}{2^2}$、4 次の Runge-Kutta 法では $\frac{1}{2^4}$ になる。

なお、刻みをどんどん細かくする程いいかというと、細かくする程計算時間が掛かるだけでなく、丸め誤差や情報落ち誤差が累積するので、結局精度もある程度以上は上がらなくなる。このため、2 次や 4 次の Runge-Kutta 法のように最初から性能のよい公式を使うことは大変有利だと言える。

演習 3 Eular 法と 2 次の Runge-Kutta 法のプログラムを打ち込み、さらに手直しして 4 次の Runge-Kutta 法のプログラムも用意せよ。これらでさまざまな値の平方根を刻み幅を変えて計算し、上で説明した誤差に関する性質が成り立っているかどうか確認してみよ。

3 乱数とランダムアルゴリズム

3.1 乱数とは

乱数 (random numbers) とは、「でたらめな数」である。ではあんまりだからもっと正確に言うと、乱数列とは「ある分布に従う、互いに独立な事象を表す、確率変数の実現値の列」を言い、乱数とはその中の 1 つの値を言う。互いに独立ということは、ある点までの乱数列が分かったからといって、次の乱数がいくつであるかは予測できないことを意味する。

また、**分布** (distribution) とは、どの範囲の値がどのくらい出現しやすいかを表す (図 4)。たとえば、区間 $[0, 1)$ の一様分布であれば、乱数の範囲は 0 以上 1 未満で、その間のどの数も同じくらいの確からしさで出現する。これを**一様乱数**という。また、偏りのないサイコロを振って出る目の数は 1 以上 6 以下の整数値だが、どの数も同じ確率で出現するため、これも一様乱数である。

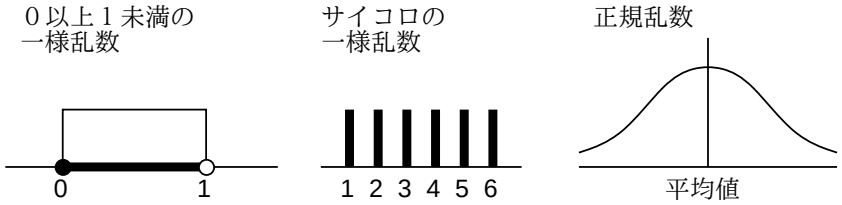

図 4: 乱数と分布

この他によく使われる乱数としては、正規分布に従う**正規乱数**がある。正規分布は中央が一番高く両側にすそを引いたツリガネ形の分布であり、試験の偏差値などでおなじみである (試験が受験者の集団に対して易しすぎたり難しすぎた

りヘンな問題だったりすると、分布が綺麗な正規分布でなくなるので偏差値による順位推定が役に立たなくて問題だったりする)。

3.2 疑似乱数

疑似乱数 (pseudorandom numbers) とは、計算機である計算を順次行うことによって生成される数値の列だが、乱数列のように思えるようなものを言う。既にさんざん学んで来たように、計算機による計算はプログラムによって完全に決定的に決まるので、「でたらめな」数を生成するのは思いのほか難しい。

疑似乱数のアルゴリズムとして知られているものには、たとえば次のものがある。

- 自乗採中法 — w ビットの数を 2 乗すると $2w$ ビットになるので、そこから中央付近の w ビットを取り出して「次の数」とする。これを繰り返すことで w ビットの乱数列を生成する。古くからあるがあまり良くはない。
- 線形合同法 — $x_{i+1} = (x_i \times a + c) mod m$ により次々に値を生成していく。良い疑似乱数とするためには、パラメタ a, c, m の選定に注意が必要。
- メルセンヌツイスター (MT) — 1997 年に松本　眞、西村拓士が開発した乱数アルゴリズム。

どのようなアルゴリズムでも、計算方法が決まっている以上、順次 x_i を生成していくうちに前に出て来たものが再度現れたら、それ以降の列は前と同じものの繰り返しになる。これを周期といい、もちろん周期が長いものが望ましい。MT は 32 ビットで $2^{19937} - 1$ という長い周期を保証できる点で画期的だった。周期の他に統計的独立性の検定などもクリアしている。

Java では `Math.random()` によって区間 $[0, 1)$ の一様乱数が得られるが、そのアルゴリズムたぶん線形合同法で、MT ほど良いとは言えないらしい。とりあえず、シミュレーションなどに使ってみる分には問題ないと思われるが。

このほか、最近では OS(コンピュータ上で常に稼働している基本ソフトウェア) が乱数機能を提供してくれている場合が多い。OS の乱数機能は、外部割り込み (ユーザのキーボード入力など) 等に基づいた「ランダムさ」を活用するので、疑似乱数のような周期の問題がない。ただし速い速度で乱数を生成消費すると「ランダムさ」の供給が追い付かなくなることがあるという弱点がある。また、最近では CPU チップ自体に物理的な乱数発生装置を持つものもある。Mac OS-X では「`new FileReader("/dev/random")`」により OS の乱数機能からバイト列を読み出すストリームを作成できる。

3.3 ランダムアルゴリズム

ランダムアルゴリズム (randomized algorithm) とは、(疑似) 乱数を活用して、ランダムなふるまいを持たせたアルゴリズムを言う。これに対し、通常の決定的な動作を行うアルゴリズムは**決定的アルゴリズム** (deterministic algorithm) と呼ばれる。

ランダムアルゴリズムは、設計によっては決定的アルゴリズムよりすぐれた性能を持たせることができる。たとえば、1 億要素の配列で「半分には値 a が入っているがその場所は分からない」場合と「まったく値 a が入っていない」場合とがあり、どちらであるかを判断する必要があるものとする。決定的アルゴリズムでは、たとえば先頭から順番に値 a があるかどうか見て行くことになるだろうが、値 a が全部後半に詰まっている可能性もあるので、最悪で 5 千万要素を見る必要がある。後ろから順に見るとしても、値 a が全部前半に詰まっているかも知れないので同じである。ここで、乱数を用いて 1 億の位置からランダムに 1 つ選び、その値が a かどうかを判断することを 1 万回繰り返したとする。その結果 1 回も値 a に遭遇しなければ、「値 a はない」と判断してまず問題ない。この判断が間違っている確率は $\frac{1}{2^{10000}}$ であり、それはこの計算をするコンピュータが故障する確率よりはるかに小さいのだから。

このような、微小だが 0 でない「間違う」確率を持ったアルゴリズムを**モンテカルロアルゴリズム** (Monte Carlo algorithms) という。[2]これに対し、間違うことはないが運が悪い場合に性能が低下するアルゴリズムを**ラスベガスアルゴリズム** (Las Vegas algorithms) という。[3]たとえば、クイックソートはピボットの選択が悪いと性能が低下することを述べた。そこで、どの要素をピボットとして用いるかを乱数で決めるようにすると、これはラスベガスアルゴリズムとなる。なぜなら、乱数がすべて「悪い要素 (その区間の最大や最小の要素)」を選び続ける確率は非常に小さいので、よほど運が悪くない限り高速に整列が行え、そして運が悪い場合は実行時間が長く掛かることになるが、整列はやはり正しく行えるからである。

[2]モンテカルロはヨーロッパにあるカジノで有名な都市の名前。
[3]ラスベガスは米国にあるカジノで有名な都市の名前。

3.4 モンテカルロ法

モンテカルロ法とは、**シミュレーション** (simulation) などにおいて乱数を活用する手法を言う。たとえば、交通の流れを実際に観察する代わりに、乱数を用いてランダムに車を (プログラム内で) 発生させ、それらの車がどのように流れて行くかを見ることで、さまざまな方法で交通信号を制御してよい方法を見つけ出すことなどができる。

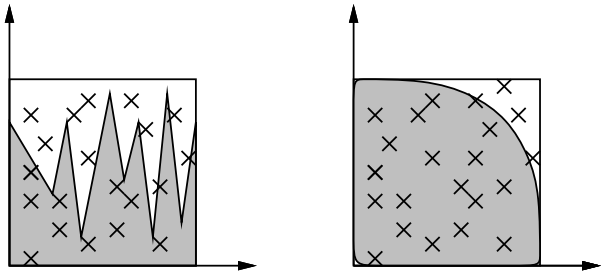

図 5: モンテカルロ法による数値積分

また、モンテカルロ法は数値積分にも使うことができる (図 5)。すなわち、積分する範囲の関数値の最大より大きい値を選んで長方形の領域を考え、その範囲内に乱数で多数の点を打ち、関数値より下にある点の比率を求める。積分とは「その関数の下側の面積を求める」ことだから、長方形の面積にその比率を掛けたものが積分値 (の近似値) として使える。

そんな面倒なことをするよりシンプソンのアルゴリズムでいいのでは? しかし、シンプソンのアルゴリズムなどは、対象とする関数が連続かつ微分可能 (なめらか) でないと使えない。そのような性質が期待できないような分野では、モンテカルロ法が有力な手法の 1 つとなる。

たとえば半径 1 の四分の一円の面積を求めて (それを 4 倍することで) π の近似値を計算してみよう (もちろん円周は十分連続かつ微分可能だけれどそれは置いておいて)。

```
public class R10Sample6 {
  public static void main(String[] args) {
    int count = new Integer(args[0]).intValue(), less = 0;
    for(int i = 0; i < count; ++i) {
      double x = Math.random(), y = Math.random();
      if(x*x + y*y < 1.0) { ++less; }
    }
    System.out.println((4.0*less)/count);
  }
}
```

実行させてみると次の通り。

```
% java R10Sample6 100000
3.13868
% java R10Sample6 1000000
3.140568
% java R10Sample6 10000000
3.1412964
%
```

有効数字 3〜4 桁ではしょうもないと思いますか? 実際には、3〜4 桁の有効数字が得られれば十分な仕事というのは結構多い。来年の GDP の生長率が 10.1 だろうと 10.2 だろうと 3 桁目はさして重要ではないでしょう?

演習 4 モンテカルロ法で数値積分を行うときの、精度 (有効桁数) と試行の数との関係について考察せよ。

3.5 乱数とゲーム

最後にお楽しみの話題としてゲームに言及しておこう。ゲームの中には、将棋や囲碁のように (先手後手を決める以外は) ランダム性を使わないものもあるが、多くのゲームでは (サイコロやカードのシャッフルなどを通じて) ランダム性を採り入れている。ランダム性を採り入れることで、ゲームの「場面」が毎回違ったものになり新鮮さが保たれ、また複数プレーヤで行う場合に「上手下手」以外の要因が入って下手な人にも勝つチャンスが生まれ勝負の行方に興味が持てるようになる。

ここでは簡単なゲームとして「数当て」を作ってみる。そのルールは次の通り。

- 計算機は内部で 4 桁の数を「思い浮かべ」る (4 桁の数字はすべて異なる)。
- プレーヤはその 4 桁の数を「当てる」ことをめざして、自分も 4 桁の数を入力する。
- 計算機は 2 つの 4 桁の数を照合して、「同じ位置に同じ数がある (これをヒットと呼ぶ)」個数と、「同じ数があるがただし違う位置にある (これをブローと呼ぶ) 個数とを数えて知らせる。
- プレーヤはその情報を見て再度チャレンジする。
- 10 回以内のチャレンジで当たればプレーヤの勝ち、さもなければプレーヤの負けとする。

Java プログラムを次に示す。4 桁の異なる数字を作る方法がちょっと分かりづらいかも知れない。ここでは文字列 "0123456789"からランダム番目を取って先頭に移すことを 4 回やってから先頭の 4 文字を取るようにした。この程度でも結構遊べるでしょう?

```
import java.io.*;

public class R10Sample7 {
  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String str = "", ans = "0123456789";
    for(int i = 0; i < 4; ++i) {
      int j = (int)(10*Math.random());
      ans = ans.charAt(j) + ans.substring(0,j) + ans.substring(j+1,10);
    }
    ans = ans.substring(0,4);
    for(int i = 1; i <= 10; ++i) {
      System.out.print(i + ": Your Guess?> ");
      str = in.readLine(); if(str.equals(ans)) { break; }
      int hit = 0, blow = 0;
      for(int j = 0; j < 4; ++j) {
        for(int k = 0; k < 4; ++k) {
          if(str.charAt(j)==ans.charAt(k)) { if(j==k) {++hit;} else {++blow;} }
        }
      }
      System.out.println("hit = " + hit + ", blow = " + blow);
    }
    if(str.equals(ans)) { System.out.println("You Won!!"); }
    else { System.out.println("You Lose!! correct answer = " + ans); }
  }
}
```

演習 5 乱数を使ったゲームを何か作ってみよ。上の例題の改良 (改造) 版でもよい。

A　本日の課題 10A

「演習 1」〜「演習 5」のどれか 1 つについて、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 10A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 「連立方程式」「微分方程式」の数値解法について分かりましたか。

   Q2. 乱数の活用ができるようになりそうですか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題はありません。