シミュレーション

カードコレクション

■ この話は，次の本から引用したものである．

『いやでも楽しめる算数』清水 義範，西原 恵理子著，講談社.

この本によれば，著者の清水氏がある雑誌で次のような問題を提起した．

> 昔，キャラメルのおまけで，アニメのキャラクターだった「赤胴鈴之助」のカードがあり,「赤」,「胴」,「鈴」,「之」,「助」の 5 種類があった．運が良い人は 5 個買うだけで全部揃うだろうが，運が悪い人はなかなか揃わないだろう．全部揃うまで平均で何個買えばよいだろうか．ここに，5 枚のカードは全て均等に入っているものとする．

数学的には期待値を求める問題である．無限を扱わなければならないが，特に難しくもなく，高校生レベルである．ただ，この本は数学 (もしくは算数) を早い段階で投げ出した人向けに書かれており，かなり面白い．著者は数学の専門家ではなく作家であるが，分かりやすい説明がなされている (ただし数学的にはやや不完全である). 新聞に紹介されたこともあり，多くの回答が寄せられ，そのうち半分程度が正解だったそうである．その回答の中に，かなり異質なものが一つあったという．かの回答者は，数学的な答はさっぱり分からないが，その代わりに計算機の中で,「大勢の仮想のコレクター」にカードを集めてもらい，その平均を取ったそうである．その数値は厳密な答にかなり近く，清水氏はその発想に驚嘆している[†]. 様々な場面でこのようなシミュレーションは重要な役割を担っている．

■ では，この「カード集めシミュレーション」を Mathematica でやってみよう．その前に，プログラムで用いるいくつかの組み込み関数について説明する．まず，シミュレーションに不可欠と思われる乱数についてである．`Random[]` は 0 から 1 の間の実数をランダムに与える．

```
Random[]
0.37109
```

(1)

この結果はもちろん人によって異なる．数値などの入力が不要という点で，`Random` という関数は特殊である．ところで，真の乱数を計算機で発生させることは不可能である．なぜなら，計算機は時刻などの情報を元に一定のアルゴリズムで値を算出するのであり，その意味で完全に予想不可能であるとは言えないからである[‡]. そこで，計算機で発生させる乱数を「擬似乱数」と呼ぶ．Mathematica ブックには次のように書かれている．

> `Random[]` で得られる数列は厳密には「真の乱数」ではないが，実用上「十分に乱数的」である．

この「実用上十分に乱数的」である数列をいかに発生させるかは，重要な研究テーマである．

話が脱線してしまったが，次のようにすると，1 から 5 までの整数が等しい確率で出力される．

```
Random[Integer,{1,5}]
2
```

(2)

[†]私 (後藤) は，計算機でシミュレーションできる程の人が数学的な答えが全く分からないことの方に驚いた．もともと文系の人が計算機を扱うようになることはそれほど珍しくないのかもしれない．

[‡]粗悪なアルゴリズムだと，例えば偶数と奇数が交互に現れる，という様な規則のある数列になる

察しの良い者なら分かるだろうが,「赤」,「胴」,「鈴」,「之」,「助」のカードの代わりに「1」,「2」,「3」,「4」,「5」のカードだと考えよう，というのである．上で 2 が返ってきたということは，おまけに「2」のカードが入っていたと解釈することができる．

FreeQ[L,x] は，リスト L に x が入っていなければ True を，入っていれば False を出力する．

```
FreeQ[{1,2,3},4]
True
```

(3)

■ いよいよプログラムに取り掛かろう．次の関数は，一人の仮想コレクターが 5 枚のカードをそろえるために何枚のカードを買ったかを与える．一応説明するならば，局所変数 n は「それまでに買ったカードの枚数」，L は「カードのコレクション」，card は「今買ったカード」を表す．最初の状態は n=0 でコレクションは 1 枚もない．コレクターはコレクションが 5 枚揃うまで買い続け，買ったカードがコレクションにない場合のみそのカードをコレクションに加える．コレクションが揃ったら，それまでに買ったカードの枚数 n を出力する．

```
collection[] := Module[{n,L,card},
   n = 0;L = {};
   While[Length[L] < 5,
     n = n + 1;card = Random[Integer,{1,5}];
     If[FreeQ[L,card],L = Append[L,card]]];
   n]
```

(4)

次は私が初めて実験した結果である．あまり運が良いとは言えないようだ．数値を何も入力しなくて良いのは Random と同様.

```
collection[]
16
```

(5)

n 人のコレクターの結果をリスト形式で出力するプログラム．

```
test[n_] := Module[{L,c},
   L = {};
   Do[c = collection[];
     L = Append[L,c],{i,1,n}];
   L]
```

(6)

まずは 100 人くらいに買わせてみる．最後のセミコロンは無駄にデータを出力させないためであり，後で 10000 人くらいに買わせるときには絶対に忘れない方が良い．このデータでは，最も運の悪い人は 23 枚買い，最も運の良い人は最小の 5 枚で済んだようである．最も運の良い人は全部で 4 人であった．

```
L = test[100];

Max[L]
23

Min[L]
5

Count[L, 5]
4
```

(7)

次に，リスト L の平均を与える関数を定義する．今回の平均は 10.89 であった．皆はどうなっただろうか．

```
average[L_] := N[Apply[Plus, L]/Length[L]]

average[L]
10.89
```

(8)

これではあまり精度が良くないので，今度は 10000 人に買わせて平均を取ろう．数秒時間がかかるが，これでやや厳密な答えに近くなるだろう．

```
L = test[10^4]; average[L]
11.3961
```

(9)

円周率

■ 次に話題を変えて，円周率を求める実験をしてみよう．その前に，用いる組み込み関数について説明する．`Table[exp,{m},{n}]` は exp が n 個並んだリストが m 個並んだリストを出力する (少しややこしいが，例を見れば分かるだろう)．

```
Table[1, {2}, {3}]
{{1, 1, 1}, {1, 1, 1}}
```

(10)

`Select[L,condition]` はリスト L の中から条件 condition に合うもののリストを出力する．1 つ目の例は偶数を，2 つ目の例は 1 より大きいものを選んでいる．2 つ目の例は記号 `#,&` を用いて何だか見慣れないが，これは無名関数と呼ばれる手法を利用している．ここでそれを詳しく説明している余裕はないので，このような書き方をするのだな，と思って欲しい．

```
Select[{1, 2, 3}, EvenQ]
{2}

Select[{1, 2, 3}, # > 1 &]
{2, 3}
```

(11)

■ 次は平面の領域 $(-1,1)\times(-1,1)$ 内にランダムに n 個の点を取る関数である．試しに 3 個の点を取ってみる．

```
ten[n_] := Table[Random[Real, {-1, 1}], {n}, {2}]

ten[3]
{{0.96375, -0.809976}, {0.0409335, -0.589838}, {-0.907943, -0.50237}}
```

(12)

次に 1000 個の点をプロットしてみる．ほぼ一様に分布しているのが分かるだろう．それから，1000 個のうち，単位円内にある点の個数を数える．

```
t = ten[10^3]; ListPlot[t];

Length[Select[t, #[[1]]^2 + #[[2]]^2 < 1 &]]
771
```

(13)

正方形の領域の面積が 4，その内部の単位円の面積が π であるから，正方形内に適当に点を取ったときに，それが単位円の中に入る確率は $\pi/4$ である．よって，次の計算で円周率が計算できると期待できる．

```
N[4 %/1000]
3.084
```

(14)

まあ，あまり精度は良くないようだ．もっと点の個数を増やしたらどうだろうか．また同じ事をやるのは面倒なので，関数を定義してしまおう．

```
enshu[n_] := Module[{t, c},
   t = ten[n];
   c = Length[Select[t, #[[1]]^2 + #[[2]]^2 < 1 &]];
   N[4c/n]]
enshu[10^4]

enshu[10^5]
```

(15)

少しは精度が上がったのではないだろうか．

近似値と誤差

■ 計算機を扱う際に常に注意しなければならないこととして，近似値は所詮近似値であり，それによって誤差が生まれる可能性がある，ということがある．簡単な例として，次を実行してみよう．$7\times 1/7 = 1$ であるはずなのに，0.142857 は近似値であるから 7 を掛けても 1 にはならない．

```
N[1/7]
0.142857

7 * 0.142857
0.999999
```

(16)

次は同じ事をしているようだが，結果は異なる．

```
N[1/7]
0.142857

7 * %
1.
```

(17)

なぜ今度は 1 になるのだろうか．実は出力が 6 桁であっても，Mathematica 内部ではもっと先まで計算しており，7 を掛けた際にほとんど 1 になる，と判断する．この辺りの事情は次を実行してみれば分かる．

```
N[1/7]
0.142857

% - 0.142857
1.42857 × 10^-7
```

(18)

このように，Mathematica は誤差を感じさせない工夫をいろいろとしているが，いつでもうまくいくと勘違いしてはいけない．

■ 5 次方程式 $x^5 + 2x + 1 = 0$ の解の近似値の 1 つを a とおき，それを $x^5 + 2x + 1$ に代入する．0 になるべきところが，少し誤差が出ている．

```
NSolve[x^5 + 2x + 1 == 0, x]
{{x → -0.701874 - 0.879697 i}, {x → -0.701874 + 0.879697 i}, {x → -0.486389},
  {x → 0.945068 - 0.854518 i}, {x → 0.945068 + 0.854518 i}}

a = %[[1, 1, 2]]
-0.701874 - 0.879697 i

x^5 + 2x + 1/.x → a
2.22045 × 10^-16 + 2.22045 × 10^-16 i
```

(19)

今度は Solve で解を求める．代数的に表せないので複雑なシンボル (正しくはルートオブジェクトという) が出力されるが，これは解として厳密な性質を有する．

```
Solve[x^5 + 2x + 1 == 0, x]

{{x → Root[1 + 2 #1 + #1^5 &, 1]}, {x → Root[1 + 2 #1 + #1^5 &, 2]}, {x → Root[1 + 2 #1 + #1^5 &, 3]},
  {x → Root[1 + 2 #1 + #1^5 &, 4]}, {x → Root[1 + 2 #1 + #1^5 &, 5]}}

b = %[[1, 1, 2]]

Root[1 + 2 #1 + #1^5 &, 1]

x^5 + 2x + 1 /. x → b

1 + 2 Root[1 + 2 #1 + #1^5 &, 1] + Root[1 + 2 #1 + #1^5 &, 1]^5

Simplify[%]

0
```

(20)

このように，厳密値を扱うことができるのが Mathematica の強力な所であるが，近似値を扱う場合は細心の注意が必要である．