経営情報研究
第19巻第2号(2012年2月), 53－60ページ

研究論文

2^{64}を法とする逆数合同法乱数

牧　野　　純

Inversive Congruential Random Numbers with Modulus 2^{64}

Jun MAKINO

【要　約】Eichenauer らによって提案された 2 の冪を法とする逆数合同法乱数生成法は任意の語長に対して容易に実装が可能である．この論文では，最近の開発環境で一般的になってきた 64 ビット整数に対して，この生成法を実装し，その統計的な性質を検定し，生成速度を測定した．統計的検定の結果は，64 ビットの線形合同法と比較すると，多少良いと言えるが，誕生日間隔検定には不合格であった．また，生成速度は線形合同法よりかなり遅い．したがって，逆数合同法は 64 ビット整数の場合も良い乱数生成法とは言えない．

経営情報研究 第 19 巻第 2 号(2012 年 2 月), 53－60 ページ

1 はじめに

PC の環境は 32 ビットから 64 ビットへと移行しつつある. アーキテクチャが AMD の x86-64 を採用するようになって 10 年以上が経過し，オペレーティングシステムも 64 ビットが主流となりつつある.

開発環境においても 64 ビット整数は基本型として扱われるようになり, Java では long 型が 64 ビットの整数型と規定されている．C による開発環境として最も一般的と思われる gcc (GNU C Compiler)においても 64 ビット整数の型が利用できるようになっている．符号付整数型が long long 型，符号なし整数の型が unsigned long long 型である.

さて，擬似乱数生成法は簡単な演算をいくつか組み合わせて定義される漸化式を用いて，数列を生成するものである．それらは，演算に算術演算を用いる生成法と，論理演算を用いる生成法に分類される．この論文で考察するのは，算術演算を用いる乱数生成法である.

算術演算として用いられるのは乗算や加算であり，これらの演算，とくに乗算は，整数の上位ビットほどランダムに別の整数に変換する．したがって，語長が長いほど算術演算が有効に働くと考えられる.

算術型の乱数生成法として最もよく知られているものが線形合同法である．とくに，整数型のビット長を e ビットとするとき，2^e を法とする線形合同法

$$x_{n+1} = (a x_n + c) \bmod 2^e$$

は実装が非常に簡単で,

```
x = a * x + c;
```

とすればよい．算術演算における桁あふれは無視されるが，それがちょうど $\bmod 2^e$ の計算になっているからである.

線形合同法にはよく知られた欠点がある．生成する連続した k 個の乱数を k 次元空間の点と考えると，それらの全体は k 次元空間に格子状に並ぶのである．これは漸化式の線形性に基づく.

この線形合同法の欠点を克服すべく，Eichenauer は，逆数を用いた非線形な合同法を提案した[1]．逆数合同法は格子構造を持たないことで注目されたが，実際に統計的な検定を行ってみると，その性質は思わしくなかった.

L'Ecuyer 等は乱数の統計検定パッケージ TestU01 を開発し，文献[2]において主要な乱数生成法に対して検定を実行した結果を報告している．そこでは，素数 $2^{31}-1$ を法とする逆数乱数生成法について，Crush について 5 検定項目，BigCrush については 10 検定項目ほどについて不合格になるとの報告がある．しかし，これらの乱数の周期は $2^{31}-1$ と短く，より多くの乱数を用いて検定を行う Crush や BigCrush にこうした生成法が不合格となるのは必然的な結果と思われる.

そこで，この論文では，2^e を法とする逆数合同法を考察する．この乱数も Eichenauer ら[3]によって考案された（[4]の 3.2.2 の問題 36 (p.40, p.558)にも紹介されている）．
それは漸化式

$$x_{n+1} = (a x_n^{-1} + c) \bmod 2^e$$

に基づくもので，$x_0 = 1$, $e \geq 3$ ならば，$a \bmod 4 = 1$, $c \bmod 4 = 2$ のとき周期 2^{e-1} をもつ（つまり周期にすべての奇数が現れる）ことが証明されている．

L'Ecuyer 等は 2^e を法とする逆数合同法については検定をしていない．しかし，次節で見るように，法が $m = 2^{64}$ のときの逆数合同法を検定してみることは容易であり，しかもその周期は 2^{63} と長く，Crush や BigCrush にかけてみる価値がある．

2^e を法とする線形合同法は，$a \bmod 4 = 1$, $c \bmod 2 = 1$ のとき最大周期 2^e をもつ．しかし，その第 0 ビット（最下位ビット）は周期が 2，第 1 ビットは周期が 4，第 2 ビットは周期が 8 といった具合に，一般に第 k ビットは 2^{k+1} の周期をもつビット列となり，下位のビットはランダムとは言えない．

この事情は 2^e を法とする逆数合同法でも同じである．この場合は，乱数はすべて奇数なので，第 0 ビットは常に 1 であり，周期は 1 である．第 1 ビットは周期が 2，第 2 ビットは周期が 4 といった具合に，一般に第 k ビット位置は 2^k の周期をもつビット列となる．したがって，下位のビットはランダムとは言えないので注意が必要である．統計的検定には 64 ビットのうち上位の 32 ビットを用いた．

2　2^{64} を法とする逆数合同法の実装

2.1　逆数の求め方

x を奇数とするとき

$$xy \equiv 1 \pmod{2^e}$$

を満たす y が存在し，これを 2^e を法とする x の逆数と呼ぶ．

2^e を法とする奇数の逆数を計算するアルゴリズムは容易に書き下すことができる．実際，Knuth の教科書では，これは練習問題に含まれている（文献[4]の 4.5.2 の問題 17 (p354, p642)参照）．

x を奇数として，その逆数の下位 k ビットが既知とする．すなわち

$$xy_k \equiv 1 \pmod{2^k}$$

なる y_k が求められたとする．このとき

$$xy_{2k} \equiv 1 \pmod{2^{2k}}$$

となる y_{2k}（すなわち，逆数の下位 $2k$ ビット）はつぎの式で計算できる．

$$y_{2k} \equiv y_k(2 - xy_k) \pmod{2^{2k}} \qquad (1)$$

$y_1 \bmod 2 = 1$（奇数の逆数は奇数）であるから，y_{64} を求めるためには式(1)を繰り返し適用して $y_2, y_4, y_8, y_{16}, y_{32}, y_{64}$ の順に計算していけばよい．その際，式(1)は両辺の下位 $2k$ ビットが一致することを表しており，それ以上のビットについては何の条件も課されないことに注意する．したがって，アルゴリズムの実装の際には，これら上位ビットは計算に都合が良いように扱うことができる．

2.2 逆数合同法の実装

図1は上記の方法で逆数を計算し，乱数を生成する逆数合同法のC言語のプログラムである．

逆数合同法を利用して多くの乱数を生成する場合，逆数を求める計算をなるべく高速に行う必要がある．そのため，`init_inv` 関数では，16ビット整数の逆数はあらかじめすべて計算して配列に保存しておく．この関数では，式(1)を用いて，$y_1 = 1$ から始めて，順に y_2, y_4, y_8, y_{16} までの計算を，すべての16ビット奇数について計算しておくのである．16ビット奇数は32768個しかないので，乱数生成に先立って一度だけこれらすべての逆数を求めておくことはたいしたオーバーヘッドとはならない．また，それらすべてを記憶しておくために必要なメモリーは64KBで，これも多くの場合は問題にならない．図1のプログラムでは配列 `inv16` への参照を高速に行うため，配列の大きさを倍の65536（128KB）としている．

この16ビット整数の逆数 y_{16} の表を用いてさらに，y_{32} そして y_{64} を求める計算は x を引数とする `inv` 関数で行っている．

もし，`init_inv` 関数で y_{32} までの計算をすべての32ビット奇数について行って，それらを表にしておけば，`inv` 関数では y_{64} を求める計算が残るだけとなり，逆数の計算はさらに高速化できる．しかし，32ビット奇数は2147483648個もあり，これらのすべての逆数を計算しておくためにはかなりの時間がかかる．また，それらの結果を保存しておくのに必要なメモリは8GBにもなる．したがって，32ビットの逆数の表を使うことは非現実的である．

`rng` が逆数合同法乱数を求める関数である．この関数は乱数を1個生成するために `inv` 関数

を 1 回呼び出している．乱数 x の初期値は 1 としてあるが，これは任意の奇数でかまわない．また，乗数と加数も $a \bmod 4 = 1$, $c \bmod 4 = 2$ であれば，適当に選んでよい．もちろん，あまりに規則的なビットパターンを含むような定数は避けるべきである．

```
/* 逆数を計算する関数 */

unsigned short inv16[65536];

void init_inv(void)
{
   unsigned short x, y;
   inv16[1] = 1;
   for (x = 3; x > 1; x += 2) {
      y = 1;
      y = y * (2 - x * y);
      y = y * (2 - x * y);
      y = y * (2 - x * y);
      y = y * (2 - x * y);
      inv16[x] = y;
   }
}

unsigned long long inv(unsigned long long x)
{
   unsigned long long y;
   y = inv16[x & 0xFFFF];
   y = y * (2 - x * y);
   y = y * (2 - x * y);
   return y;
}

/* 逆数合同法乱数 */

unsigned long long x = 1;

unsigned long long rng(void)
{
   return x = 3141592653589793237ULL * inv(x) + 8462643383279502ULL;
}

/* 乱数を使うプログラム例 */

#include <stdio.h>

int main(void)
{
   int i;
   init_inv();
   for (i = 0; i < 10; i++)
      printf("%20llu\n", rng());
   return 0;
}
```

図 1: 2^{64} を法とする逆数合同法

`main` 関数は，これらの関数の利用例を示す簡単なプログラムである．この関数は，10 個の乱数を出力する．`rng` 関数を用いて乱数を生成する前に，一度 `init_inv` 関数を呼び出して逆数の計算の準備をすることを忘れてはならない．このプログラムの出力は図 2 のようになる．

```
3150055296973072739
2085855973891612917
12860188991264233711
4358703551711658249
2102428100400656635
5201709297190188797
13458565154305306951
13463957281803991441
12729180233097374099
4955521971248799621
```

図 2: プログラム例の出力

3 統計的検定

乱数がランダムかどうかを調べるためには, 経験的検定を行う. 実際に乱数を発生してみて，さまざまな統計量を計算し，それらが真の乱数の分布と矛盾しないかどうかを統計的に検定するのである．このような目的で，多くの検定項目が考案されてきた．数列が真にランダムとみなせるためには，どのような見方をしても，それが真の乱数と同じ統計的性質を持たなければならないからである．

今日では, こうした検定項目をいくつかセットにした乱数検定パッケージが提供されている．こうしたパッケージは，パッケージ内のすべての項目の検定に合格する乱数は，パッケージに含まれない他の多くの検定項目についても合格することを目標に設計されている．

このような検定パッケージのうち，最近よく使われるのが L'Ecuyer らによる TestU01 である[2]．TestU01 には SmallCrush, Crush, BigCrush と呼ばれる 3 種類のセットが用意されていて，この順に簡便な検定から本格的な検定までが可能となっている．SmallCrush は約 2^{28} 個の乱数を用いて 10 の項目を検定し，Crush は約 2^{35} 個の乱数を用いて 96 の検定項目を，BigCrush は 2^{38} 個の乱数を用いて 106 の検定項目を計算するという徹底した検定セットで, そのすべての検定に合格する乱数発生法はごく限られている．BigCrush に合格することは新しい乱数発生法を開発するときの目標となっている．

今回は, 図 1 の逆数合同法乱数を TestU01 の SmallCrush, Crush, BigCrush にかけてみた．また，比較のため，同じ 2^{64} を法とする線形合同法についてもこれらの検定を実行した．

その結果，逆数合同法は SmallCrush のすべての検定に合格したが，Crush では検定番号 17 の誕生日間隔検定に，BigCrush では検定番号 19 と 21 の誕生日間隔検定に不合格であった．

一方，64bit 線形合同法も，SmallCrush には合格した（乗数によっては不合格）ものの，

Crush では 3 つ（検定番号 15, 16, 17）の，BigCrush では少なくとも 5 つ（検定番号 15, 17, 18, 19, 21）の誕生日間隔検定に不合格となった．

2 つの乱数発生法について，不合格となった検定項目の数を表 1 の SmallCrush, Crush, BigCrush の列にまとめた．

表 1: 検定不合格数と生成時間

乱数生成法	SmallCrush	Crush	BigCrush	生成時間
逆数非線形合同法	0	1	2	17.71 nsec
線形合同法	0	3	5	2.99 nsec

4 生成速度

乱数は大規模なシミュレーションに用いられるため，その生成速度が速いことが良い乱数生成法の条件の一つになっている．逆数合同法は，線形合同法の演算（乗算と加算 1 回ずつ）に加えて逆数の計算を余分に含む．そのため，逆数合同法の生成速度は線形合同法と比較して遅くなることは自明である．そこで，どの程度遅くなるかを見るために，定量的な測定をしてみた．

1 つの乱数を生成する時間は短かすぎて測定できないので，乱数生成の関数を `for` ループの中で 10^9 回呼び出す時間を測定し，乱数 1 個あたりの時間に換算した．実行環境は intel Core 2 Quad プロセッサ Q9450 (2.66GHz)を搭載した 64bit Linux PC である．プログラムは `gcc` に `-O2` オプションを用いて最適化した．

こうして得られた乱数 1 個あたりの生成時間について，2^{64} を法とする線形合同法と逆数非線形合同法とを比較したのが，表 1 の一番右の列である．逆数合同法による乱数生成には線形合同法の約 6 倍の時間がかかっていることがわかる．

5 おわりに

2^{64} を法とする逆数合同法を 64 ビット環境の PC に実装して，TestU01 による統計的検定にかけると同時に，乱数生成の時間を測定した．

64 ビットの逆数の計算は，32 ビット逆数をさらに 2 倍のビット長に伸長する．そのため，64 ビット逆数合同法は 32 ビット合同法より生成速度が遅くなるが，プログラムの実装自体には何の問題もない．

TestU01 を用いた統計的検定では，SmallCrush のすべての検定項目について合格するものの，Crush と BigCrush ではいくつかの誕生日間隔検定に不合格となった．これは同じ 2^{64} を法とする線形合同法と同じである．不合格になった検定項目の数は逆数合同法の方が少ないの

で，線形合同法と比較すると性質が多少良いとも言えるが，不合格の項目がある以上，逆数合同法の統計的性質が良いとは結論できない．

一方，乱数生成にかかる時間は逆数合同法は線形合同法の約 6 倍であった．逆数の計算に 4 回の乗算を含む多くの演算を必要としているためである．

線形合同法は定数の乗算，定数の加算という算術演算を用いて乱数列を生成する．一般に算術演算は語長が長いほど良い乱数を生成できると考えられる．逆数をとる演算も算術演算と考えられるので，逆数合同法にとっても語長は長い方が有利であろう．これらの乱数生成法が開発されたときは，コンピュータの語長は現在よりも短かった．そこで，現在の 64 ビット環境における逆数合同法について調べてみたわけだが，残念ながら，逆数合同法は 64 ビット整数の場合も良い乱数生成法とは言えないことがわかった．

参考文献

[1] Eichenauer, J. Lehn, J. 1986. A nonlinear congruential pseudorandom number generator, *Statistische Hefte 27* 315-326.

[2] L'Ecuyer, P. and Simard, R. 2007. TesuU01: A C Libray for Empirical Testing of RNGs, *ACM Transactions on Mathematical Software.* 33, Article 22.
（ソフトウェアは `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html` よりダウンロードできる．）

[3] Eichenauer, J. Lehn, J. and Topuzoğlu, A. 1988. A nonlinear congruential pseudorandom number generator with power of two modulus, *Math. Comp.* **51** 757-759.

[4] Knuth, D. E. 1998. *The Art of Computer Programming.* Vol. 2: Seminumerical Algorithms, 3rd ed. Addison-Wesley, Reading, MA.