代数と乱数

松本　眞（広島大学理学研究科数学専攻）

2014/4/25, 先端数学@広島大学数学科

m-mat@math.sci.hiroshima-u.ac.jp

1. 乱数：なんに使うの？

- **モンテカルロ法:**

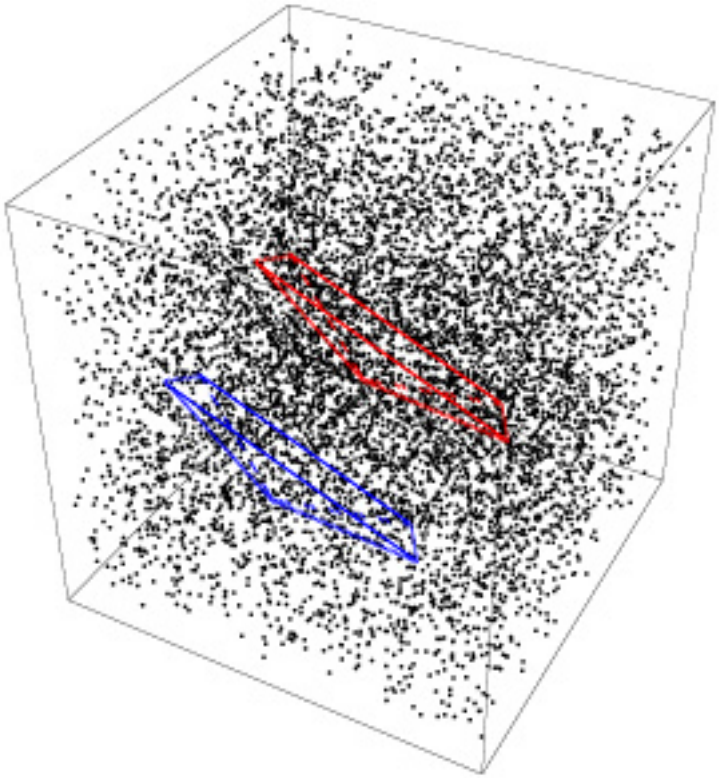
  例：ある領域の体積の近似

  赤体積$\simeq \frac{61}{7248} = 8.4 \times 10^{-3}$.

  青体積$\simeq \frac{64}{7248} = 8.8 \times 10^{-3}$.

  (真の体積: $8.333\cdots \times 10^{-3}$)

  右図は3次元中の多面体だが
  もっと高次元・複雑な図形に
  モンテカルロ法が有効



3次元単位立方体に、でたらめにN個点を打つには：

1. 区間$[0,1] := \{x \mid 0 \leq x \leq 1\}$内から、
   一様にでたらめに実数xを取ってくる
2. 同様に、区間$[0,1]$から
   一様に独立にでたらめに実数y, zを取ってくる
3. 座標(x, y, z)の点を立方体に打つ
4. 1-3をN回繰り返す

定義(ややあいまい)

- $[0,1]$区間の中から、
  「一様かつ独立に次々とでたらめな数を発生させる方法」
  を乱数発生法という。
- 得られた数列を乱数列と呼ぶ。

注：
$[0,1]$区間に限らず、サイコロのように
「$\{1,2,3,4,5,6\}$ の中からでたらめな数を発生させる方法」
も乱数発生法という。

- モンテカルロ法（つづき）

  確率的現象を含んだあらゆる現象のシミュレーションに
  乱数は必要である。これらもモンテカルロ法と呼ばれる。
  （モンテカルロ市はモナコの首都、カジノのメッカ）

  物理、化学：核反応シミュレーション

  生物科学：たんぱく質折りたたみシミュレーション

  金融工学：株価変動

  遊び、芸術：ゲーム、漫画の模様（トーン）

- 暗号への応用 (略)
- 確率的アルゴリズム (略)

2. どうやって生成するか

…それが問題だ。
“決定的な動作しかしない計算機では、生成できない”

物理乱数発生器:

- 物理雑音から拾ってくる: もっとも素朴な方法
- 例：サイコロ、熱雑音、株価の変動
- 問題点: コスト、スピード、再現不能性.

再現不能性とは、同じ乱数列を再現するのに、
それらをすべて記録しておかないとならないこと。

再現性が必要となる場合：

- 追試
- 最適化

参考：核シミュレーションでは乱数は何兆個も消費される
⇒ それらをすべて記録しておくのは効率が悪い。

擬似乱数発生法:
漸化式を用いて、乱数のように見える数列を生成する方法

例: 線形合同法 Linear Congruential Generator
(LCG, Lehmer '60)

- ある整数x_1を初期シードとして選ぶ。
- 次の漸化式により$x_2, x_3, \ldots$を次々に生成する：

$$x_{n+1} = 1103515245 x_n + 12345 \bmod 2^{32}.$$

例 $x_1 = 3$ならば

$$\begin{aligned}
3 \times 1103515245 + 12345 &= 3310558080 \ (\mathrm{mod} 2^{32}) \to 3310558080 = x_2 \\
3310558080 \times 1103515245 + 12345 &= 3653251310737941945 \ (\mathrm{mod} 2^{32}) \to 465823161 = x_3 \\
465823161 \times 1103515245 + 12345 &= 514042959637601790 \ (\mathrm{mod} 2^{32}) \to 679304702 = x_4 \\
679304702 \times 1103515245 + 12345 &= 749623094657194335 \ (\mathrm{mod} 2^{32}) \to 2692258143 = x_5
\end{aligned}$$

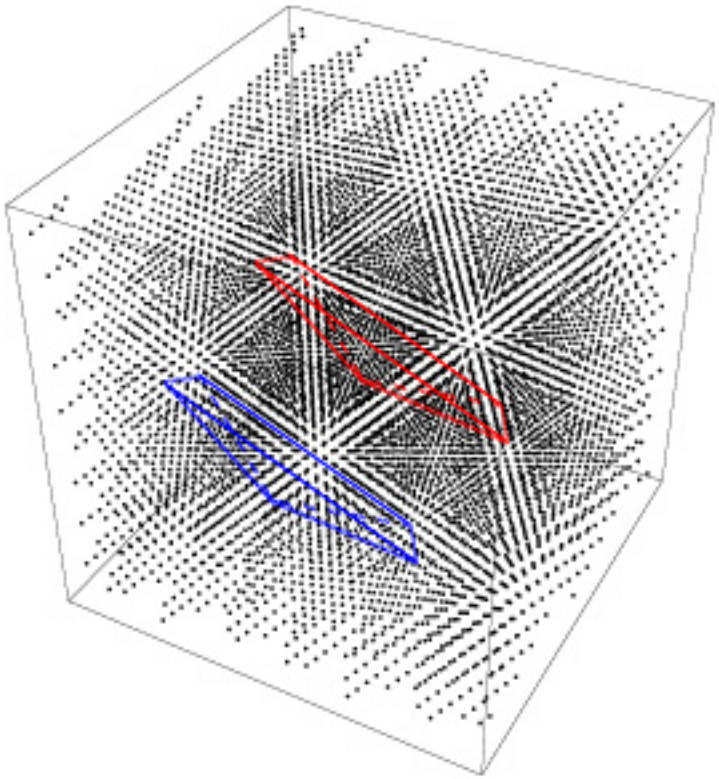
擬似乱数のメリット:

- 漸化式と初期シードを記録しておけば、
  誰でも同じ数列を高速に再現できる
- 高速で低コスト

問題点：「乱数と呼んでいいのか」
（von Neumann 「漸化式で乱数を作るのはある種の罪」）

たとえば:

- 先の線形合同法による数列の周期は、
  初期シードの選び方によらず2^{32}。
- この生成法は、70年代から80年代にかけて
  ANSI-Cなどの標準擬似乱数であった。
- 現代のパソコンは数分で2^{32}個の乱数を使ってしまう
- 生成される数列はかなり乱数に見えるが、
  数千万個の出力を使うと、非乱数性が現れてくる

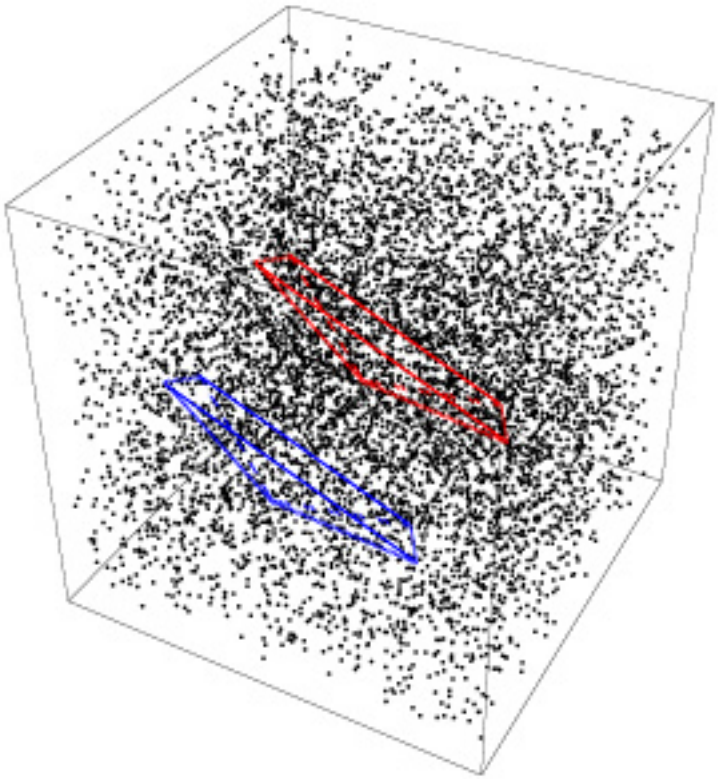


この線形合同法による
3次元ランダム点プロット

赤領域体積$\simeq \frac{62}{7253} = 8.5\times10^{-3}$.

青領域体積$\simeq \frac{45}{7253} = 6.2\times10^{-3}$.

(真の値: $8.333\cdots\times10^{-3}$)



Mersenne Twister(松本-西村 '98)
による3次元ランダム点プロット

赤領域 $\simeq \frac{61}{7248} = 8.4 \times 10^{-3}$.

青領域 $\simeq \frac{64}{7248} = 8.8 \times 10^{-3}$.

(真の値: $8.333\cdots \times 10^{-3}$)

3. 擬似乱数への要請

- 高速性

  素粒子シミュレーションなどでは、全体の計算時間の
  ４０％を擬似乱数生成が占めることもある

- 乱数性

  擬似乱数は、所詮漸化式による数列
  「乱数性」の十分に実用的な（実証可能な）定義がない

  – 周期, 分布

    乱数性の定義がないので、代わりに周期や高次元分布
    に着目して良いものを使う

線形合同法の問題点：

$$x_{n+1} = ax_n + c \bmod M$$

なる数列の周期は高々M
（$\because x_n$が取りうる値は$0, 1, \ldots, M-1$のM個しかない）

メリット：周期や分布を求めるアルゴリズムがある
限界：周期を大きくするにはMを大きくするしかなく、
掛け算や割り算が必要で生成が遅くなる

4. 循環小数と線形合同法　$c = 0$のとき、線形合同法

$$x_{n+1} = ax_n \bmod M$$

は、「$x_1 \div M$のa進小数展開に現れる余りの列」である。

例$x_1 = 1, 1 \div 7$の10進小数展開

$1 \times 10 = 10, \div 7 = 1$ 余り $3 =: x_2$

$3 \times 10 = 30, \div 7 = 4$ 余り $2 =: x_3$

$2 \times 10 = 20, \div 7 = 2$ 余り $6 =: x_4$

$6 \times 10 = 60, \div 7 = 8$ 余り $4 =: x_5$

$4 \times 10 = 40, \div 7 = 5$ 余り $5 =: x_6$

$5 \times 10 = 50, \div 7 = 7$ 余り $1 =: x_7$

$0.\dot{1}4285\dot{7}1$

$7\overline{)10}$

7

30

28

20

14

60

56

40

35

50

49

10

7

30

5. 整数から$\mathbb{F}_2$多項式へ：$\mathbb{F}_2$多項式での循環小数を利用

二元体$\mathbb{F}_2$

$\mathbb{F}_2 := \{0, 1\}$とおく。

0,1の掛け算は普通に定義して、$\mathbb{F}_2$からはみ出ない。
足し算は$1 + 1 = 2$だけが$\mathbb{F}_2$からはみ出してしまうので、

$$1 + 1 = 0$$

と定義する（2で割ったあまりを見ている）。

1+1=0から1を移項して

$$1 = -1.$$

$\mathbb{F}_2$多項式$\mathbb{F}_2[t]$

$$\mathbb{F}_2[t] := \{\sum_{i=0}^{n} a_i t^i | a_i \in \mathbb{F}_2,\ n \in \mathbb{N}\}$$

を考える。掛け算・足し算は通常の多項式同様

$$\begin{aligned}(t+1) \times (t+1) &= t(t+1) + 1(t+1) \\ &= t^2 + t + t + 1 \ = \ t^2 + 1\end{aligned}$$

といった具合に計算できる。

（$1+1=0$より$t+t=(1+1)t=0$。）

係数のみを表記することにして、

$$t^3 + t^2 + 1 = 1t^3 + 1t^2 + 0t + 1 = 1101$$

と表わすことにする。

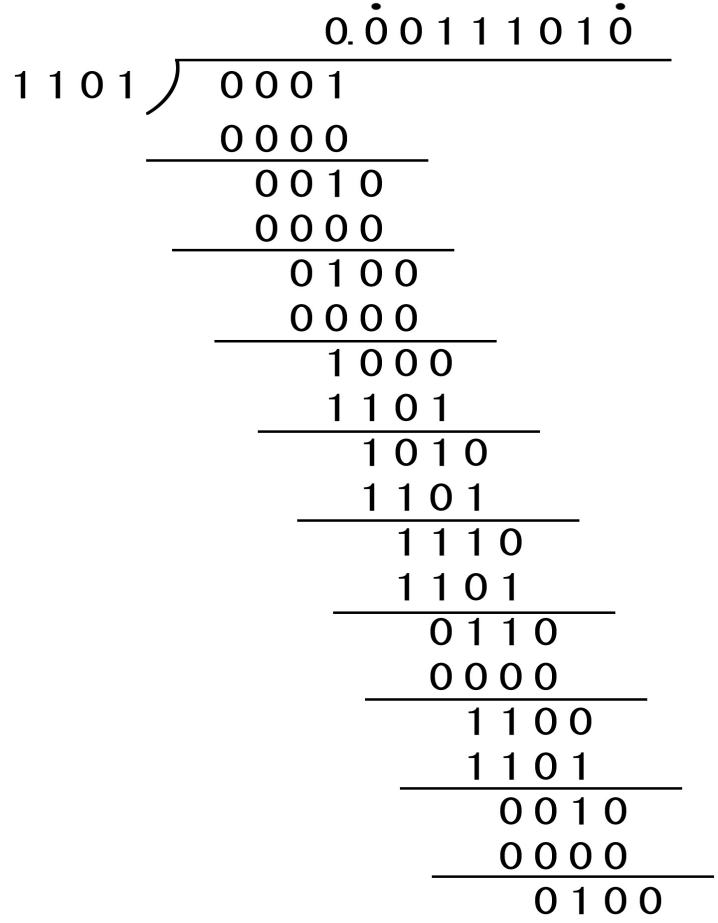
$\mathbb{F}_2$多項式での和差積商は、

「繰り上がり・繰り下がりのない世界での計算」

になる。

$$\begin{array}{r} 1\ 1 \\ \times\ 1\ 1 \\ \hline 1\ 1 \\ 1\ 1\ \ \\ \hline 1\ 0\ 1 \end{array} \qquad \begin{array}{r} t\ \ +1 \\ \times\ \ t\ \ +1 \\ \hline t\ \ +1 \\ t^2\ +t\ \ \\ \hline t^2\ +0t\ +1 \end{array}$$

$\mathbb{F}_2[t]$ の世界で $1 \div (t^3 + t^2 + 1) = 1 \div 1101$ を小数展開すると



検算：

$$
\begin{array}{r}
0.00111010011101\cdots \\
\times\ 1101 \\
\hline
0.00111010011101\cdots \\
00.00000000000000\cdots \\
000.11101001110100\cdots \\
0001.11010011101001\cdots \\
\hline
0001.00000000000000\cdots
\end{array}
$$

$1 \div (t^3 + t^2 + 1) =$

$0 + 0t^{-1} + 0t^{-2} + 1t^{-3} + 1t^{-4} + 1t^{-5} + 0t^{-6} + 1t^{-7} \cdots$

$1 \div 1101 = 0.x_1x_2x_3x_4\cdots$ とおくと $\mathbb{F}_2$ での漸化式

$$x_{n+3} + x_{n+2} + x_n = 0 \quad (n \geq 1) \quad \text{を満たす}$$

検算：

$$\begin{array}{rrrrcccccc}
 & & & 0 & . & x_1 & x_2 & x_3 & x_4 & x_5 & \cdots \\
\times & 1 & 1 & 0 & 1 & & & & & & \\
\hline
 & & & 0 & . & x_1 & x_2 & x_3 & x_4 & x_5 & \cdots \\
 & & 0 & 0 & . & 0 & 0 & 0 & 0 & 0 & \cdots \\
 & 0 & x_1 & x_2 & . & x_3 & x_4 & x_5 & x_6 & x_7 & \cdots \\
0 & x_1 & x_2 & x_3 & . & x_4 & x_5 & x_6 & x_7 & x_8 & \cdots \\
\hline
0 & 0 & 0 & 1 & . & 0 & 0 & 0 & 0 & 0 & \cdots
\end{array}$$

同じ理屈で、$1 \div (t^{607} + t^{273} + 1)$ を小数展開したものを
$0.x_1x_2x_3x_4\cdots$ とおくと漸化式

$$x_{n+607} = x_{n+273} + x_n$$

を満たす。逆に、この式を使って x_n が求められる。

（実は、この漸化式による 0-1 列の周期は $2^{607} - 1$ となる。）

このような$\mathbb{F}_2$上の高階線形漸化式を用いて0-1列を生成、
擬似乱数として用いる方法を
　Tausworthe法あるいは
　Linear Feedbacked Shift Register法（LFSR法）
と言う (Tausworthe, 1965)。

メリット：

- 周期を長くしても、生成速度が遅くならない

$$x_{n+607} = x_{n+273} + x_n$$

- 周期が極大なので、「全て0」以外のビットパターンが

$$(x_n, \ldots, x_{n+607-1}) \quad (n = 1, 2, \ldots, 2^{607} - 1)$$

　のなかにちょうど一度だけ現れる（607次元均等分布性）

6. ベクトル化：$\mathbb{F}_2$多項式から$\mathbb{F}_2$線形変換へ

GFSR (Lewis-Payne '73) 計算機ワード長の$\mathbb{F}_2$ベクトル列を

$$\vec{x}_{n+p} := \vec{x}_{n+q} + \vec{x}_n$$

で生成(+は$\mathbb{F}_2$ベクトルとしての和)

例：4ビット計算機なら$1101 = 0100 + 1001$

整定数p, qをうまく選ぶと周期$2^p - 1$にできる

- 各桁はTausworthe法で生成される数列に一致
- 高速だが、各桁の間に情報のやり取りがない
- 乱数性に問題あり（特にランダムウォークで）

より一般の$\mathbb{F}_2$線形変換

Twisted GFSR (松本-栗田良春 '92, '94):
Twisterと呼ぶ$\mathbb{F}_2$係数正方行列Aを導入する：

$$\vec{x}_{n+p} = \vec{x}_{n+q} + \vec{x}_n A.$$

- A は桁の間の情報を混ぜる

  $\Rightarrow$ より長周期: $2^{32p} - 1$で良い乱数性を持つ(32:ワード長)

- Aは次のようなものを選ぶ:定数ベクトル$\vec{a}$により

$$\vec{x}A = \begin{cases} \text{shiftright}(\vec{x}) & (\vec{x}\text{の最下位ビットが}0\text{の場合}) \\ \text{shiftright}(\vec{x}) + \vec{a} & (\vec{x}\text{の最下位ビットが}1\text{の場合}) \end{cases}$$

  $\Rightarrow$高速に計算可能

- 高次元均等分布性を改善するため$\vec{x}_n T$を出力('94)

Mersenne Twister: (松本-西村拓士 '98):
TGFSRの周期を大きな素数$2^{32p-r}-1$の形にするため、

$$\vec{x}_{n+p} = \vec{x}_{n+q} + \vec{x}_{n+1}B + \vec{x}_n C$$

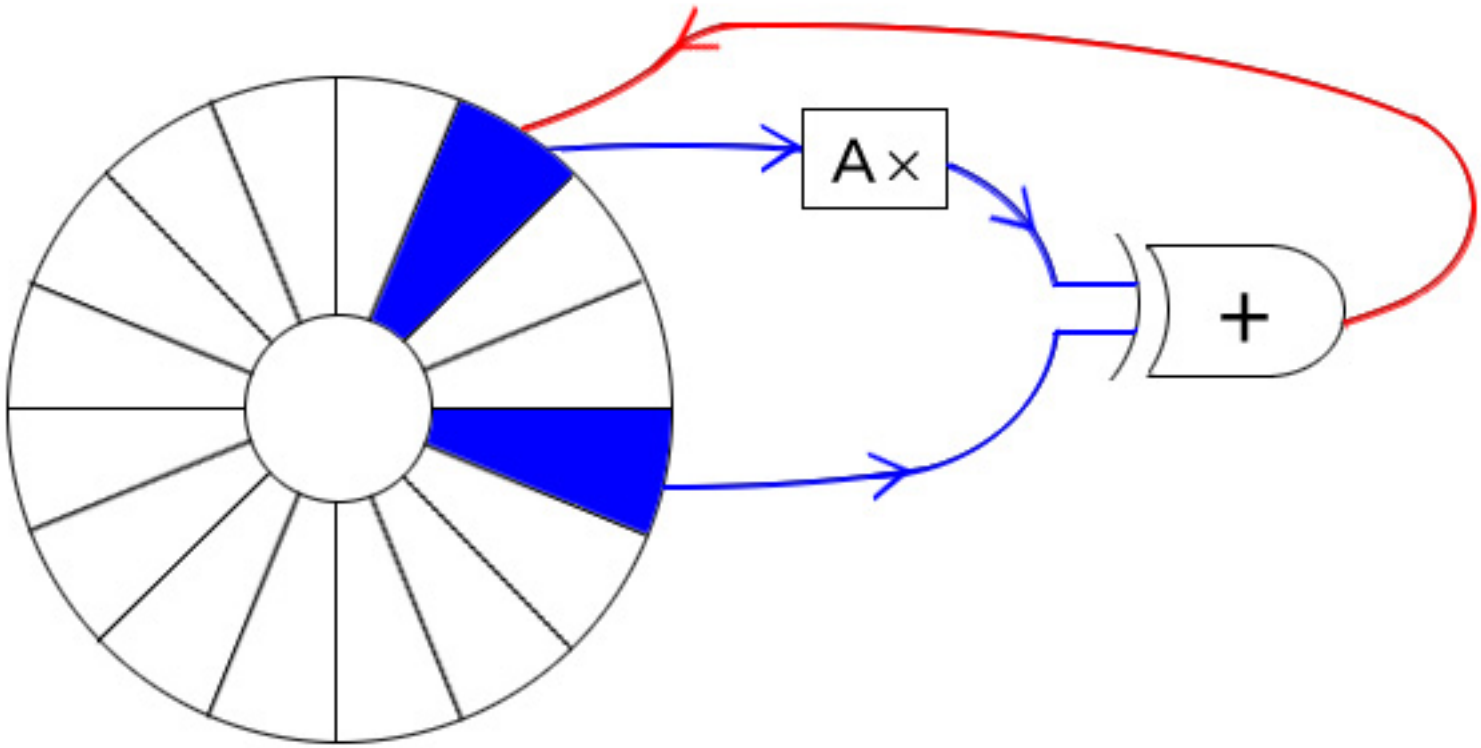
と行列を二つ導入し、漸化式にr次元のKernelを持たせる
$\Rightarrow$ 素数周期の判定は容易なため、
超長周期が実現可能: $2^{624\times 32-31}-1 = 2^{19937}-1$.

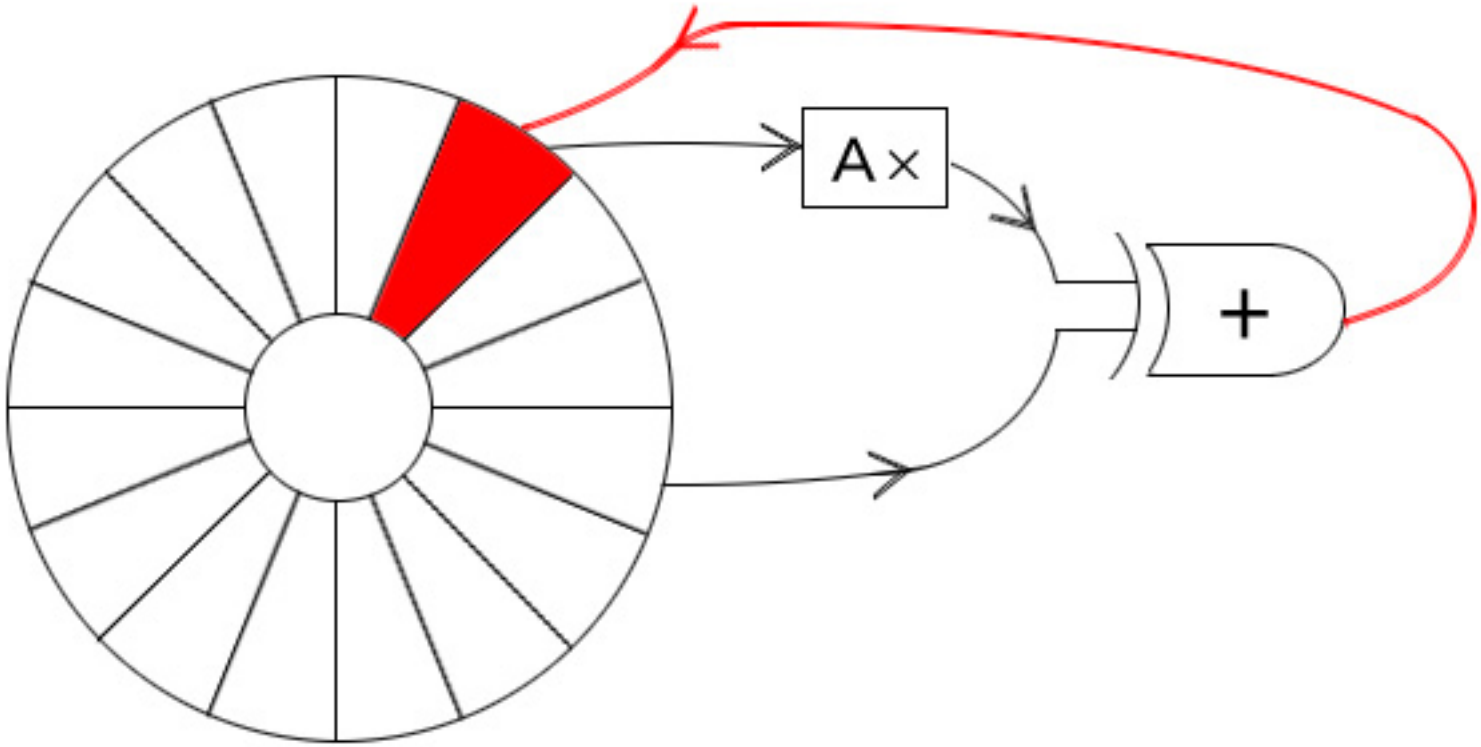
注：2^N-1の形の素数をメルセンヌ素数という

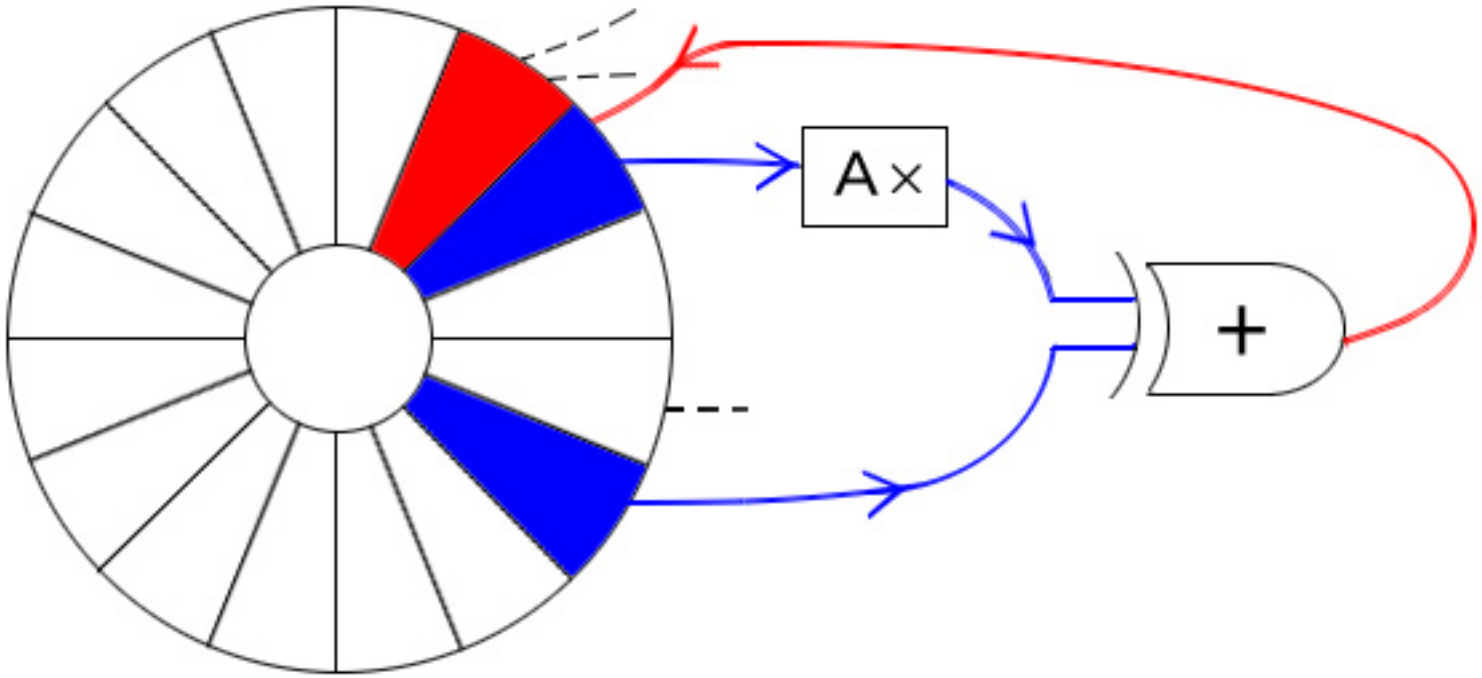
7. Mersenne Twisterの性能

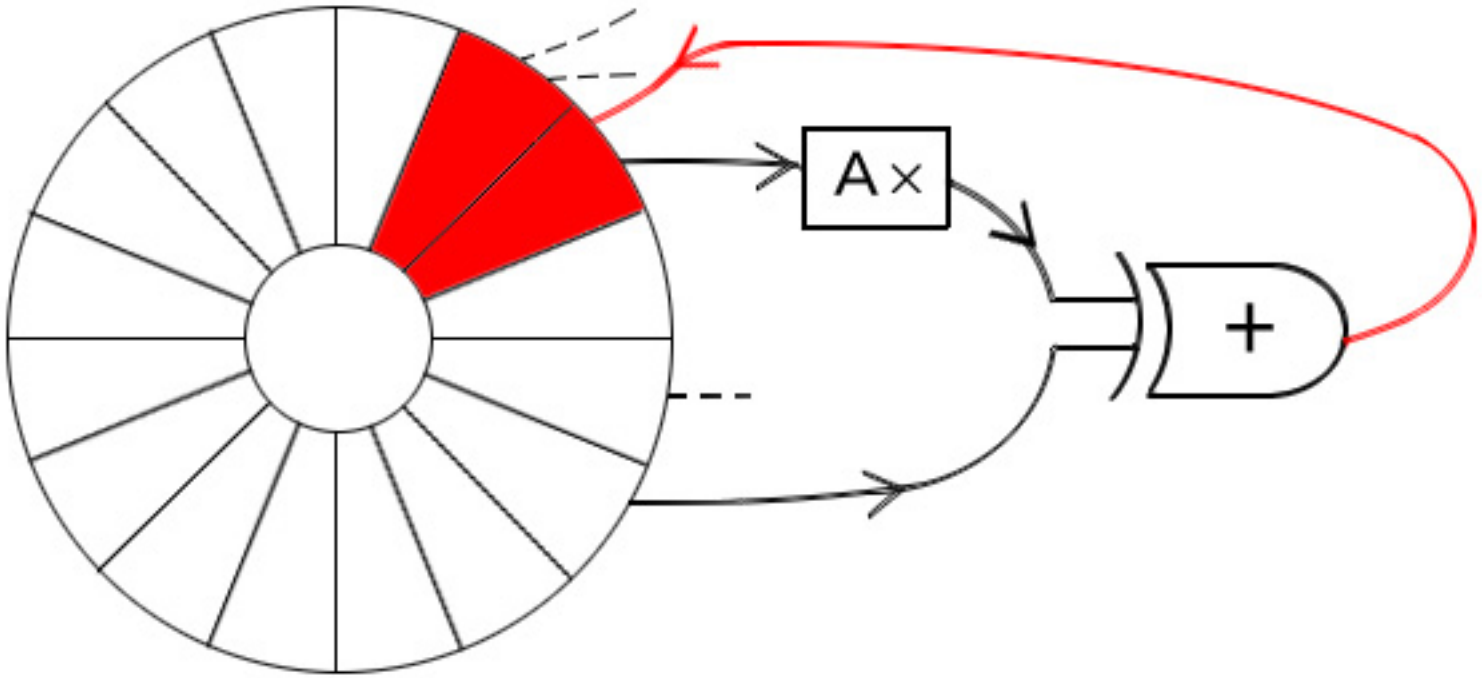
- 周期：$2^{19937}-1$
- 32ビット整数列として、623-次元均等分布している
- 高位ビットはより高次元均等分布している
- 生成速度は、近年の線形合同法よりも高速

なぜMT(TGFSR)は高速か？⇐ラウンドロビン実装




A×
+


A×
+


A×
+

8 高次元均等分布性

定義（あらまし）
擬似乱数列がk次元均等分布しているとは、
出力列をk個ずつに区切ってk次元立方体内の点を
一周期に渡り発生したとき、均等に分布していること

定義（正確）
有限集合Xに値をとる擬似乱数列が
「k次元均等分布している」とは、
「k個ずつ組にしてX^k内の点を一周期に渡り発生すると
X^k内のどの元も同数回ずつ現れる」こと

例：周期８の１ビット(0,1)列

$$000101110001011\cdots$$

連続3組を一周期に渡り並べると

$$000, 001, 010, 101, 011, 111, 110, 100,$$

となり、可能な８パターンを全て一度ずつとる
$\Rightarrow$ 3次元均等分布している

例：周期16の2ビット(0,1,2,3)列

$$0010203112132233001020311213\cdots$$

$$00, 01, 10, 02, 20, 03, 31, 11, 12, 21, 13, 32, 22, 23, 33, 30$$

$\Rightarrow$ 2次元均等分布している

高次元均等分布性の意義:

経験的には大抵：

均等分布の次元　＞
シミュレーションに現れる多変数関数の変数の個数の最大値

⇓

シミュレーションの結果は正しい.

MTの場合は

- 32ビット整数列として623次元均等分布
- 高位ビットはさらに高次元
  （例えば上位3ビットは6240次元均等分布）

注:

- MT以前はせいぜい20次元均等分布程度
- 実用上は、そこまで高次元均等分布していなくても
  大抵大丈夫

9 MTに用いられている数学

- 周期の計算：

  ベクトル値線形漸化式の特性多項式$\chi(t)$を求めれば、

  周期＝$\chi(t)$を法としてのtの乗法位数

  - 行列式$\chi(t) = \det(tI_n - A)$の展開

    $\Rightarrow$ 行列サイズが19937×19937だと遅すぎる

    $\Rightarrow$ Berlekamp-Massey法('68)という

    数列から特性多項式を求めるアルゴリズムの利用

    （$\mathbb{F}_2[t]$の完備化である$\mathbb{F}_2((t^{-1}))$における互除法）
  - Frobenius作用を用いる周期の計算法

- 高次元均等分布性の計算：

  Lenstraのアルゴリズム ('85) Couture-L'Ecuyer-Tezuka ('93)

  （$\mathbb{F}_2((t^{-1}))$ における格子の幾何）

- 0-1のバランスの計算：

  符号理論における MacWilliams恒等式 ('77)

  （離散フーリエ反転により、擬似乱数列の出力の分布が

  数列の満たす関係式の分布により求まる）

10 まとめ：数学の予期せぬ効用

- $1+1=0$ の数学の研究は、ガロア $(1830$ ごろ$)$ に遡る
- 当時は応用の見えなかった純粋数学が、
  現在実用されている。
- $\mathbb{F}_2[t]$ と整数は良く似ており、
  前者が扱いやすい（現代整数論の指導原理の一つ）

11 現在の研究

- SIMD-oriented Fast Mersenne Twister (SFMT, 斎藤睦夫と）
  最近のCPUは128ビット演算命令
  (Single Instruction Multiple Data, SIMD)
  を持っている。

  SIMD命令やパイプライン処理を最大限に生かした
  漸化式を考案。

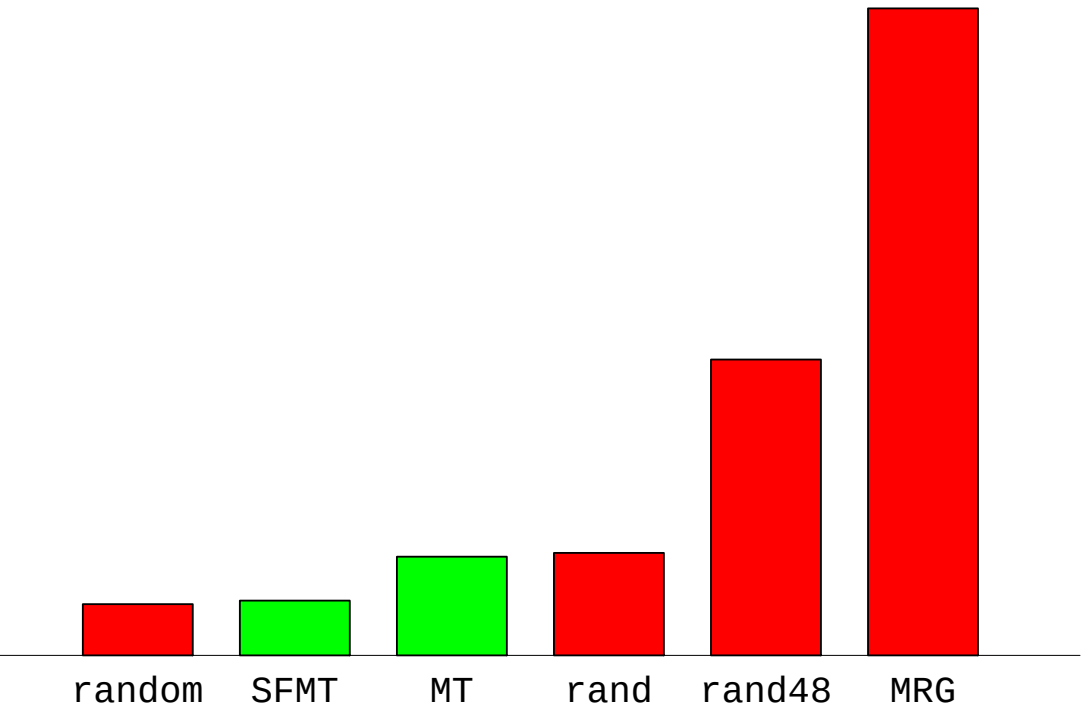
  SFMTは、SIMDを使わなくてもMTよりも高速。
  SIMDを使うと4倍程度高速。

  周期はMTと同じ、高次元均等分布性はMTより
  改善されている。
  （現在ホームページで公開中）

- WELL (with Panneton, L'Ecuyer '06)
  高次元均等分布性について、理論的上限を達成し、MTに近い高速性を持つ
- Pulmonary Mersenne Twister （肺つきMT）
  MTより高速で、悪い初期化からの立ち直りが早い
  (斎藤睦夫、原本博史、Francois Panneton, 西村拓士と)
- CryptMT: 暗号用MT, MTの出力を32ビット整数と思い、奇数化してつぎつぎ積算し、最上位8ビットだけ使う
  (斎藤睦夫、西村拓士、萩田真理子と)

速度比較

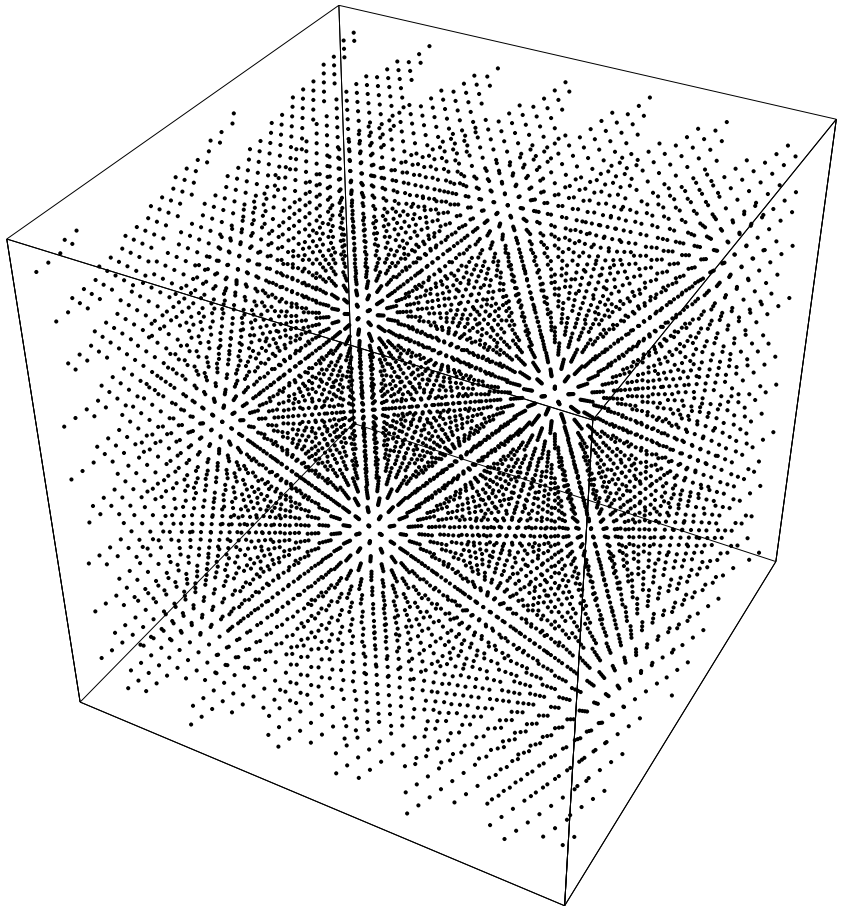
le



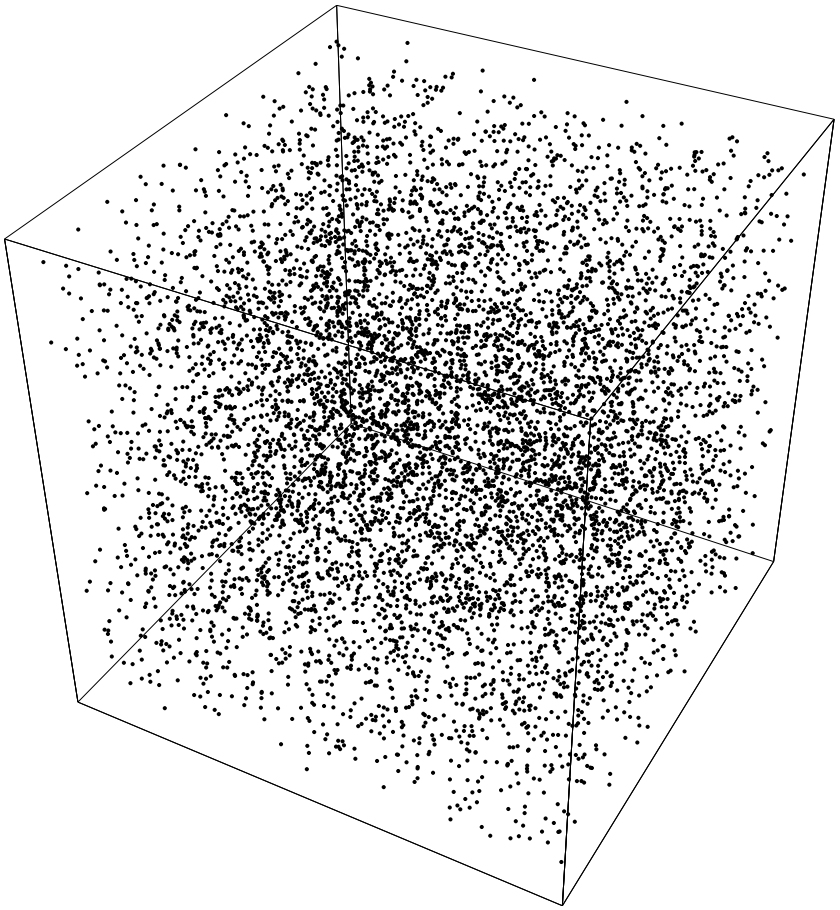
random: ラグ付き
フィボナッチ周期 $\sim 2^{63}$
rand: LCG 周期 2^{32}
SFMT SIMD Fast MT
MT: Mersenne Twister
　周期 $2^{19937} - 1$
rand48: LCG 周期 2^{48}
MRG: L'Ecuyer
　周期 $\sim 2^{186}$

12 終わりに：注意喚起

現在も、品質の悪い擬似乱数が広く使われている



ANSI-C標準擬似乱数('70-'90)



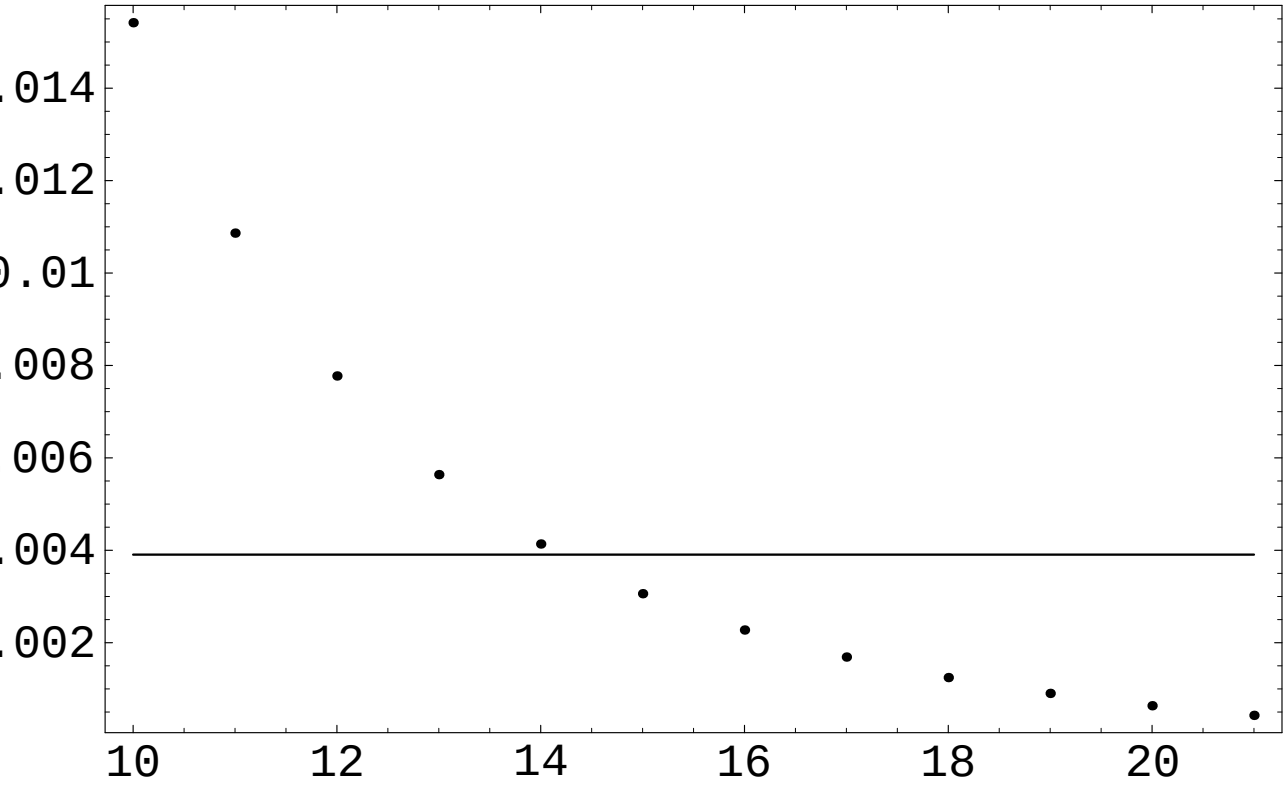
mt19937('98)

新しく提唱されたもの中にも悪いものが多い

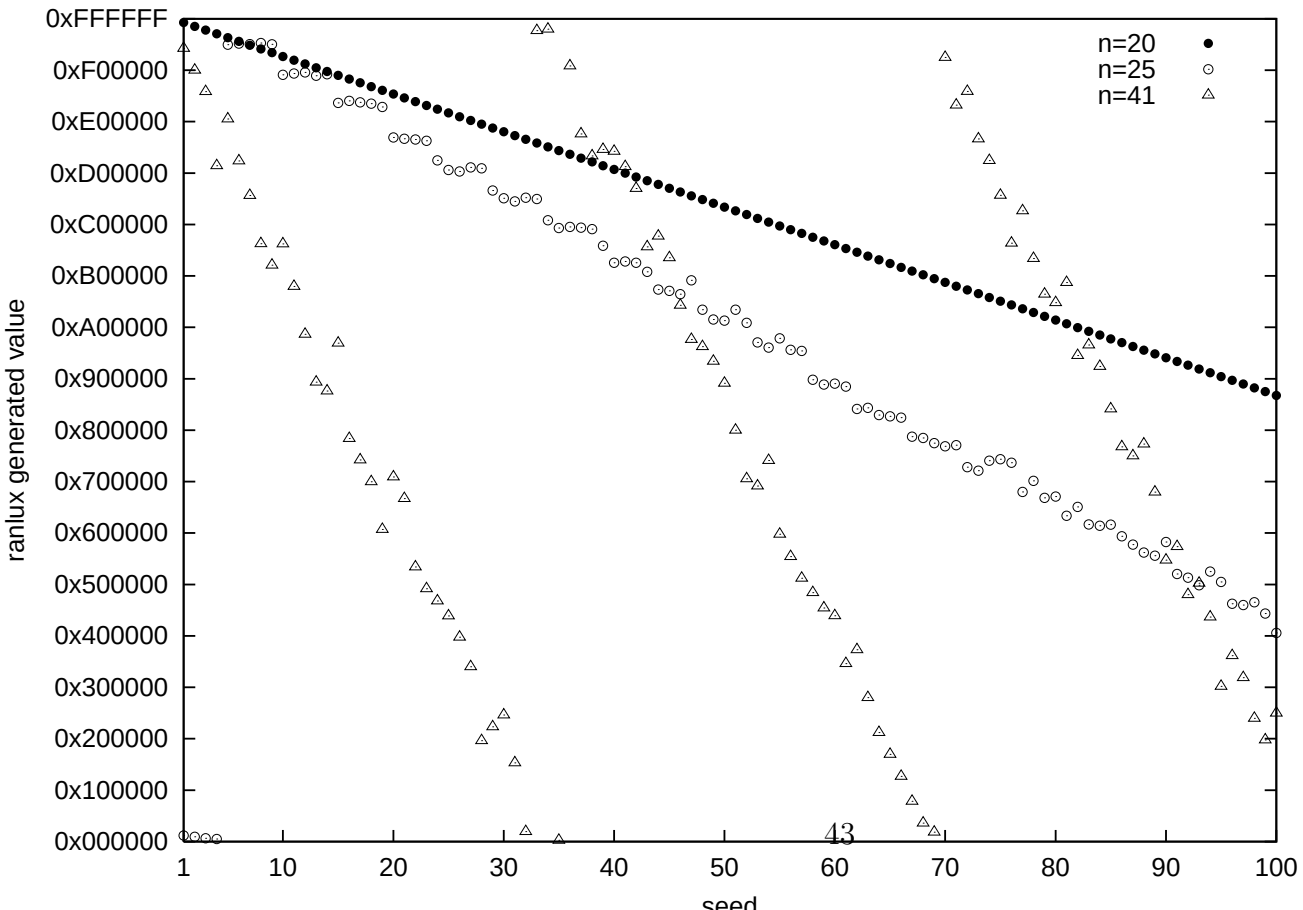
`random`: ’90-現在UNIX系C言語での標準的擬似乱数の最下位ビット0-1を見る。

横軸：過去31回中の1の個数

縦軸：その条件下で、次の8回が全て0の確率：



`ranlux`(カオス理論に基づく擬似乱数, Lüscher '94)で、
$20, 25, 40$番目の出力（縦軸）を、初期シードを$1, 2, 3, \ldots, 100$
（横軸）と動かしてプロット



初期値を系統的に選んだときの、非乱数性

GNU Scientific Libraryに入っている
58個の擬似乱数発生法のうち、最新のものも含めて
45個にこのような問題が観測された。
（Mersenne Twisterでは観測されなかった。）

総まとめ

擬似乱数は極めて大量に用いられる。微細な統計的偏りや、初期値へのわずかな依存性が、計算機の高速化・大規模化に伴いシミュレーションを狂わせる可能性がある。

この際、使用者は擬似乱数が原因だと中々気づけない。

$\Rightarrow$　精密なデタラメさを高速生成する必要性

それに答えるのが、「1+1=0の数学」

講演者は、

「擬似乱数をMTに変えたら、うまく動くようになった」

というメールをたくさんもらっている。

擬似乱数研究の混迷

⇐ 理論的かつ実用的な「擬似乱数の定義」がないため

「擬似乱数の定義」へのまるで異なる３アプローチ：

1. 記録しておかない限り再生不能なものを乱数という
   (Kolmogorov-Chaitin, '60末)
2. 数列の一部から、他の部分が
   計算量的に計算できないものをいう(Blum-Blum-Shub, '86)
3. 周期や高次元分布性といった指標を用いて、
   良い漸化式を探す(古典的, '45-)

MT は古典的な3番のアプローチだが、
良い指標を持つ漸化式を探すのに現代数学の手法を用いた。

終わり