Random Numbers

Pseudo Random Number Generators

Roberto Innocente

SISSA, Trieste
for the joint ICTP/SISSA MHPC.it master course

April, 2018 - **Edition 2.1**

Where it all began ?

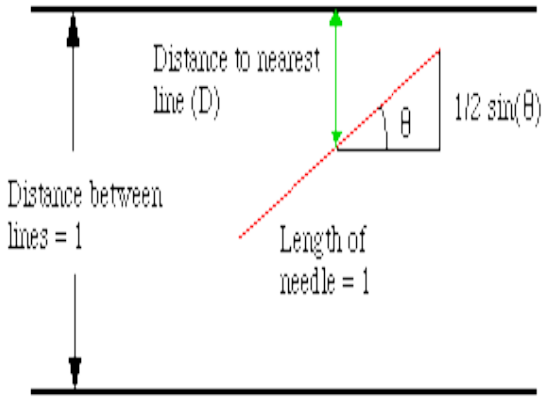
At the end of 1700 George Leclerc, Comte de Buffon tried to compute π with random experiments.

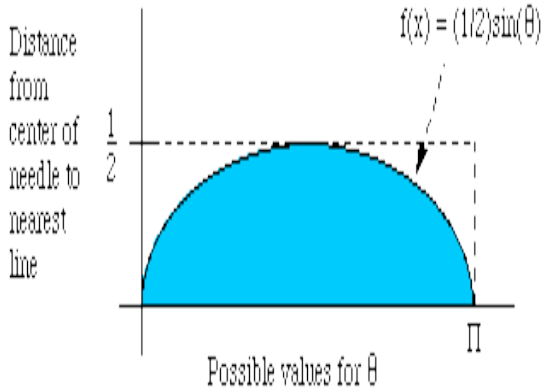
George-Louis Leclerc, Comte de Buffon, 1777 : Montecarlo method *ante-litteram*

A needle of length 1 is thrown over a lined paper with lines every 1 unit . The probability that it hits a line can be computed as :

$$Shaded\ \ Portion\ \ Area = \int_{\theta=0}^{\pi} \frac{sin(\theta)}{2} d\theta = \left[-\frac{cos(\theta)}{2} \right]_0^{\pi} = 1$$

$$Prob = \frac{Shaded\ \ Portion\ \ Area}{Area\ \ of\ \ rectangle} = \frac{1}{\pi/2} \quad \rightarrow \quad \pi = \frac{2}{Prob}$$





Example : Buffon's needle in R

```
# R code
buffonp <- function(n){
  k=0;
  for(i in 1:n){
        theta=runif(1,min=0,max=pi)
        y=runif(1,min=0,max=1/2);
        if (y+1/2*sin(theta)>1/2) k = k + 1 }
  return (k/n)
}
for (i in 1:6){
  w=10^i; bp=buffonp(w)
  cat('rn=',w,' computed pi=',2.0/bp,' error=',abs(pi-2.0/bp
      ),'\n')
}
#rn= 10  computed pi= 2  error= 1.141593
#rn= 100  computed pi= 3.389831  error= 0.2482379
#rn= 1000  computed pi= 3.04878  error= 0.09281217
#rn= 10000  computed pi= 3.134305  error= 0.007287686
#rn= 1e+05  computed pi= 3.138584  error= 0.003008469
#rn= 1e+06  computed pi= 3.144595  error= 0.003002102
```

RNG, TRNG, PRNG I

- **TRNG** (True Random Number Generators)
  - noise based e.g from atmospheric noise *https://www.random.org/*
  - free running oscillator (FRO) : simplest and cheapest way
  - chaos based
  - quantum based e.g. Geiger counters, fluctuations of vacuum *https://qrng.anu.edu.au/*
- **PRNG** (Pseudo Random Number Generators) or simply **RNG**
  - algorithmic

We want to be able to produce RN fast, in a cheap way, we need repeatability : we need algorithmic RNG !!.
There is a subclass of PRNG that will not be covered here and it is the class of **cryptographically robust** PRNG. These require the special quality of *unpredictability* that is computationally expensive and is never shown by the efficient PRNG we need for simulations.

Divide and conquer

Generation of Random Numbers is usually splitted into :

1. Generation of uniformly distributed integers x_i in $[0 \ldots (m-1)]$
2. Mapping of integers in $[0 \ldots (m-1)]$ to uniformly distributed reals in $U(0,1)$ using $u_i = \frac{x_i}{m}$. In many cases the 1st step is allowed to produce 0 while usually we want the 2^{nd} step not to produce it. Therefore often is used $u_i = \frac{x_i+1}{m+1}$.
3. Mapping of uniformly distributed reals in $\mathscr{U}(0,1)$ to the wanted CDF (Cumulative Distribution Function) $F(x)$ using most of the time its inverse $F^{-1}(x)$

This is the reason why we will center our discussion about uniform random numbers on $(0,1)$: $\sim \mathscr{U}(0,1)$.

How to map a uniform variate or deviate $\sim \mathscr{U}(0,1)$ in a differently distributed variate? I

We say that a sequence of numbers is a *sample from* a cumulative distribution function F, if they are a realization of a rv with CDF F. If F is the uniform distribution over $(0,1)$ then we call the samples from F *uniform deviates/variates* and we write $\sim \mathscr{U}(0,1)$.

Theorem : inversion

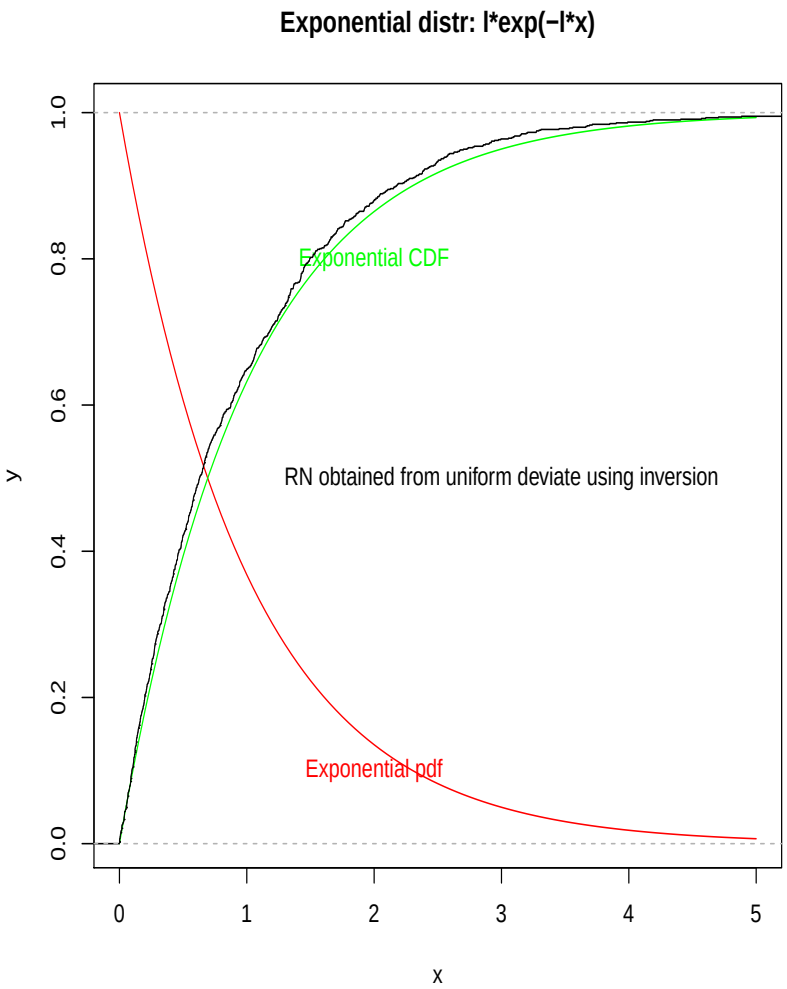
Suppose $U \sim \mathscr{U}(0,1)$ and F to be a continuos strictly increasing cumulative distribution function (CDF). Then $F^{-1}(U)$ is a sample from F .

Transform to a Normal deviate :

Box-Muller transform is more efficient:

1. generate $U_1 \sim \mathscr{U}(0,1)$ and $U_2 \sim \mathscr{U}(0,1)$
2. $\theta = 2\pi U_2$, $\rho = \sqrt{-2 \log U_1}$
3. $Z_1 = \rho \cos\theta$ is a normal variate

Example in R



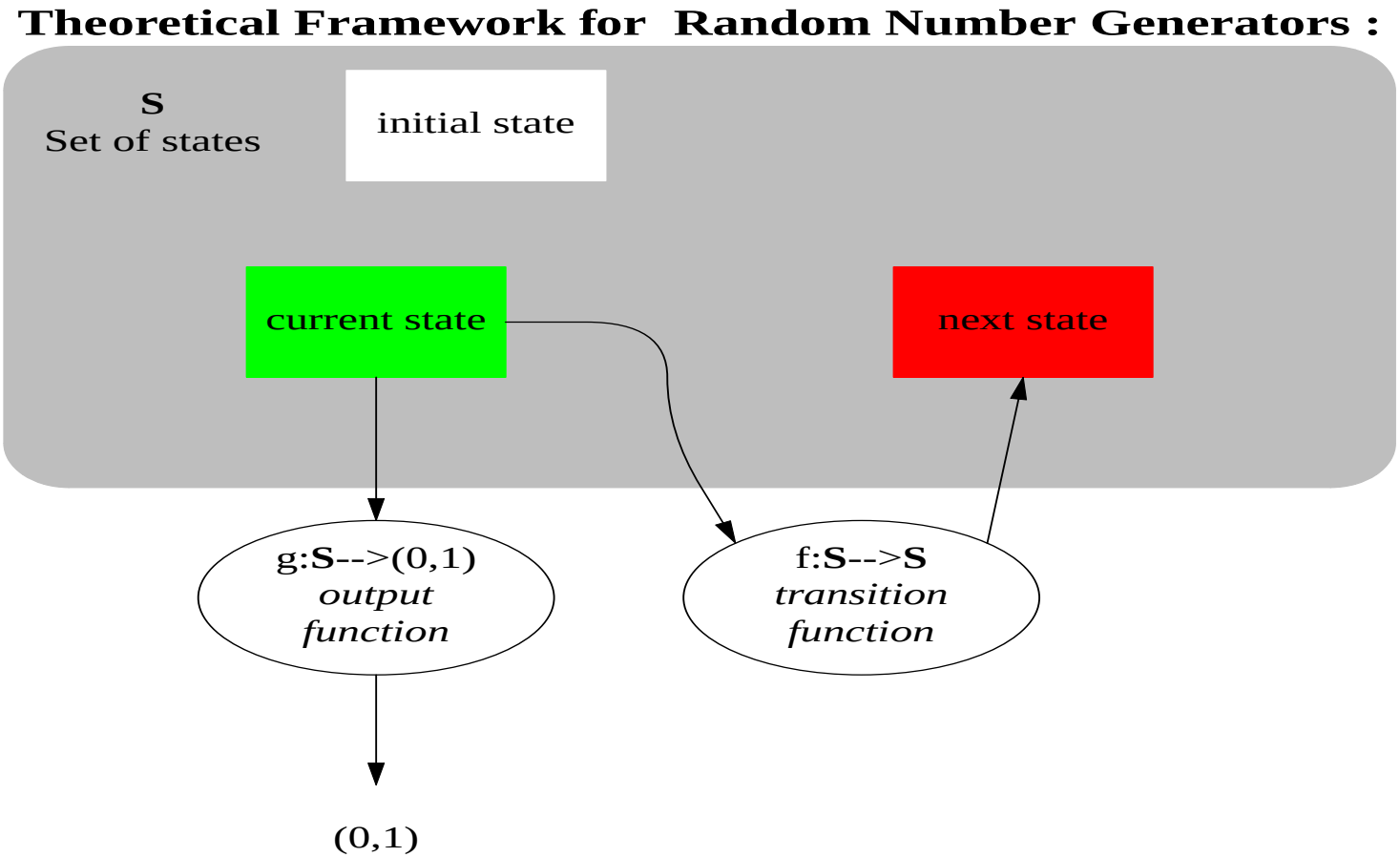

```
# exponential distribution: R code
pdf(file='exp.pdf')
lambda = 1; x=seq(0,5,0.05)
y=exp(-lambda*x); z=1-exp(-lambda*x)
plot(x,y,type='n')
title('Exponential distr: l*exp(-l*x
    )')
lines(x,y,col='red'); lines(x,z,col
    ='green')
text(3,0.5,'RN obtained from uniform
     deviate using inversion')
text(2,0.8,'Exponential CDF',col='
    green')
text(2,0.1,'Exponential pdf',col='
    red')

invcdf<-function(yy) { lambda = 1;
    xx=-log(1-yy)/lambda;
return (xx); }
w=runif(1000); ic = invcdf(w)
lines(ecdf(ic),xlim=c(0,5),ylim=c
    (0,1))
```

Qualities of Good RNG

- Good Theoretical Basis
- Long Period
- "pass" Empirical Tests
- Efficient
- Repeatable
- Portable

Theoretical framework

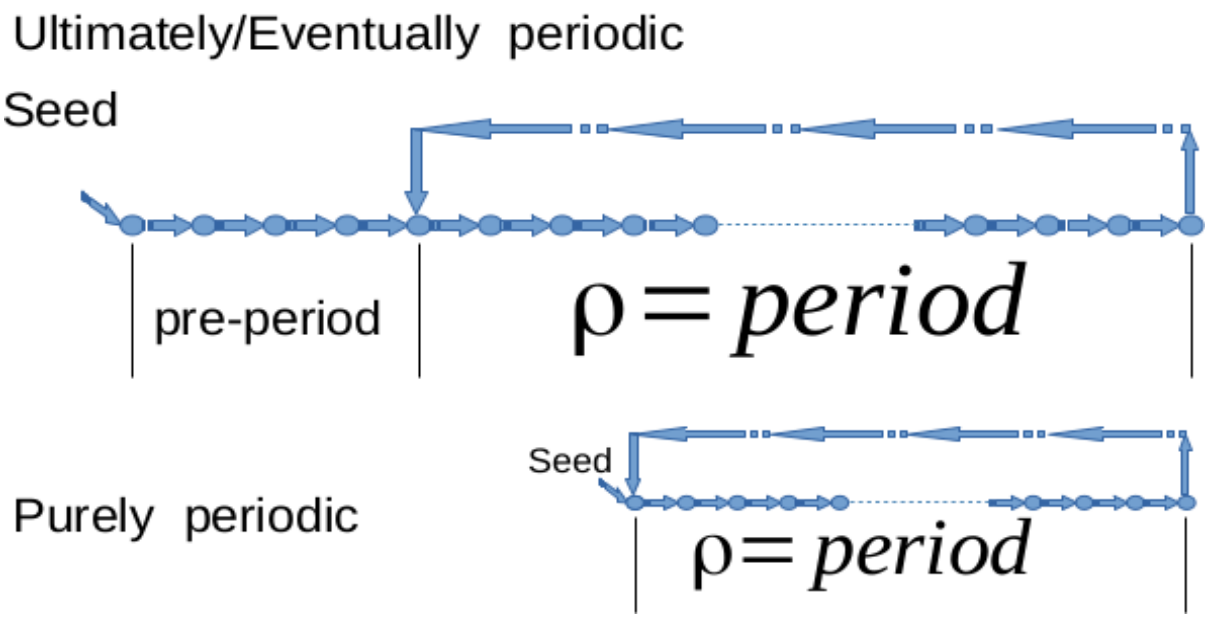


S set of all states/seeds (e.g. 1 integer -> 2^32 states)
f:**S**-->**S** *transition function* that moves the rng to the next state
g:**S**-->(0,1) *output function* that from a state outputs a number in the (0,1) interval

An upper bound on the period of the generator is the cardinality of **S** : |**S**|

Pre-period, Period of a RNG

Because the space for the state is finite and the RNGs should continue to produce rn, RNGs need to cycle. They can cycle in 2 ways : coming back to the initial state or coming back to an intermediate state skipping some initial states.



History of field based on scholars leaders

Von Neumann was maybe the first to devise an algorithm : the *middlesquare method*. A few leaders in the field during more than 75 years of electronic computing were in succession :

- Donald Knuth
- George Marsaglia
- Pierre L'Ecuyer
- Makoto Matsumoto

Donald Knuth

- born 1938, PhD at Caltech, worked at Stanford, now Professor Emeritus at Stanford
- Writer of the first bible of algorithms. 3 and now 4 books nick-named **TAOCP**: *The art of computer programming*.
- Creator of TEX and METAFONT and the Computer Modern family of fonts. Writer of the 5 books about them : *Computer and typesetting: A,B,C,D,E*
- Volume 2 of *The Art Of Computer Programming : Seminumerical Algorithms(1998)* dedicates the 189 pages of chapter 3 to *Random Numbers*. In it there is also the description of a battery of rng tests.

Knuth reports his attempt at a *Super-random* ng. Sometimes complexity hides simple behaviour

1. Take N to be the most significant digit of X. Steps 2–13 are repeated exactly $N + 1$ times.
2. Let M be the second most significant digit of X. Jump to step $3 + M$.
3. If $X < 5 \times 10^9$, set $X = X + 5 \times 10^9$.
4. Replace X by $\lfloor X^2/10^5 \rfloor \bmod 10^{10}$. (The notation $\lfloor x \rfloor$ means the largest integer n with $n \leq x$.)
5. Replace X by $(1001001001 \times X) \bmod 10^{10}$.
6. If $X < 10^8$ then set X to be $X + 9814055677$; otherwise, set X to be $10^{10} - X$.
7. Interchange the lower-order five digits of X with the higher-order five digits of X.
8. Replace X by $(1001001001 \times X) \bmod 10^{10}$.
9. For each digit d of X, decrease d by 1 if $d > 0$.
10. If $X < 10^5$, set X to be $X^2 + 99999$; otherwise, set X to be $X - 99999$.
11. If $X < 10^9$, set X to be $10 \times X$ and repeat this step.
12. Replace X by the middle 10 digits of $X(X - 1)$.
13. If $N > 0$, decrease N by one and return to step 2. If $N = 0$, the algorithm terminates with the current value of X as the next value in the sequence.

The first time Knuth ran this program, it converged quickly to 6065038420 (a fixed point of the algorithm). After this time it was mainly converging to a cycle of length 3178 !

Congruences: Integers modulo n, $\mathbb{Z}/n\mathbb{Z}$

$$a \sim b \text{ iff } n|(b-a)$$

This is an equivalence relation over $\mathbb{Z}$ and usually we indicate the classes that arise with their lowest non negative integer representative. So in $\mathbb{Z}/3\mathbb{Z}$ we use : $0, 1, 2$ for

- $\ldots, -6, -3, 0, 3, 6, 9, \ldots$
- $\ldots, -5, -2, 1, 4, 7, 10, \ldots$
- $\ldots, -4, -1, 2, 5, 8, 11, \ldots$

and so on . When we need to perform arithmetic, if we take the modulus only for the result or for each term changes nothing :

$$(27+12) \mod 5 = (27 \mod 5 + 12 \mod 5) \mod 5 = (2+2) \mod 5 = 4$$

$$(27+12) \mod 5 = 39 \mod 5 = 4$$

Theorem: $\mathbb{Z}_n$ is a commutative ring with identity.

It comes from the properties of $\mathbb{Z}$. If you ever asked yourself why integers are denoted by $\mathbb{Z}$, it comes from the initial of *Zahlen* that is the german word for numbers.

Small Commutative Rings $\mathbb{Z}_n$ I

$\mathbb{Z}_2, +$

$+$	0	1
0	0	1
1	1	0

$\mathbb{Z}_2^*, *$

$*$	1
1	1

$\mathbb{Z}_3, +$

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\mathbb{Z}_3^*, *$

$*$	1	2
1	1	2
2	2	1

Small Commutative Rings $\mathbb{Z}_n$ II

$\mathbb{Z}_4, +$

$+$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$\mathbb{Z}_4^*, *$

$*$	1	2	3
1	1	2	3
2	2	0	2
3	3	2	1

Small Commutative Rings $\mathbb{Z}_n$

$\mathbb{Z}_5, +$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\mathbb{Z}_5^*, *$

$*$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Theorem : If p is prime then $\mathbb{Z}_p$ is a finite field.

Both $(\mathbb{Z}_p, +)$ and $(\mathbb{Z}_p \setminus 0 = \mathbb{Z}_p^*, *)$ are abelian groups.

Theorem : The multiplicative group $\mathbb{F}^*$ of a finite field is cyclic.

There exist a *primitive* element α that generates the group : $1, \alpha, \alpha^2, \ldots$ are all the elements of the field.

Abstract structure

Tell what you can about the structure defined by this table ..

$$\mathbb{H}, \bullet$$

$\bullet$	k	w	d	n
k	d	k	n	w
w	k	w	d	n
d	n	d	w	k
n	w	n	k	d

Linear Congruential Generators or LCG I

LCG notation

$$x_{n+1} \equiv (a * x_n + c) \pmod{m}$$

will be indicated by $LCG(m, a, c, x_0)$. m is called the *modulus*, a the *multiplier*, c the *increment*, x_0 the starting value or **seed**. We use c like Knuth does and we set $b = a - 1$ for convenience.

Introduced by Lehmer in 1949 [14] . Sometimes when $c = 0$ they are called **Multiplicative LCG** or **MLCG** and denoted by $MLCG(m, a)$. When $c \neq 0$ **Mixed Linear Congruential**.
Lehmer generator is $u_0 \neq 0, u_{n+1} \equiv (23 * u_n) \pmod{10^8 + 1}$

- ANSIC $LCG(2^{31}, 1103515245, 12345, 12345)$
- MINSTD $LCG(2^{31-1}, 7^5, 0, 1)$
- RANDU $LCG(2^{31}, 2^{16}, 0, 1)$
- APPLE $LCG(2^{35}, 5^{13}, 0, 1)$
- Super-duper $LCG(2^{32}, 69069, 0, 1)$
- NAG $LCG(2^{59}, 13^{13}, 0, 2^{32} + 1)$
- DRAND48 $LCG(2^{48}, 25214903917, 11, 0)$

Linear Congruential Generators or LCG II

- $c = 0$ takes less time to compute, but cuts down the period of the sequence that anyway can still be long
- it can be proved that

$$x_{n+k} \equiv (a^k * x_n + (a^k - 1)c/b) \pmod{m}$$

that expresses the $n + k$ term in terms of the n term. In particular respect to x_0.

$$x_k \equiv (a^k * x_0 + (a^k - 1)c/b) \pmod{m}$$

That is: the subsequence consisting of every k^{th} term is also an LC sequence.

- Choice of m : should be large because it's a limit for the period ρ of the rng, should make it simple to compute $(a * x_n + c) \pmod{m}$,

Linear Congruential Generators or LCG III

$MLCG(m, a)$:

If m is prime and a is a *primitive root* of m and $x_0 \neq 0$ then the sequences $\{x_n\}$ are periodic with period length $\rho = m - 1$ and the generator is called a *full period* MLCG.
If $m = 2^w$ then the maximal period is $\rho = 2^{w-2} = m/4$ and is attained in particular when $a \equiv 5 \pmod 8$.

[17] We want a large m to make the grid of RN finer. But we need to keep m not larger than a computer word so that we can do operations efficiently. Therefore we choose $m \leq 2^{32}$ for 32-bit processors or $m \leq 2^{64}$ for 64-bit processor. The theory is nicer if m is prime or is a power of 2 like 2^w. Common choices:

32-bit proc		64-bit proc	
Prime	2^k	Prime	2^k
$m = 2^{31} - 1$	$m = 2^{32}$	$m = 2^{48} - 59$	$m = 2^{64}$
		$m = 2^{63} - 25$	
		$m = 2^{64} - 59$	

Linear Congruential Generators or LCG IV

$LCG(m, a, c, x_0)$:

An LCG has full period m if and only if :

1. The GCD(Greatest Common Divisor) of m and c is 1.
2. if q is a *prime* that divides m then q divides $(a-1)$.
3. if 4 divides m, then 4 divides $(a-1)$

(Hull-Dobell Theorem)

A lot of work has been done on these generators especially to give *multipliers* that provide as little as possible of *Marsaglia's effect*. You can't use them without reading [17] that for common word sizes computes the highest prime smaller than the largest integer and gives good *multipliers* , e.g.:

LCG : low order bits are less random

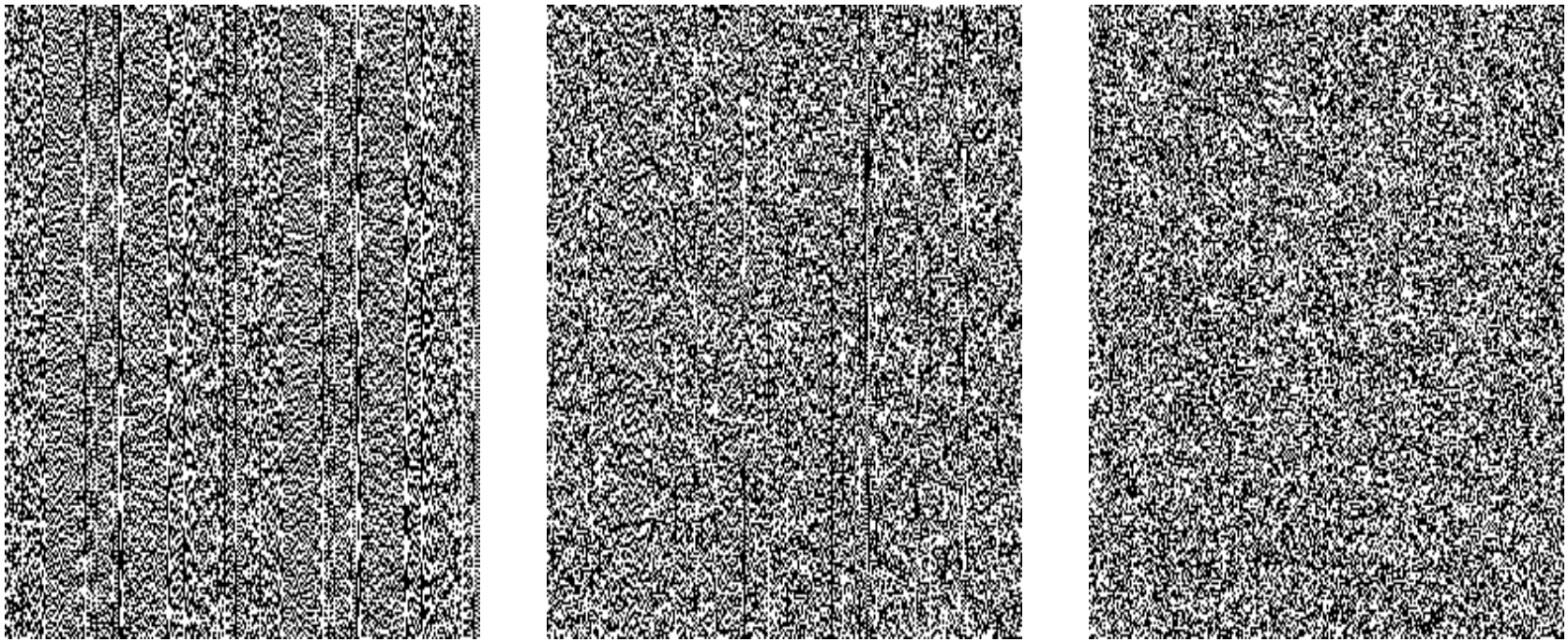


Figure: Lowest order bit 256x256 b_0, 256x256 third bit b_2, 256x256 successive bits

George Marsaglia

- born 1924, † 2011, PhD Ohio State, then University of Florida, University of Washington
- discovered what is called *Marsaglia's effect*. The successive n-tuples generated by Linear Congruential Generators (LCG) lie on a small number of equally spaced hyperplanes in n-dimensional space.
- developed the *diehard* statistical tests for rng, 1996
- developed many of the well known methods for generating rn : *multiply-with-carry*, *subtract with borrow*, *xorshift* , *KISS93*, *KISS99*, ...

Marsaglia's theorem

THEOREM 1. *If $c_1, c_2, \ldots, c_n$ is any choice of integers such that*

$$c_1 + c_2 k + c_3 k^2 + \ldots + c_n k^{n-1} \equiv 0 \text{ modulo } m,$$

then all of the points $\pi_1, \pi_2, \ldots$ will lie in the set of parallel hyperplanes defined by the equations

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = 0, \pm 1, \pm 2, \ldots .$$

There are at most

$$|c_1| + |c_2| + \ldots + |c_n|$$

of these hyperplanes which intersect the unit n-cube, and there is always a choice of $c_1, c_2, \ldots, c_n$ such that all of the points fall in fewer than $(n!m)^{1/n}$ hyperplanes.

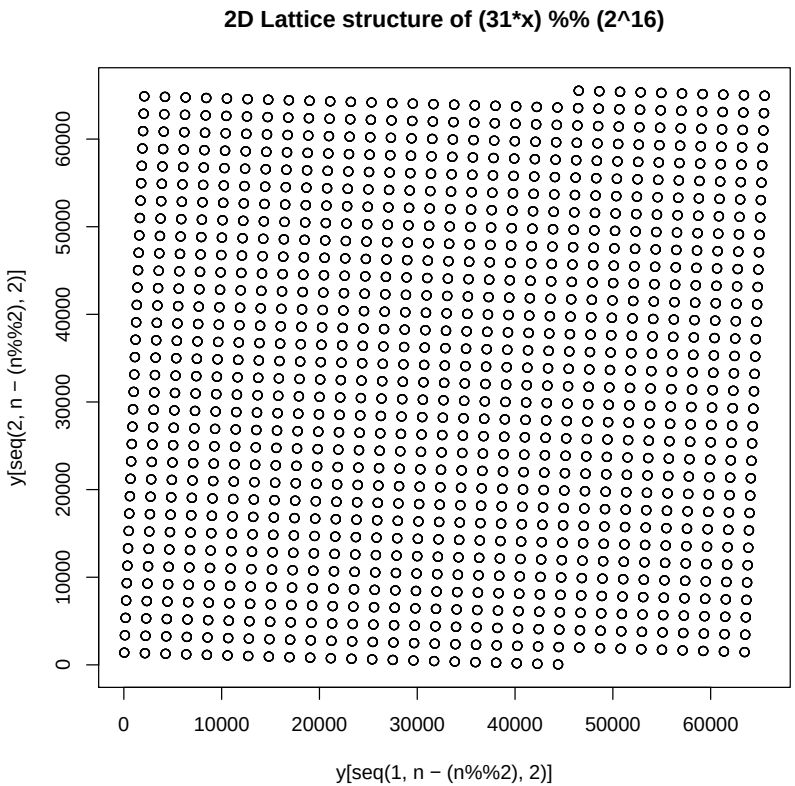
Here is a table of $(n!m)^{1/n}$ for the most common values of m, powers of 2:

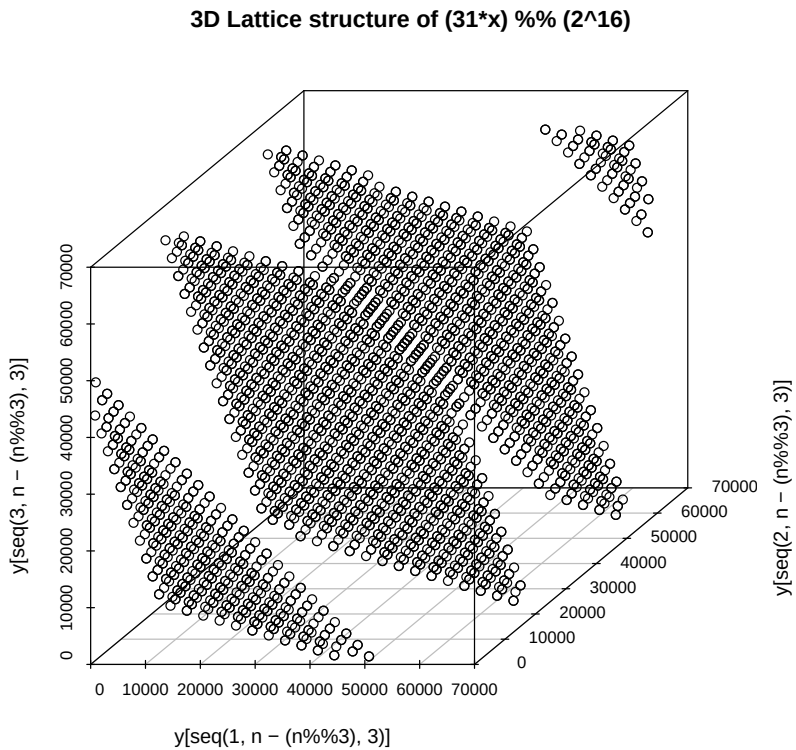
Upper Bound for the Number of Planes Containing All n-tuples

	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$
$m = 2^{16}$	73	35	23	19	16	15	14	13
$m = 2^{24}$	465	141	72	47	36	30	26	23
$m = 2^{32}$	2,953	566	220	120	80	60	48	41
$m = 2^{35}$	5,907	952	333	170	108	78	61	51
$m = 2^{36}$	7,442	1,133	383	191	119	85	66	54
$m = 2^{48}$	119,086	9,065	2,021	766	391	240	167	126

Lattice structure of LCG generators

Marsaglia's effect. Successive t-uples obtained from an LCG generator fall on, at most, $(t!m)^{1/t}$ parallel hyperplanes, where m is the modulus used in the LCG(marsaglia1968) :





Marsaglia's article *Random Numbers fall mainly in the plane* is a pun on *My Fair Lady* refrain *The Rain in Spain stays mainly in the plain*.

TABLE 5. LCGs with Good Figures of Merit, for $m = 2^e$ and $c = 0$

m	a, a^*	$M_8(m,a)$	$M_{16}(m,a)$	$M_{32}(m,a)$
2^{30}	177911525, 17372909	0.74878 *	0.53850	0.53850
	156051869, 52274357	0.69501	0.67940 *	0.64413
	143133861, 233896749	0.69305	0.66791	0.66791 *
2^{31}	594156893, 452271861	0.75913 *	0.50244	0.50244
	558177141, 413965533	0.68978	0.68749 *	0.59450
	602169653, 448899357	0.67295	0.67116	0.67116 *
2^{32}	741103597, 887987685	0.75652 *	0.53707	0.53707
	1597334677, 851723965	0.70068	0.67686 *	0.64694
	747796405, 204209821	0.66893	0.66001	0.66001 *
2^{33}	2185253333, 173170557	0.75896 *	0.49707	0.49707
	2174241325, 1406965157	0.68312	0.68289 *	0.62250
	2167985045, 1720311741	0.67787	0.67787	0.66548 *
2^{34}	11481271045, 3694381517	0.75466 *	0.56806	0.56806
	4324911125, 1620027197	0.67429	0.67105 *	0.58062
	4327278197, 3586136541	0.65630	0.65336	0.65336 *
2^{35}	8670442045, 2200188181	0.75818 *	0.51264	0.51264
	8622619205, 6073108621	0.68055	0.67255 *	0.60467
	22260253805, 7113024869	0.66619	0.66604	0.66604 *
2^{36}	4092856269, 14224997637	0.75662 *	0.50169	0.50169
	17229873325, 856580901	0.68442	0.67057 *	0.65570
	17246906533, 12512050989	0.69761	0.66579	0.66579 *
2^{48}	49402601338917, 5567195800493	0.75801 *	0.58062	0.58062
	70189847242853, 69036053825901	0.67618	0.66857 *	0.61586
	21749276838573, 66473811011877	0.65702	0.64692	0.64692 *
2^{60}	276137484736346373, 96397229732113357	0.75277 *	0.48916	0.48916
	150878991426218621, 243765350249586389	0.65527	0.65510 *	0.59498
	271413322654087621, 111008605039107341	0.64851	0.64851	0.64435 *
2^{63}	3512401965023503517, 1447878736930374069	0.74926 *	0.50092	0.50092
	2444805353187672469, 2079243811257762237	0.70937	0.66091 *	0.61403
	1987591058829310733, 1702126216606895045	0.64490	0.64060	0.63994 *
2^{64}	1181783497276652981, 4292484099903637661	0.76039 *	0.42672	0.42672
	7664345821815920749, 1865811235122147685	0.67778	0.66115 *	0.54884
	2685821657736338717, 1803442709493370165	0.65961	0.63932	0.63932 *
2^{128}	25096281518912105342191851917838718629, 55640593262044302480766460352317677869	0.76598 *	0.55122	0.55122
	23766634975743270097972271989927654085, 67836365537811707609274168323887561741	0.65708	0.65708 *	0.55662
	92563704562804186071655587898373606109, 42195469826238322466821139555285835125	0.63462	0.63462	0.63405 *

Figure: from l'Ecuyer 1988, MLCG : $m = 2^e$, $c = 0$

TABLE 4. LCGs with good figures of merit, for $m = 2^e$ and c odd

m	a	$M_8(m,a)$	$M_{16}(m,a)$	$M_{32}(m,a)$
2^{30}	438293613	0.75107 *	0.58300	0.58300
	523592853	0.70068	0.67686 *	0.64694
*	0.64694			
	116646453	0.67718	0.67420	0.67107 *
2^{31}	37769685	0.75896 *	0.51494	0.51494
	26757677	0.68312	0.68289 *	0.62474
	20501397	0.67787	0.67787	0.66548 *
2^{32}	2891336453	0.75466 *	0.56806	0.56806
	29943829	0.67429	0.67105 *	0.58062
	32310901	0.65630	0.65336	0.65336 *
2^{33}	3766383685	0.75029 *	0.56952	0.56952
	32684613	0.68055	0.67255 *	0.62595
	5080384621	0.66619	0.66604	0.66604 *
2^{34}	52765661	0.74421 *	0.54362	0.54362
	50004141	0.68442	0.67057 *	0.65570
	67037349	0.69761	0.66579	0.66579 *
2^{35}	22475205	0.74676 *	0.59182	0.59182
	15319397	0.67472	0.66933 *	0.60508
	15550228621	0.65734	0.65552	0.65552 *
2^{36}	12132445	0.75179 *	0.51869	0.51869
	8572309	0.66450	0.66389 *	0.63361
	33690453	0.68461	0.65808	0.65760 *
2^{40}	330169576829	0.75723 *	0.46879	0.46879
	42595477	0.67959	0.66436 *	0.60251
	33261733	0.65941	0.65477	0.65477 *
2^{48}	181465474592829	0.75812 *	0.54668	0.54668
	77596615844045	0.67653	0.66906 *	0.61130
	10430376854301	0.66530	0.64759	0.64759 *
2^{60}	454339144066433781	0.75956 *	0.57465	0.55002
	21828622668691829	0.65844	0.65566 *	0.63458
	395904651965728677	0.63944	0.63944	0.63944 *
2^{63}	9219741426499971445	0.73715 *	0.54235	0.54235
	2806196910506780709	0.69668	0.66519 *	0.60754
	3249286849523012805	0.64507	0.63523	0.63523 *
2^{64}	2862933555777941757	0.75673 *	0.55283	0.54445
	3202034522624059733	0.66164	0.66041 *	0.60256

Figure: from I'Ecuyer 1988, LCG : $m = 2^e$,c odd

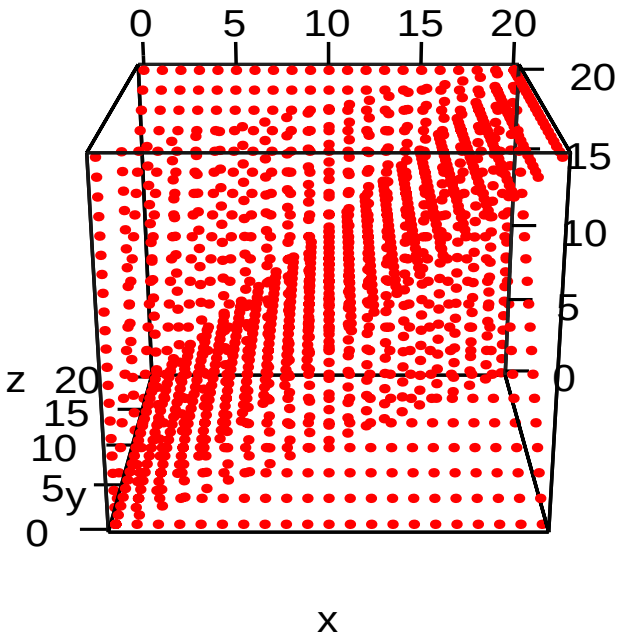
TABLE 2. LCGs with good figures of merit (continued)

m	a, a^*	$M_8(m,a)$	$M_{16}(m,a)$	$M_{32}(m,a)$
$2^{54}-33$	9131148267933071, 17639054895509756	0.71956 *	0.59136	0.59136
	3819217137918427, 6822546395505148	0.67456	0.65646 *	0.60358
	11676603717543485, 13197393252146039	0.66189	0.63663	0.63250 *
$2^{55}-55$	33266544676670489, 11719476530693442	0.73046 *	0.61066	0.55598
	19708881949174686, 32182684885571630	0.65421	0.65091 *	0.61035
	32075972421209701, 15995561023396933	0.62948	0.62948	0.62948 *
$2^{56}-5$	4595551687825993, 6128514294048584	0.72026 *	0.57724	0.57724
	26093644409268278, 69294271672288492	0.67840	0.66318 *	0.58207
	4595551687828611, 2389916809994467	0.64778	0.64243	0.62784 *
$2^{57}-13$	75953708294752990, 66352637866891714	0.72732 *	0.57473	0.56026
	95424006161758065, 2274812368615087	0.64856	0.64856 *	0.59464
	133686472073660397, 113079751547221130	0.64588	0.62957	0.62957 *
$2^{58}-27$	101565695086122187, 56502943171806276	0.77453 *	0.55885	0.55885
	163847936876980536, 256462492811829427	0.68047	0.66531 *	0.54314
	206638310974457555, 28146528635210647	0.64632	0.63406	0.63406 *
$2^{59}-55$	346764851511064641, 287514719519235431	0.71819 *	0.54325	0.54325
	124795884580648576, 526457461907464601	0.64928	0.64760 *	0.62279
	573223409952553925, 81222304453481810	0.64258	0.63111	0.63111 *
$2^{60}-93$	561860773102413563, 79300725740259852	0.72541 *	0.50786	0.50786
	439138238526007932, 998922549734761568	0.66098	0.65258 *	0.60350
	734022639675925522, 67273627956685463	0.66024	0.62375	0.62375 *
$2^{61}-1$	1351750484049952003, 2078173049752560138	0.71028 *	0.54999	0.54276
	1070922063159934167, 212694642947925581	0.63769	0.63769 *	0.56108
	1267205010812451270, 1283839219676404755	0.63648	0.62092	0.62092 *
$2^{62}-57$	2774243619903564593, 1983373718104285921	0.72982 *	0.61073	0.59560
	431334713195186118, 1159739479727509578	0.64966	0.64180 *	0.59560
	2192641879660214934, 2674546532986414750	0.62431	0.62374	0.62374 *
$2^{63}-25$	4645906587823291368, 60091810420728157	0.73855 *	0.50741	0.50741
	2551091334535185398, 9006541669060512547	0.65169	0.64418 *	0.58261
	4373305567859904186, 6458928179451363983	0.62582	0.62582	0.62497 *
$2^{64}-59$	13891176665706064842, 9044836419713972268	0.74105 *	0.36297	0.36297
	2227057010910366687, 17412224886468018797	0.68377	0.64579 *	0.52405
	1826344031245878947 1, 811465980874026894	0.63276	0.62970	0.62970 *
$2^{127}-1$	82461096547334812307256211668490605096, 335418441556692015730452773549859611 33	0.74702 *	0.50027	0.50027
	11378330613449548425753788132509481 5818, 549754870954195569833520341422926719	0.63462	0.62590 *	0.56105
	29590761937684265566924671478132826269			

Figure: l'Ecuyer 1988, LCG : m prime

Points with at least 2 coords equal are skipped

If we use these generators in 2 dim the diagonal points where $(x = y)$ are skipped because the generators produce only once every possible outcome. These are the excluded points in a 20x20x20 lattice :



Minimal Standard MINSTD Park-Miller rng(1988)

$7^5 = 16807$, $2^{31} - 1 = 2147483647$

MINSTD

$$x_n \equiv 16807 * x_{n-1} \pmod{2^{31} - 1}\ ,\ LCG(7^5, 0, 2^{31} - 1)$$

Stephen K. Park; Keith W. Miller (1988).
Random Number Generators: Good Ones Are Hard To Find

> *Given the dynamic nature of the area, it is difficult for nonspecialists to make decisions about what generator to use. "Give me something I can understand, implement and port... it needn't be state-of-the-art, just make sure it's reasonably good and efficient." Our article and the associated minimal standard generator was an attempt to respond to this request. Five years later, we see no need to alter our response other than to suggest the use of the multiplier a = 48271 in place of 16807.*

Fibonacci, Lagged Fibonacci(Marsaglia 1983) I

Probably you know the Fibonacci's sequence : an attempt made by Leonardo Fibonacci (aka il Pisano, born in Pisa ~ 1175, $\dagger \sim 1245$) to model the growth of a population of rabbits $x_n = x_{n-1} + x_{n-2}$. Why not to generate rn based on a longer previous history ?

Fibonacci

$$X_n = X_{n-1} \diamond X_{n-2} \pmod{m}\ ,\ \text{denoted } F(1, 2, \diamond)$$

has poor distribution qualities.

Lagged Fibonacci

$$X_n = X_{n-k} \diamond X_{n-l} \pmod{m}\ ,\ \text{denoted } F(k, l, \diamond)$$

$\diamond$ is a generic operator from $+, -, *, \oplus$. $\oplus$ is the XOR binary operator. k , l are called *lags* . Lags larger then 16 produce good rng. $k = 24, l = 55$ was studied extensively, like $30, 127$. They

Fibonacci, Lagged Fibonacci(Marsaglia 1983) II

were used extensively, but in the ' 90 it was discovered that they fail a famous test of randomness (but a workaround exists). The one proposed by Marsaglia is $x_n \equiv x_{n-5} + x_{n-17} \pmod{2^k}$. Period of this lagged Fibonacci is $2^k * (2^{17} - 1)$, quite longer than LCGs. State is an array of 17 integers.
For a prime m and some choices of k, l, properly choosing an initial vector gives a sequence with a period of $m^k - 1$ [35].

Multiply With Carry : **MWC**, Marsaglia

Concatenates 2 16-bit multiplies with carry (period $\sim 2^{60}$) :

MWC:

Initial values: $z_0 = 362436069$, $w_0 = 521288629$

$$z_n \equiv 36969 * (z_{n-1} \& (2^{16} - 1)) + z_{n-1} >> 16$$

$$w_n \equiv 18000 * (w_{n-1} \& (2^{16} - 1)) + w_{n-1} >> 16$$

$$output = (z_n << 16) + w_n$$

Add With Carry AWC , Subtract With Borrow SWB (Marsaglia and Zaman 1991)

AWC

```
x[i] = (x[i-r] + x[i-s]+c[i-1]) \% m;
c[i] = (x[i-r]+x[i-s]+c[i-1])/m;
```

Initial state : $S[0..k-1]$ contains k initial integers $(x_0, \ldots, x_{k-1})$ and $c = c_0$, $k = max(r, s)$. Today considered not good.

SWB

```
/*Global static v a r i a b l e s :*/
static UL z=362436069,w=521288629,jsr=123456789,
          jcong=380116160;
static UL a=224466889,b=7584631,t[256],x=0,y=0,
          bro;
static unsigned char c=0;

#define SWB (c++, bro=(x<y),t[c]=(x=t[UC (c+34)])-(y=t[UC (c+19)]+bro))
```

[16]

32-bit integer GFSR using lagged Fibonacci with subtraction, $F(100, 37, -)$. State 100 integers, 400 bytes. Period $\sim 2^{129}$

Knuth-TAOCP-2002

$$x_j = (x_{j-100} - x_{j-37}) \pmod{2^{30}}$$

Multiple Recursive Generator : **MRG I**

MRG:

$$x_i = a_1 x_{i-1} + \ldots + a_k x_{i-k} \pmod{m}\ ,\ i \geq k$$

where m and k are positive integers called *modulus* and *order* and the *coefficients* $a_1, \ldots, a_k$ are in $\mathbb{Z}_m$. The *state* at step i is $s_i = (x_{i-k+1}, \ldots, x_i)^T$ (a vector of length k). The *initial state* s_0 is required to be different from all 0. When $m = p$ is a prime number the ring $\mathbb{Z}_p$ is a *finite field* and it is possible to choose the a_j in such a way that the period reaches $\rho = p^k - 1$ (Knuth, 1998). This maximal period is achieved iff the *characteristic polynomial* of the recurrence $P(z) = z^k - a_1 z^{k-1} - \ldots - a_k$ is a *primitive polynomial*. Alanen and Knuth gave 3 conditions for verifying the primitivity of $P(z)$. In addition, a maximum-period MRG is known to be *equidistributed up to k-dimensions* : every t-uple of $\mathbb{Z}_p$

Multiple Recursive Generator : **MRG** II

appears exactly p^{k-t} times over the entire period $p^k - 1$, except the all-zeroes t-uple that apeears one time less.(See Niederreiter[9])

Matrix Congruential Generators I

An MRG can be implemented as a **matrix multiplicative congurential generator**, which is a generator with state $S_t = \mathbf{X}_t \in \{0, \ldots, m-1\}^k$ for some modulus m and transition :

$$\mathbf{X}_t = A\mathbf{X}_{t-1} \pmod{m}, t = 1, 2, \ldots$$

The output is often taken to be :

$$\mathbf{U}_t = \frac{\mathbf{X}_t}{m}$$

where A is an invertible $k \times k$ matrix and $\mathbf{X}_t$ is a $k \times 1$ vector.

Long Polynomial Division

Long Polynomial division is very important in finite fields :

$$
\begin{array}{rl|l|l}
 & 2x^3+3x^2+x-1 & x^2+1 & \textbf{Divisor} \\
- & 2x^3+2x & 2x+3 & \textbf{Quotient} \\
\hline
 & 3x^2-x-1 & & \\
- & 3x^2++3 & & \\
\hline
 & -x-4 & & \textbf{Remainder}
\end{array}
$$

Finite fields of prime power order $\mathbb{F}_{p^n}$

There are no fields of order 6 because $6 = 2 * 3$. We have instead finite fields of order $4 = 2^2, 8 = 2^3, 9 = 3^2, 16 = 2^4, 27 = 3^3$. For any **prime** p and integer n, we have a finite field of order p^n.

Theorem : All finite fields of the same order are isomorphic.

. . .

Theorem : For every prime p and integer n there is a finite field of order p^n.

We have already seen the $\mathbb{Z}_p$ fields, all the remaining fields are isomorphic to the polynomials with coefficients in $\mathbb{Z}_p$ modulo an **irreducible** polynomial of order n.

$\mathbb{F}_4 = GF(2^2)$, Finite fields of not prime characteristic

Polynomials over $\mathbb{Z}_2$ modulo $x^2 + x + 1$.

$\mathbb{F}_4, +$

+	0	1	x	x+1
0	0	1	x	x+1
1	1	0	x+1	x
x	x	x+1	0	1
x+1	x+1	x	1	0

$\mathbb{F}_4, *$

*	1	x	x+1
1	1	x	x+1
x	x	x+1	1
x+1	x+1	1	x

$$(x+1)*(x+1) = x^2 + x + x + 1 = x^2 + 1 \mod (x^2 + x + 1) = -x = x$$

Bijection between integers and polynomials over $\mathbb{Z}_p$

$$a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0$$

Every a_i can take all values in $\mathbb{Z}_p$, therefore a p-digit can represent one of the a_i, and $n+1$ p-digits can represent all the p^{n+1} polynomials of order n or less over $\mathbb{Z}_p$.
For instance let's take $\mathbb{Z}_3$ and see what poly the number 14 represents. If we write 14 in base 3 we get $112_3 = 1 * 3^2 + 1 * 3 + 2$.
Therefore it represents the polynomial over $\mathbb{Z}_3$:

$$x^2 + x + 2$$

Number Theory (*hors d'oeuvre*) : Mersenne primes I

Mersenne numbers are those integers of the form $2^k - 1$.
Mersenne primes are those mersenne numbers that are primes. A basic theorem says that **if** $2^k - 1$ **is prime then also** k **is prime**.
Since 1997 all newfound mersenne primes were discovered thru the *Great Internet Mersenne Prime Search* (`https://en.wikipedia.org/wiki/Great_Internet_Mersenne_Prime_Search`) In *maxima* you find some with (memory hog) :

```
for k:1 thru 20000 step 2 do
if primep(k) then if primep(2^k-1) then print(k);
```

Number Theory (*hors d'oeuvre*) : Mersenne primes II

2^2-1	2^3-1	2^5-1	2^7-1
$2^{13}-1$	$2^{17}-1$	$2^{19}-1$	$2^{31}-1$
$2^{61}-1$	$2^{89}-1$	$2^{107}-1$	$2^{127}-1$
$2^{521}-1$	$2^{607}-1$	$2^{1279}-1$	$2^{2203}-1$
$2^{2281}-1$	$2^{3217}-1$	$2^{4253}-1$	$2^{4423}-1$
$2^{9869}-1$	$2^{9941}-1$	$2^{11213}-1$	$2^{19937}-1$
$2^{21701}-1$	$2^{23209}-1$	$2^{44497}-1$	$2^{86243}-1$
$2^{110503}-1$	$2^{132049}-1$	$2^{216091}-1$	$2^{756839}-1$
$2^{859433}-1$	$2^{1257787}-1$	$2^{1398269}-1$	$2^{2976221}-1$
$2^{3021377}-1$	$2^{6972593}-1$	$2^{13466917}-1$	$2^{20996011}-1$
$2^{24036583}-1$	$2^{25964951}-1$		

Table 1.2 Known Mersenne primes (as of Apr 2005), ranging in size from 1

Number Theory (*hors d'oeuvre*) : Mersenne primes III

Recently Richard Brent ([30]), devised a new fast algorithm to find primitive polynomials of Mersenne prime degree and reported 12 new found very large primitive trinomials over $\mathbb{Z}_2$:

r	s	Date
42 643 801	55981, 3706066, 3896488, 12899278, 20150445	2009
43 112 609	3569337, 4463337, 17212521, 21078848	2009
57 885 161	none	2013
74 207 281	9156813, 9999621, 30684570	2016

TABLE 1. New primitive trinomials x^r+x^s+1 of degree a Mersenne exponent r, for $s \leq r/2$. For smaller exponents, see references in [5] or our web site [1].

Number Theory : modular arithmetic : ring $\mathbb{Z}/n\mathbb{Z}$ and finite fields $\mathbb{Z}_p$ I

Integers modulo m or *congruence classes* form a commutative ring.
If p is *prime* then $\mathbb{Z}_p$ is a **finite field** also called **Galois field** GF.
If $gcd(a, m) = 1$ the least positive h for which $a^h \equiv 1 \pmod{m}$ is called the **multiplicative order** of a modulo m.
if p is prime and $gcd(g, p) = 1$ and the multiplicative order of g modulo p is $(p-1)$ then g is called a **primitive root**. ($p-1$ is the max multiplicative order of g according to *Fermat's little theorem*).
We said for every prime p, $\mathbb{Z}_p$ is a finite field. Are those the only finite fields ? No. All finite fields have cardinality p^n and finite fields with same cardinality are *isomorphic*. Exponents of the non *prime fields* of order p^n are the remainder classes of polynomials over $\mathbb{Z}_p$ (with coefficients in $\mathbb{Z}_p$) modulus a monic *irreducible polynomial* of degree n over $\mathbb{Z}_p$. For every n there is always at least one. A notation for a generic finite field is $\mathbb{F}_{p^n}$ or $GF(p^n)$.

Number Theory : modular arithmetic : ring $\mathbb{Z}/n\mathbb{Z}$ and finite fields $\mathbb{Z}_p$ II

Irreducible polynomials over finite fields Are those polynomials that cannot be factored into non trivial polynomials over the same field.
(Crandall, Pomerance, 2005) [10]
Theorem : If $f(x)$ is a polynomial in $\mathbb{F}_p[x]$ of positive degree k, the following statements are equivalent :

1. $f(x)$ is irreducible;
2. $gcd(f(x), x^{pj} - x) = 1$ for each $j = 1, 2, \ldots, \lfloor k/2 \rfloor$
3. $x^{p^k} \equiv x \pmod{f(x)}$ and $gcd(f(x), x^{p^{k/q}} - x) = 1$ for each prime $q \mid k$.

Algorithm 2.2.9 (Crandall): is $f(x)$ irreducible over $\mathbb{F}_p$?

Number Theory : modular arithmetic : ring $\mathbb{Z}/n\mathbb{Z}$ and finite fields $\mathbb{Z}_p$ III

```
[initialize]
g(x) = x
[Testing loop]
for p:1 thru floor(k/2) {
g(x) : g(x)^p mod f(x);
d(x) : gcd(f(x),g(x)-x);
if d(x) != 1) return(NO);
}
return(YES);
```

Primitive polynomials are those that have a root that is a primitive root, that is, its powers generate all the elements of the finite field. AN *irreducible polynomial* $F(x)$ of degree m over $GF(p)$ where p is prime is a *primitive polynomial* if the smallest integer such that $F(x)|x^n - 1$ is $n = p^m - 1$.
In the case of trinomials over $GF(2)$ the test is simple. For every r that is the exponent of a *Mersenne prime* $2^r - 1$ a trinomial of degree r is primitive iff it is *irreducible*.

Packages for Computations in Finite Fields

- **Maxima** : a symbolic algebra package Descendant of the famous MIT Macsyma, now in the public domain. An all round CAS but not efficient as others.
- **GAP** Groups,Algorithms and Programming started at the beginning of the '90 at RWTH Aachen, last distribution came from Colorado State.
- **Magma** probably today one of the top for professional activity in the field, maintained by University of Sidney, but quite expensive
- **Axiom**
- **sagemath**
- plus of course the main CAS systems : Mathematica, Maple, MATLAB, ...

An escape : Maxima, package *gf* for finite fields computations I

*F.Caruso, et. al.***Finite fields Computations in Maxima**

```
gf_set_data(p,m(x) );
gf_set_data(3,x^3+x
    ^2+1);
a:2*x^3+x^2+1;
b:x^2-1;
gf_add(a,b);
gf_mult(a,b);
gf_inv(b);
gf_div(a,b);
gf_mul(a,gf_inv(a) );
gf_exp(a,2);
make_list(gf_random(),
    i,1,4);
```

```
mat:genmatrix(lambda
    (\[i,j\],
    gen_random()),3,3)
    ;
gf_irreducible_p();
gf_primitive_poly_p();
gf_primitive();
gf_index(a);
gf_p2n(a);
gf_n2p();
gf_logs(3);
gf_powers(2);
```

Combined generators I

They were a great advance in the RNG arena. Here the heuristic is that combining generators maybe of not so good quality for today standard and shuffling, adding or selecting could make a better generator. One class that was thoroughly studied was that of **Combined MRG**. In some cases the theory can predict the period.
Methods used:

- Add rn from 2 or more generators. If x_i and y_i are sequences in $[0..(m-1)]$ then $x_i + y_i \pmod{()m}$ is also a sequence in $[0..(m-1)]$.
- XOR rn from 2 or more generators (Santa, Vazirani 1984)
- Shuffle with a rn generator x_i the output from another rn generator y_i (Marsaglia, Bray 1964) (e.g. keep last 100 items from sequence y_i use x_i to choose from this buffer.

Combined LCG I

Proposition If the w_i are L independent discrete rv such that w_i is uniform between 0 and $d-1$:

$$P(w_i = n) = \frac{1}{d}$$

then

$$W = \sum_{j=1}^{L} w_i \pmod{d}$$

is uniform over $0 \ldots (d-1)$.

Proposition if we have a family of L generators where the generator j has period p_j and evolves according to the *transition function*

$$s_{j,i} = f_j(s_{j,i-1})$$

then the period of the sequence $s_i = (s_{1,i}, \ldots, s_{L,i})$ where $s_0 = (s_{1,0}, \ldots, s_{L,0})$ is a given seed is the least common multiple of $p_1, \ldots, p_L$.

Combined MRG I

An MRG of **order** m is defined by :

$$x_n = a_1 x_{n-1} + \ldots + a_k x_{n-k}$$

$$u_n = x_n / m$$

where$m$ and k are positive integers and each a_i belongs to $\mathbb{Z}_m$. This recurrence has maximal period length $m^k - 1$ attained iff m is prime and the characteristic polynomial
$P(z) = z_k - a_1 z^{k-1} \ldots - a_k$ is primitive. The last condition to avoid too many computations can often be achieved with only 2 non zero coefficients like a_r and a_k with $1 <= r < k$. If we have L MRGs $\forall l \mid 0 \leq l < L - 1$:

$$x_{l,n} = a_{l,1} x_{l,n} + \ldots + a_{l,k} x_{l,n-k} \pmod{m_l}$$

Combined MRG II

with m_l distinct primes and the recurrences have order k and period $m_l^k - 1$, let d_l be arbitrary integers each prime with m_l for each l, define :

$$w_n = \sum_{l=1}^{L} d_l \frac{x_{l,n}}{m_l} \pmod 1$$

$$z_n = \sum_{l=1}^{L} d_l x_{l,n} \pmod{m_1}$$

$$u_n = z_n / m1$$

then w_n is exactly equivalent to an MRG with modulus $m = m_1 m_2 \ldots m_L$. (L'Ecuyer 1998).

Wichman-Hill generator

This was one of the earliest combined generators. It combines 3 LCG.

Wichman-Hill

$$X_t = 171X_{t-1} \pmod{m_1}\ ,\ (m_1 = 30629)$$

$$Y_t = 172Y_{t-1} \pmod{m_2}\ ,\ (m_2 = 30307)$$

$$Z_t = 170Z_{t-1} \pmod{m_3}\ ,\ (m_3 = 30323)$$

$$U_t = \frac{X_t}{m_1} + \frac{Y_t}{m_2} + \frac{Z_t}{m_3}$$

The period of the triples (X_t, Y_t, Z_t) is shown to be :
$(m_1 - 1)(m_2 - 1)(m_3 - 1) \sim 6.95 \times 10^{12}$
Performs well in tests, but the period is small.

L'Ecuyer MRG32k3a combined MRG I

A very famous combined MRG that was used extensively. Employs 2 MRG of order 3. The approximate period is $3 * 10^{57}$. It passes all tests in TestU01. It is implemented in *MATLAB*, *Mathematica*, *IntelMKL* library, *SAS*, etc.

MRG32k3a

$$X_t = (1403580 * X_{t-2} - 810728 * X_{t-3}) \pmod{m_1},\ m_1 = 2^{32} - 209$$

$$Y_t = (527612 * Y_{t-1} - 1370589 * Y_{t-3}) \pmod{m_2},\ m_2 = 2^{32} - 22853$$

$$U_t = \frac{X_t - Y_t + m_1}{m_1 + 1} \text{ if } X_t \leq Y_t\ ,\ \frac{X_t - Y_t}{m_1 + 1} \text{ if } X_t > Y_t$$

L'Ecuyer MRG32k3a combined MRG II

```
#define norm 2.32830654929572e-10
#define m1   4294967087.0
#define m2   4294944443.0
#define a12     1403580.0
#define a13n     810728.0
#define a21      527612.0
#define a23n    1370589.0
#define SEED 12345
static double s10 = SEED,
   s11 = SEED,s12 = SEED,s20 =
       SEED,
   s21 = SEED, s22 = SEED;
double MRG32k3a (void)
{
   long k; double p1, p2;
   /* Component 1 */
   p1 = a12 * s11 - a13n * s10;
   k = p1 / m1; p1 -= k * m1;
   if (p1 < 0.0) p1 += m1;
   s10 = s11; s11 = s12; s12 = p1;
   /* Component 2 */
   p2 = a21 * s22 - a23n * s20;
   k = p2 / m2; p2 -= k * m2;
   if (p2 < 0.0) p2 += m2;
   s20 = s21; s21 = s22; s22 = p2;
   /* Combination */
   if (p1 <= p2)
      return ((p1 - p2 + m1) *
          norm);
   else
      return ((p1 - p2) * norm);
}
```

In *MATLAB*/*Octave* :

L'Ecuyer MRG32k3a combined MRG III

```
m1=2^32-209; M2=2^32-22853;
ax2p=1403580; ax3n=810728;
ay1p=527612;ay3n=1370589;

X=[12345 12345 12345] % initial X
Y=[12345 12345 12345] % initial Y

N=100; % compute N rn
U=zeros(1,N);
for t:1:N
        Xt=mod(ax2p*X(2)-ax3n*X(3),m1);
        Yt=mod(ay1p*Y(1)-ay3n*Y(3),m2);
        if Xt <= Yt
                U(t)=(Xt-Yt+m1)/(m1+1);
        else
                U(t)=(Xt-Yt)/(m1+1);
        end
        X(2:3)=X(1:2);X(1)=Xt;Y(2:3)=Y(1:2);Y(1)=Yt;
end
```

Fourier DFT (spectral) test

Coveyou [33].
This test is now also in the NIST test library SP800-22.

- The sequence of 0 and 1 is changed to -1 and 1
- The DFT is applied to discover peaks in this sequence
  The Fourier coefficients : $S_j(X)_{j=0}^{n-1}$ are produced :

$$S_j = \sum_{k=0}^{n-1} x_k cos \frac{2\pi kj}{n} - i \sum_{k=0}^{n-1} x_k sin \frac{2\pi kj}{n}$$

$$c_j(X) = \sum_{k=0}^{n-1} x_k cos \frac{2\pi kj}{m}$$

$$s_j(X) = \sum_{k=0}^{n-1} x_k sin \frac{2\pi kj}{m}$$

Spectral test I

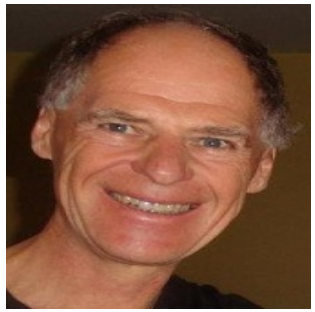
Knuth says: *all good rng pass this test, all bad fail it : it is a very important test*. Usually the set of overlapping vectors :

$$L_s = \{(x_n, x_{n+1}, \ldots, x_{n+s-1}) \mid n \geq 0\}$$

is considered. This set exhibits a lattice structure for many pseudorandom number generators such as LCG, multiple recursive, lagged-Fibonacci, add-with-carry, subtract-with-borrow, combined LCG, combined MRG. The test measures the maximal distance d_s between adjacent parallel hyperplanes that cover all vectors x_n.
http://random.mat.sbg.ac.at/tests/theory/spectral/
An algorithm is based on the dual lattice derived from L_s. The maximal distance is equal to one over the shortest vector in the *dual lattice*.

Pierre L'Ecuyer

- University of Montreal, Canada
- developed together with R.Simard **TestU01** : a C library that performs empirical randomness tests, 2007
- developed the famous *combined generator* **MRG32k3a**
- developed with F.O. Panneton and M.Matsumoto
  WELL (*Well Equidistributed Long-period Linear* rng : one of the rising stars among rng)



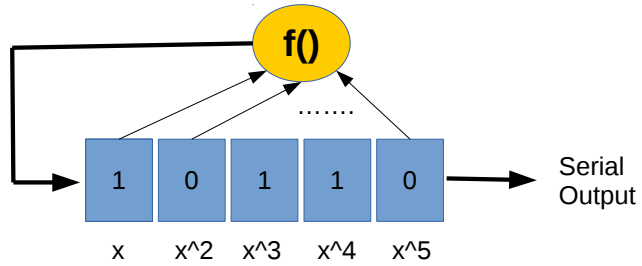
Linear Feedback Shift Register LFSR I

Differently from the others this is a **random bit generator** and not a random integer or float generator. The theory of this sequence generator has its roots in error correcting codes and cryptography (in particular streaming ciphers). It was devised thinking about an easy hardware implemention of it so that it can be very fast and efficient.
Golomb [12] is the standard reference for this generator.
It can easily be implemented in hardware as a sequence of flip-flops that at every clock push their content to the element on the right. An input is provided by a feedback connection based on a linear function (usually an XOR that on $\mathbb{F}_2$ is the same as an add operation) on some bits of the register (the rightmost bit should be used , otherwise the LFSR is called singular and is not of interest).

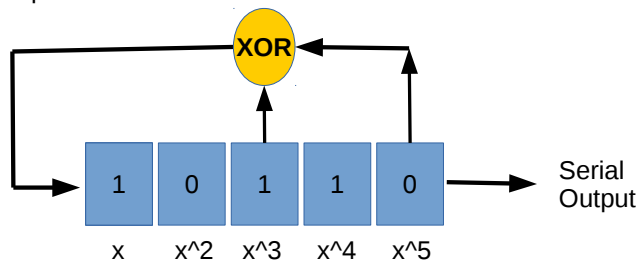
Linear Feedback Shift Register LFSR II

Shift Register

Linear Feedback Shift Register LFSR



A common function used for the **feedback** is the adddition in the finite field F2 (= XOR) between some of the bits in the register (called *taps).*
The LFSR is connected with a polynomial in F2 with all the powers of x xored for the feedback. For instance



$$P(x)=1+x^3+x^5, LFSR(5,1+x^3+x^5)$$

How you indicate a LFSR ?
With LFSR(L,poly). Eg.
$LFSR(4, 1 + x + x^4)$

Theorem : Let $P(x)$ be a connection polynomial of degree L over $\mathbb{F}_2[x]$:

- If $P(x)$ is *irreducible* over $\mathbb{F}_2$ then for each nonzero seed produces an output sequence with period the least N such that $P(x)$ divides $1 + x^N$.
- If $P(x)$ is a *primitive polynomial* then each seed produces an output sequence of maximal length $2^L - 1$.

Linear Feedback Shift Register LFSR III

```
/* C loop for a LFSR */

unsigned istate = 0xffffffffu;
unsigned lfsr = istate;
unsigned lsb;
/* x^32+x^7+x^5+x^3+x^2+x+1 is a primitive polynomial over
    GF(2) */
unsigned mask = 0xEA000001u; /* taps positions */
unsigned pc; /* population count */

 do {
      lsb = lfsr & 1;
      lfsr >>= 1;                    /* Shift register */
      pc = __builtin_popcount(lfsr & mask);
      if ( pc & 1) {   /* the same as a parity bit on the
          taps*/
              lfsr |= 0x80000000u;
      }
 } while (1);
```

If the hw has no support for population count :

Linear Feedback Shift Register LFSR IV

```
/* if the hw has no support for popcnt op */
const int popcnt8[256] = {
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8
};

return  popcnt8[ (b32 >> 24) & 0xFF ] +
        popcnt8[ (b32 >> 16) & 0xFF ] +
        popcnt8[ (b32 >>  8) & 0xFF ] +
        popcnt8[  b32        & 0xFF ];
```

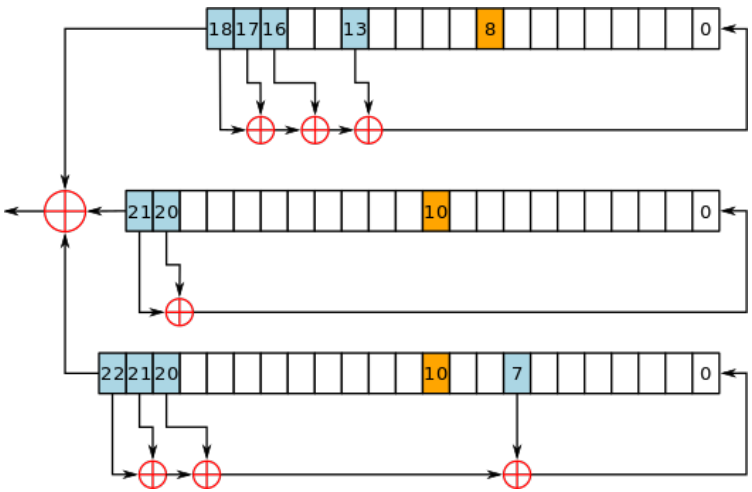
Primitive trinomials over $\mathbb{F}_2$ I

m	k	m	k	m	k
2	1	93	2	193	15
3	1	94	21	194	87
4	1	95	11	196	3
5	2	97	6	198	9
6	1	98	11	199	34
7	1	100	15	201	14
9	1	102	29	202	55
10	3	103	9	204	27
11	2	105	4	207	43
12	3	106	15	209	6
14	5	108	17	210	7
15	1	110	33	212	105
17	3	111	10	214	73
18	3	113	9	215	23
20	3	118	33	217	45
21	2	119	8	218	11
22	1	121	18	220	7
23	5	123	2	223	33
25	3	124	19	225	32
28	1	126	21	228	113
29	2	127	1	231	26
30	1	129	5	233	74
31	3	130	3	234	31
33	10	132	17	236	5
34	7	134	57	238	73
35	2	135	11	239	36
36	9	137	21	241	70
39	4	140	15	242	95
41	3	142	21	244	111
42	7	145	52	247	82
44	5	146	71	249	35
46	1	147	14	250	103
47	5	148	27	252	15
49	9	150	53	253	46
52	3	151	3	255	52
54	9	153	1	257	12
55	7	154	15	258	71
57	4	155	62	260	15
58	19	156	9	263	93
60	1	159	31	265	42
62	29	161	18	266	47
63	1	162	27	268	25
65	18	166	37	270	53
66	3	167	6	271	58
68	9	169	34	273	23
71	6	170	11	274	67
73	25	172	1	276	63
74	35	174	13	278	5
76	21	175	6	279	5
79	9	177	8	281	93
81	4	178	31	282	35
84	5	180	3	284	53
86	21	182	81	286	69
87	13	183	56	287	71
89	38	185	24	289	21
90	27	186	11	292	37
92	21	191	9	294	33

Irreducible/Primitive trinomials $x^m + x^k + 1$ over $\mathbb{F}_2$. Handbook of Applied Cryptography, Menezes ([13])

LFSR are ubiquitous : GSM A5/1 encryption I

GSM uses 3 different LFSR xored between them to encode your phone calls. Every 4.615 ms 114 bits produced by the xored LFSRs are xored with the data and transmitted :



Variable clocking using a *majority voting* scheme.

Two or three of the yellow boxes will present the same bit value, only those will clock the shift register.

LFSR1:$LFSR(19, x^{19} + x^{18} + x^{17} + x^{14} + 1)$

LFSR2:$LFSR(22, x^{22} + x^{21} + 1)$

LFSR3:$LFSR(23, x^{23} + x^{22} + x^{21} + x^{8} + 1)$

In the GPRS/EDGE modes A5/3

(KASUMI=MISTY) is used instead (384 bits at a

time).

RNG based on Linear recurrences on $\mathbb{F}_2$ I

General framework introduced by Niederreiter and then L'Ecuyer. It comprises most of the methods. $\mathbb{F}_2$ is the *finite field* with two elements, 0 and 1. General framework :

- $\mathbf{x}_i = \mathbf{A}\mathbf{x}_{i-1}$ (transition function)
- $\mathbf{y}_i = \mathbf{B}\mathbf{x}_i$ (output function)
- $u_i = \sum y_{i,l-1} 2^{-l} = .y_{i,0} y_{i,1} y_{i,2} \dots$

where $\mathbf{x_i} = (x_{i,0}, x_{i,1}, \dots, x_{i,k-1})^T \in \mathbb{F}_2^k$ and $\mathbf{y_i} = (y_{i,0}, y_{i,1}, \dots, y_{i,w-1})^T \in \mathbb{F}_2^w$. $\mathbf{A}$ is a $k \times k$ *transformation matrix* and $\mathbf{B}$ a $w \times k$ *output matrix*.
The *characteristic polynomial* of the matrix $\mathbf{A}$ can be written:

$$P(z) = det(\mathbf{A} - z\mathbf{I}) = z^k - a_1 z^{k-1} - a_2 z^{k-2} - \dots - a_{k-1} z - a_k$$

Both x_i and y_i obey the same recurrence

$$x_{i,j} = a_1 x_{i-1,j} + \dots + a_k x_{i-k,j} mod 2$$

RNG based on Linear recurrences on $\mathbb{F}_2$ II

The period of this sequence is full $= 2^k - 1$ if $P(z)$ is a *primitive polynomial* over $\mathbb{F}_2$.
More the matrices are sparse, more efficient is the computation to get a rn, but this against the fact that if the matrix don't mix enough the state, the resultant generator will have poor statistical qualities.

Tausworthe generator (1965) I

You can get random integers or floats from an LFSR random bit generator:
run an LFSR generator l times and get l bits from it and consider them as the binary fraction of a float in $(0, 1)$ or an integer $[0..(2^l - 1)]$.
$x = (b_0, \ldots, b_{l-1})^T$ produced by a LFSR (usually based on a trinomial)

$$b_i = b_{i-p} \oplus b_{i-p+q}$$

are taken to represent the fraction of a float or an integer with l bits (**l-wise decimation** of the sequence of b_i). If l is relatively prime with $2^p - 1$ (the period of the LFSR), also the period of the *l-tuples* will be $2^p - 1$. The blocks x_i satisfy the same recurrence of the sequence of bits b_i :

$$x_i = x_{i-p} \oplus x_{i-p+q}$$

Tausworthe generator (1965) II

and are connected with the trinomial

$$x^p + x^r + 1$$

where $r = p - q$. The initial state is a sequence of p bits.

Tausworthe (1965)

An RNG built on this is sometimes indicated as $\mathbf{R}(\mathbf{r}, \mathbf{p})$

$$x_n = 0.b_{nl}b_{nl+1}b_{nl+2}\ldots b_{nl+(l-1)}$$

$$U_n = \sum_{j=0}^{l-1} b_{nl+j}2^{-j-1}$$

This method is, of course, inefficient.

Generalized feedback shift register GFSR, (Lewis and Payne 1973) I

A better way is to use the LFSR in parallel over a word of w bits. In this case the seed is made of p words. It generates random integers in $0..(2^w - 1)$. ([31])

GFSR (Lewis, Payne) :

$$x_n = x_{n-p} \oplus x_{n-p+q}$$

It has period $(2^p - 1)$ when p, q are chosen according to a primitive polynomial as we have seen for the LFSR. In fact this is the implementation of the same LFSR for each bit of the computer word. But the initial values (p integers) should be chosen carefully to avoid pitfalls. The original generator of Lewis and Payne was quite slow to initialize the n seeds because to be sure that the matrix of the rows of the $n \times p$ words had no linear dependence it

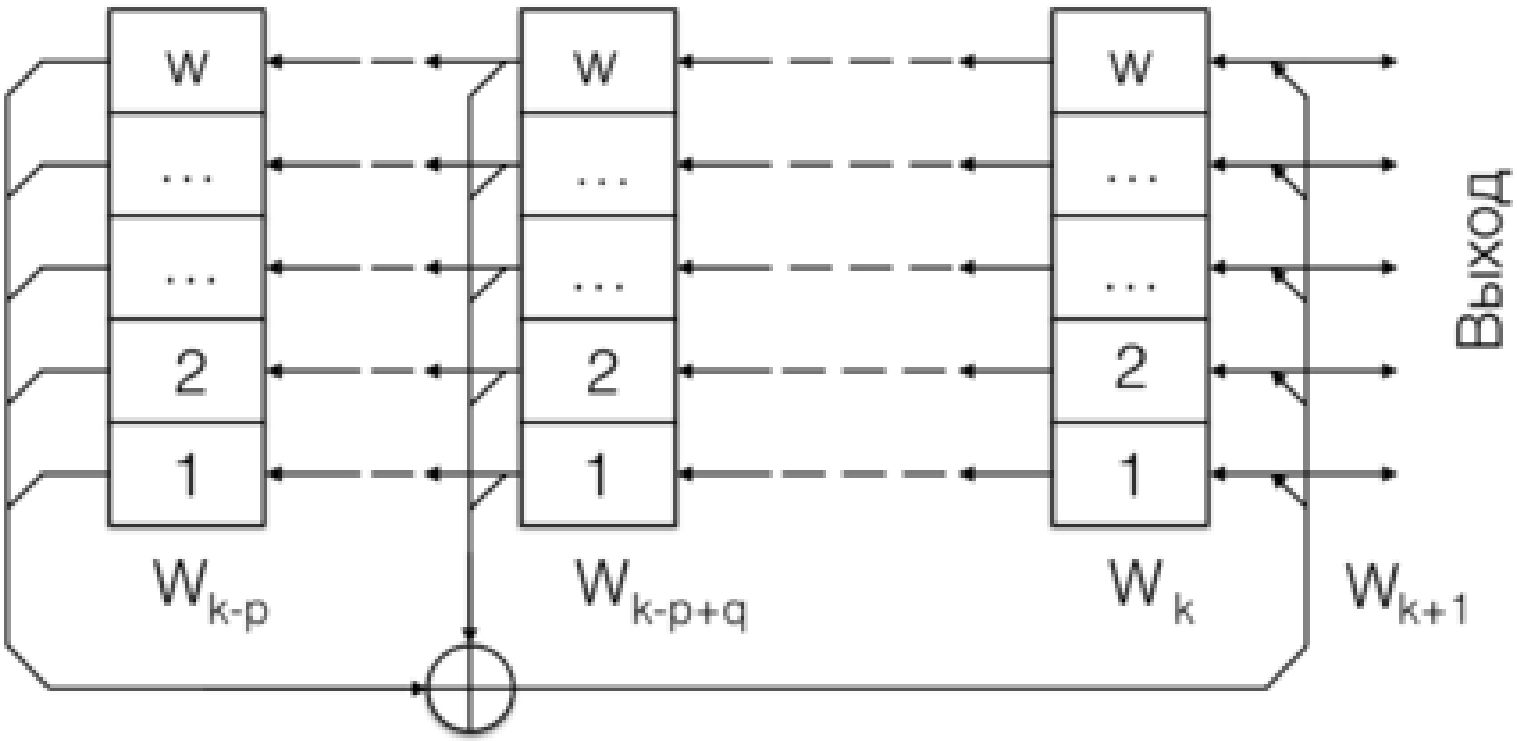
Generalized feedback shift register GFSR, (Lewis and Payne 1973) II

was initializing each of the row with the same LFSR delayed for each row. Then Arvillias and Maritsas and later Collings and Tezuka made the initialization less problematic. GFSR based on trinomials suffer of a gross flaw discovered by Schmid [32] in 1995. The generator proposed by Lewis and Payne is based on the polynomial :

$$x^{98} + x^{27} + 1$$

with a delay of 9000 bits between each column initialization.
See next picture.

Generalized feedback shift register GFSR, (Lewis and Payne 1973) III



S.Kirckpatrick-Stoll R250 (1981) [34]

It is famous because one of the authors is the physicist of the simulated annealing. It is a GFSR initialized by an LCG :

$$X_n = X_{n-103} \oplus X_{n-250}$$

Still present in libraries, used for a long time by many physicists. Flaws found by Ferrenberg [25] and Schmid [32].

Characteristic polynomial of a matrix **A** I

$$\mathbf{A}.\mathbf{x} = \lambda\mathbf{x}$$

In the eigenvalues/eigenvectors problems you have to find the set of couples $(\lambda_i, \mathbf{x_i})$ such that the **action** of the array on them preserves the direction of the vectors (it is just a dilation).

$$\lambda\mathbf{x} - \mathbf{A}.\mathbf{x} = \lambda\mathbf{I}\mathbf{x} - \mathbf{A}\mathbf{x} = (\lambda\mathbf{I} - \mathbf{A}).\mathbf{x} = \mathbf{0}$$

Then in principle to solve the problem you just need to find when the matrix $(\lambda\mathbf{I} - \mathbf{A})$ is singular. Only in that case the product can be 0 with $\mathbf{x}$ different from the zero vector. This matrix is of the form :

$$\begin{bmatrix} \lambda - a_{11} & a_{12} & \dots & \\ a_{21} & \lambda - a_{22} & a_{23} & \dots \\ \dots & & & \\ \dots & & & \lambda - a_{n1} \end{bmatrix}$$

Characteristic polynomial of a matrix **A** II

The matrix is singular when its determinant is $= 0$. You develop the determinant with e.g Laplace expansion and you get :

$$det\left(\lambda \mathbf{I} - \mathbf{A}\right) = \lambda^n + c_{n-1}\lambda^{n-1} + \ldots + c_0$$

In finite fields the characteristic polynomial of a matrix is important because if it is **primitive** then many nice results follow.

In statistics vectors are row vectors and they multiply a matrix from the left

In mathematics and physics a vector is a column vector and you multiply a vector by a matrix this way $\mathbf{A} \cdot \mathbf{v}$. In statistics, vectors are row vectors and you multiply a matrix this way $(\mathbf{A} \cdot \mathbf{v})^{\mathbf{T}} = \mathbf{v}^{\mathbf{T}} \cdot \mathbf{A}^{\mathbf{T}}$:

$$\begin{bmatrix} x0 & x1 & x2 & x3 \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & & \\ a_{14} & a_{24} & a_{34} & \end{bmatrix}$$

I'm pointing out this because if you read one of the many papers on RNG written by statisticians maybe you would stare at this.

Makoto Matsumoto

- born 1965, professor at the dept of Mathematics University of Hiroshima, Japan
- together with Kurita developer of the first **Twisted GFSR : T800** , 1992 and Tempered Twisted GFSR **TT800**, 1994
- together with Nishimura developed the **Mersenne Twister (MT19937)** , 1998 (*probably today the most used rng*)
- with Saito **MTGP**, a variant of the Mersenne Twister for GPUs, 2009,
- with Saito **TinyMT** , a small size $(2^{128} - 1)$Mersenne Twister, , 2011
- together with Saito developed SIMD-oriented Fast Mersenne Twister (**SFMT**) , 2013

- **WELL**, F. Panneton, P. L'Ecuyer and M. Matsumoto, "Improved Long-Period Generators Based on Linear Recurrences Modulo 2"

Twisted generalized feedback shift register TGFSR I

Matsumoto,Kurita [20], [21]
Pros of GFSR : fast generation of rn, sequence has arbitrarily long period, implementation does not depend on word size.
Cons : selection of seeds its critical and good initialization is time consuming, period $2^n - 1$ is quite smaller than the storage area would allow.

$$\mathbf{x}_{l+n} = \mathbf{x}_{l+m} \oplus \mathbf{x}_l A \ , \ (l = 0, 1, \ldots)$$

where A is a $w \times w$ matrix with $0, 1$ components. With suitable choices of n, m, A the TGFSR generator attains the maximal period of $2^{nw} - 1$. Because it has maximal period it is $n - \textit{equidistributed}$. The trick is simply to update $\mathbf{x}_l$ with a twist :

$$A = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_0 & a_1 & a_2 & \ldots & a_{w-1} \end{bmatrix}$$

Twisted generalized feedback shift register TGFSR II

$$\mathbf{x}_l = (x_{l+m \pmod n}) \oplus shiftright(\mathbf{x}_l) \oplus (0 \text{ if } LSB(\mathbf{x}_l = 0) \text{ else } \mathbf{a})$$

Here Matsumoto and Kurita considered not just the 3 words of the recurrence, but all the nw bits of the state.

$$(\mathbf{x}_{l+n}, \mathbf{x}_{l+n-1}, \ldots, \mathbf{x}_{l+1}) = (\mathbf{x}_{l+n-1}, \mathbf{x}_{l+n-2}, \ldots, \mathbf{x}_l) \cdot B$$

According to the general theory developed by Niederreiter [6], this generator has period $2^{nw} - 1$ iff the characteristic polynomial $\varphi_B(x)$ of B is primitive.
In this case the nw dimensional vector runs over all the 2^{nw} vectors except zero. Matsumoto and Kurita showed that $charpoly_B(t) = charpoly_A(t^n + t^m)$ and therefore :
Theorem: Let $\varphi_A(x)$ be the charpoly of A, the period of the sequence generated by the TGFSR is $2^{nw} - 1$ iff $\varphi_A(t^n + t^m)$ is primitive.

Twisted generalized feedback shift register TGFSR III

The parameters of T400 are $w = 16, n = 25, m = 11$, $\mathbf{a} = \mathbf{0xA875}$. The parameters of T800 are $w = 32, n = 25, m = 7$, $\mathbf{a} = \mathbf{0x8EBFD028}$.

Equidistribution

We have seen that some generators have problems to be used in multidimensional contexts (e.g. LCGs). They miss many points in space because they can't repeat an output without incurring a loop. The TGFSRs for the way in which they are built instead assure us that they will present all different sequences of $n \cdot 2^w$ bit integers except the one that is all 0. So we say that the TGFSR T800 is 25-equidistributed (up to 25 dimensions). If we take in succession 25 integer from T800 as a 25-tuple and we put it in a 25 dimensional space, they fill all the points with integer coordinates.

Tempering I

The first TGFSR proposed was weak on some statistical tests despite the solid theoretical foundations on which it was laying. So, after half a year, Matsumoto and Kurita proposed a modification that later was used also by many other such generators. In particular even if it was assured genetically the 25-equidistribution it was weak for k-distributions over the order of the recurrence. The TGFSRs in which tempering was applied are prefixed with a T : TT400, TT800. The parameters for TT800 are : $w = 32, n = i25, m = 7, \mathbf{a} = 0x8EBFD028, s = 7, b = 0x2B5B2500, t = 15, c = 0xDB8B0000$.
k-distributed to v-bit accuracy means that the kv-tuples :

$$(trunc_v(x_i), trunc_v(x_{i+1}), \ldots, trunc_v(x_{i+k-1}))$$

appear the same number of times in the sequences. Tezuka pointed out that T800 is n-distributed($n = 25$) only to 2-bit

Tempering II

accuracy instead of a possible $nw/2 = 800/2 = 400$. In fact they found that for a TGFSR the bound is $n\lfloor w/v \rfloor$. They found that with an appropriate choice of $s, t, \mathbf{b}, \mathbf{c}$ the following shuffling of the bits of $\mathbf{x}$ could attain almost this bound for most of the TGFSR they checked :

$$\mathbf{y} = \mathbf{x} \oplus ((\mathbf{x} << s)\&\mathbf{b})$$

$$\mathbf{z} = \mathbf{y} \oplus ((\mathbf{y} << t)\&\mathbf{c})$$

In the TT800 these parameters are :
$s = 7, t = 15, \mathbf{b} = 0x2B5B2500, \mathbf{c} = 0xDB8B0000$ and the order of equidistribution to 2-bit becomes 400, to $3 - bits250$ and to $4 - bits$ it is 200.

Multiple Recursive Matrix Method (MRMM) (Niederreiter 1993,1995) I

Let p be a prime, $\mathbf{A}_i$ be $w \times w$ bits matrices and $\mathbf{z}_i$ be w bits row vectors(words) over the field $\mathbb{Z}_p$:

$$\mathbf{z}_{n+k} = \sum_{h=0}^{k-1} \mathbf{z}_{n+h}.\mathbf{A}_h = \mathbf{z}_n.\mathbf{A}_0 + \mathbf{z}_{n+1}.\mathbf{A}_1 + \ldots \mathbf{z}_{n+k-1}.\mathbf{A}_{k-1}$$

Theorem : The period of the sequence $\mathbf{z}_i$ over $\mathbb{Z}_p^w$ is $p^{kw} - 1$ iff the polynomial

$$det\left(x^k \mathbf{I}_w - \sum_{h=0}^{k-1} x^h \mathbf{A}_h\right)$$

of degree kw is a primitive polynomial over $\mathbb{Z}_p$.

Multiple Recursive Matrix Method (MRMM) (Niederreiter 1993,1995) II

This method generalizes many other methods :

- if $k = w = 1$ this is simply a **MLCG** $z_{n+1} = z_n a_0 \mod p$
- $k \geq 2, w = 1$ **Multiplicative Recursive Generator(MRG)** $z_{n+1} = z_n a_0 + z_{n+1} a_1 + \cdots + z_{n+k-1} a_{k-1} \mod p$
- $k \geq 2, w \geq 2$ $A_i = a_i I_w$ **GFSR**
  $p = 2, w = 32, k = 98, q = 27$ Lewis,Payne based on primitive polynomial $x^{98} + x^{27} + 1$
  $\mathbf{z}_n = \mathbf{z}_{n-98} + \mathbf{z}_{n-27} \mod 2^{32}$, where $\mathbf{z}_i$ here are 32 bits integers(period 2^{98})
- $k \geq 2, w \geq 2$ A_0 whatever, $A_1 \ldots A_{k_1} = a_i I_w$ **TGFSR**
  $p = 2, w = 32, k = 25, q = 11$ Matsumoto, Kurita
  $\mathbf{x}^{400} + \cdots + 1$

Mersenne Twister MT (1998) I

The Mersenne Twister is a TGFSR like the Mastumoto, Kurita T800, TT800. Here the 2 advances of Makumoto and Nishimura are :

- they used an incomplete array of words $p = nw - r$, so that the number of bits can be a Mersenne exponent $(2^p - 1)$ and they don't need to factor it because it is prime (also today we can't factor numbers large as 2^{2000}). In a normal TGFSR instead nw can never be a prime.
- they invented a new way to test for primitivity in this case : the *inversive-decimation method*. Anyway it took them 2 weeks to find a primitive polynomial for MT19937.

Mersenne Twister MT (1998) II

Many different versions exist. Mostly used is the revised MT19937 in C : `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`. Developed by Matsumoto and Nishimura (1998), period $2^{19937} - 1$.

MT (linear recursion over $\mathbb{F}_2$) :

$$x_{k+n} = x_{k+m} \oplus ((x_k^u \mid\mid x_{k+l}^l)A)$$

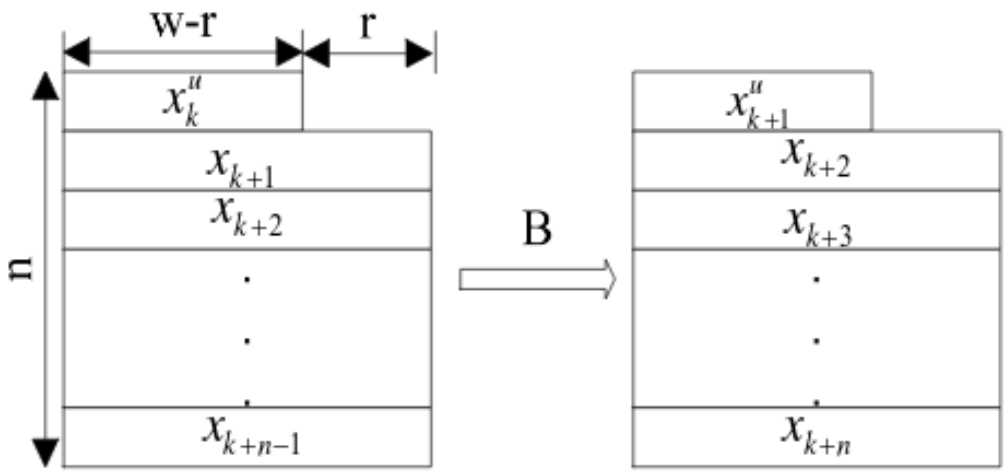
- n is the degree of recursion
- A is a $w \times w$ matrix chosen to make simple the multiplication

$$\begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & a_{w-4} & \dots & a_0 \end{bmatrix}$$

Mersenne Twister MT (1998) III

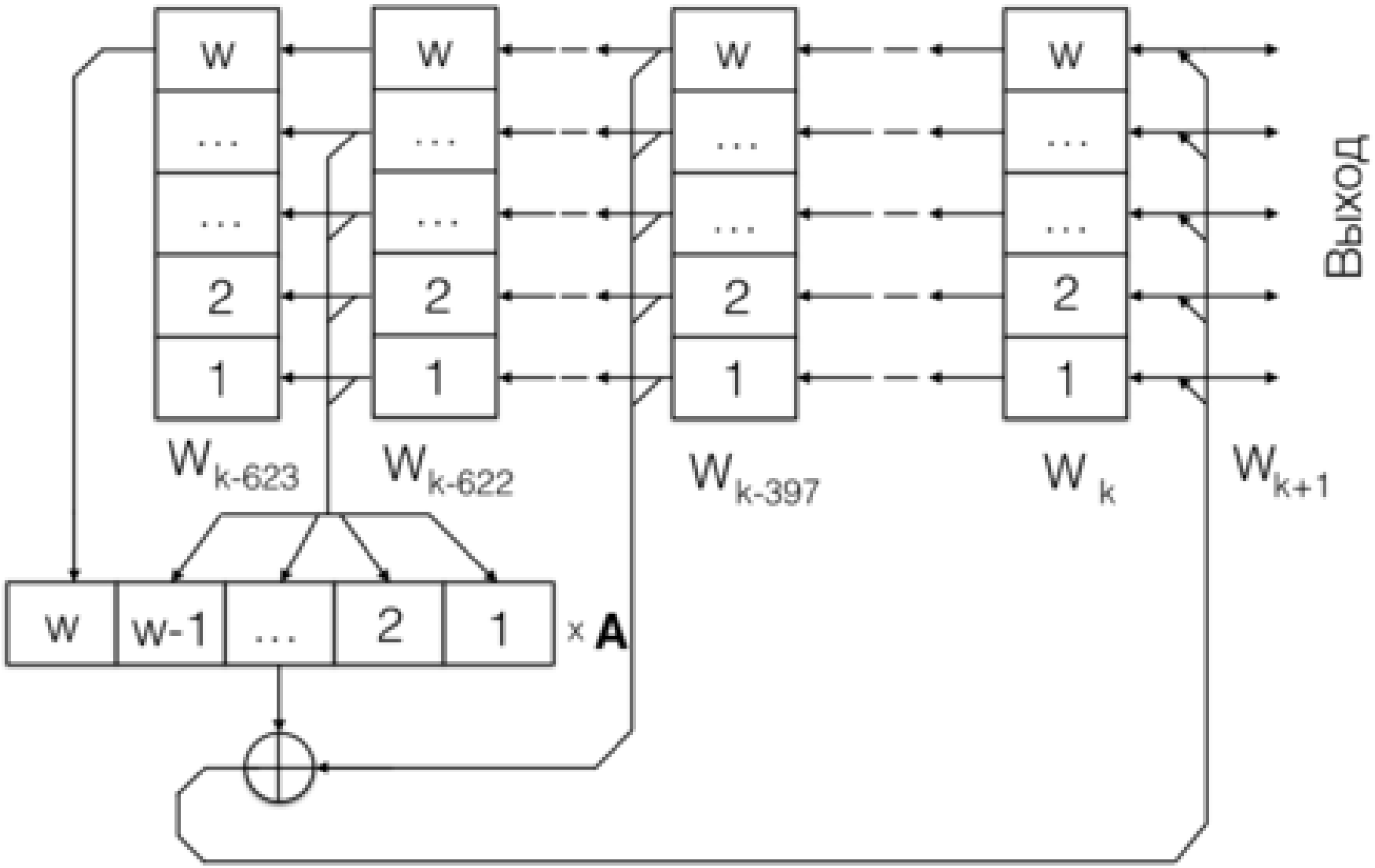
- k is $0, 1, 2, \ldots$
- $x_0, x_1 \ldots, x_{n-1}$ are initial seeds
- m is a middle index $1 \leq m \leq n$
- x^l_{k+1} are lower or rightmost r bits of x_{k+1}
- x^u_k are upper or leftmost $w - r$ bits of x_k
- $\oplus$ is bitwise XOR

Illustration of state transition (from Jagannatam):



Parameter	Value
n	624
w	32
r	31
m	397

Mersenne Twister MT (1998) IV



Mersenne Twister MT (1998) V

The MT19937 came with *batteries included* , meaning that it was already tempered similar to the TT800 :

$$\mathbf{y} = \mathbf{x} \oplus (\mathbf{x} >> u)$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << s)\ AND\ \mathbf{b})$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} << t)\ AND\ \mathbf{c})$$

$$z = \mathbf{y} \oplus (\mathbf{y} >> l)$$

$$\mathbf{a} = 0x9908B0DF, u = 11, s = 7, t = 15, \mathbf{b} = 0x9D2C5680$$

$$\mathbf{c} = EFC60000, l = 18$$

MT is a Multiple Recursive Matrix Method of Niederreiter [7] :

$$\mathbf{x}_{k+n} = \mathbf{x}_{k+m} + \mathbf{x}_{k+1} \begin{bmatrix} 0 & 0 \\ 0 & I_r \end{bmatrix} \mathbf{A} + \mathbf{x}_k \begin{bmatrix} I_{w-r} & 0 \\ 0 & 0 \end{bmatrix} \mathbf{A}$$

Mersenne Twister MT (1998) VI

A critic moved to MT is that it doesn't mix enough the state and few bits set or many of the bits set, require many million of cycles to move away from these kind of states.
This is due to the small number of terms in the associated polynomial (135 out of 19937).
WELL [26] for instance, uses almost half of the terms.

Mersenne twister : MT variants I

TinyMT (127 bits of state), 2 versions tinymt32 and tinymt64, first outputs 32 bits unsigned integers or single floats, second 64 bits unsigned integers or double precision floats, period only 2^{127-1} but very small state. State initialized by the function **TinyMTDC**. Authors: M.Saito,M.Matsumoto

MTGP MT for Graphic Processors. (Saito, Matsumoto) 32 or 64 bit integers as output or single precision or double precision floats. For the 32 bit periods of $2^{11213} - 1, 2^{23209} - 1, 2^{44497} - 1$. Cuda and OpenCL code. To provide the many parameters used there is and MTGP Dynamic Creator routine (**MTGPDC**.

SFMT SIMD-oriented Fast MT.(2006 Saito, Matsumoto) is a LFSR that generates 128 bit RN integer at each step. It uses 128 bit integers supported on modern CPU and SIMD. Source in standard C, SSE@ instructions + C, AltiVec instructions + C. Periods $2^{607} - 1, 2^{216091} - 1$.

Panneton, L'Ecuyer, Matsumoto 2006 : **WELL**

Well Equidistributed Long-period Linear , 2006
It is based on linear recurrences modulo 2 over $\mathbb{F}_2$
Panneton, F. O.; l'Ecuyer, P.; Matsumoto, M. (2006).
"Improved long-period generators based on linear recurrences modulo 2" (PDF).
ACM Transactions on Mathematical Software
`http://www.iro.umontreal.ca/~panneton/WELLRNG.html`
512, 1024, 19937, 44497 bits implementations readily available.
Their implementations makes a TGFSR become ME (Maximally Equidistributed) adding at the output a Matsumoto-Kurita *tempering* of the output (**TTGFSR** : Tempered Twisted GFSR).
Period of WELL19937 is $2^{19937} - 1$ or can be even 2^{44497}.

Parallel random numbers I

From a rn generator $\mathbf{x} = (x_n)$, we can easily obtain parallel streams :

- partitioning in **j lagged subsequences** :
  $\omega_k^{(j)} = (x_{kn+j})_{m \geq 0}$, $k \geq 2$, $0 \leq j < k$ (aka *leapfrog*)
- partitioning in **consecutive blocks** of length l :
  $\psi_l^{(k)} = (x_{kl+n})_{n=0}^{l-1}$, $k \geq 0$ (aka *sequence splitting*)
- previous methods are good if the generator allows easily to *skip-ahead* like LCGs. It is impractical for LFSR generators. For these you run the same LFSR with different initial seeds on every processor. The initialization of the seeds tables can be done with an LCG for instance.

SPRNG : Scalable Parallel Random Number Generator library I

- Developed at Florida State University (current version in C++/Fortran is 5.0) from 1999 to today by M.Mascagni et al.
- Download it from *http://www.sprng.org/Version5.0/sprng5.tar.bz2* and

```
tar xjf sprng5.tar.bz2
cd sprng5
./configure
make
cd check
./checksprng
./timesprng
```

- Based on the 5 generators :
  1. Combined Multiple Recursive Generator (MRG)
     - $z_n = x_n + y_n * 232 \mod 264$
     - $y_n = 107374182 * y_{n-1} + 104480 * y_{n-5} \mod 2147483647$
     - and x is the sequence produced by the 64 bit LCG.

SPRNG : Scalable Parallel Random Number Generator library II

2. 48 bit LCG with Prime Addend
   $x_n = a x_{n-1} + p \mod M \quad , \quad M = 2^{48}$
3. 64 bit LCG with Prime Addend
4. Modified Lagged Fibonacci Generator
   $z_n = x_n \; XOR \; y_n$
   where XOR is the exclusive-or operator and x and y are sequences obtained from Lagged Fibonacci sequences of the following form:
   - $x_n = x_{n-k} + x_{n-l} \mod M$
   - $y_n = y_{n-k} + y_{n-l} \mod M$
5. Multiplicative Lagged Fibonacci Generator
   $x_n = x_{n-k} * x_{n-l} \mod M$
6. Prime Modulus LCG
   $x_n = a * x_{n-1} \mod (2^{61} - 1)$ where the multiplier a differs for each stream

- **How to use it ?**
  - In a serial program :

SPRNG : Scalable Parallel Random Number Generator library III

1. define the macro `SPRNG_DEFAULT` to use the simple interface. In C `#define SPRNG_DEFAULT 0`
2. C users should include `sprng_cpp.h`, Fortran users `sprng_f.h`
3. If the user wants he can call an initialization function `init_sprng`. This function has 4 parameters: stream number, total number of generators, seed, multiplier.
4. calling now `sprng()` will provide a double precision number in $(0, 1)$

- In a parallel program :
  1. define the macro `SPRNG_DEFAULT` to use the simple interface `#define SPRNG_DEFAULT 0`
  2. define the macro `USE_MPI` to instruct the generator to use MPI during initialization
  3. C users should include `sprng_cpp.h`, Fortran users `sprng_f.h`
  4. before calling any `sprng` function the user should call `MPI_Init`

Salmon,etc. Shaw Research(Anton) : Parallel Random Numbers: As easy as 1,2,3 I

Counter based random numbers.

$$x_n = b_k(n)$$

Inherently parallel because there is no dependence between successive x_n in the sequence. b_k is a keyed bijection with key k. They start from well known cryptographic ciperhs that implement a keyed bijection : *AES* , *Threefish*. These are too slow for simulations. Then they try to change the algorithms reducing the cryptographic strength and complexity. What comes out they called: *ARS*, *Threefry* and *Philox* and they report very good performance for them.

Primitive polynomials of Mersenne prime order I

j	m	k (k_1, k_2, k_3)
1	2	1
2	3	1
3	5	2
4	7	1, 3
5	13	none (4,3,1)
6	17	3, 5, 6
7	19	none (5,2,1)
8	31	3, 6, 7, 13
9	61	none (43,26,14)
10	89	38
11	107	none (82,57,31)
12	127	1, 7, 15, 30, 63
13	521	32, 48, 158, 168
14	607	105, 147, 273
15	1279	216, 418
16	2203	none (1656,1197,585)
17	2281	715, 915, 1029
18	3217	67, 576
19	4253	none (3297,2254,1093)
20	4423	271, 369, 370, 649, 1393, 1419, 2098
21	9689	84, 471, 1836, 2444, 4187
22	9941	none (7449,4964,2475)
23	11213	none (8218,6181,2304)
24	19937	881, 7083, 9842
25	21701	none (15986,11393,5073)
26	23209	1530, 6619, 9739
27	44497	8575, 21034

Primitive polynomials over $\mathbb{F}_2$ of degree m where $2^m - 1$ is a mersenne prime. Trinomials/pentanomials such that $x^m + x^k + 1$ is irreducible over $\mathbb{F}_2$.

xorshift (Marsaglia 2003) I

The *xorshift* operation is to replace a $w - bit$ word with the result of the *xor* of the word itself with a shifted a places left or right version of it ($0 < a < w$). In C :

`x = x^(x<<a);` or `x = x^(x>>a);`

Xorshifts are linear operations. The matrix with all ones on the superdiagonal produces a left shift and is indicated bt **L** , therfore $I + L^a$ is a left xorshift matrix with a left shifts. AT the same time the matrix **R** with all ones only on the main subdiagonal produces a right shift and $I + R^a$ is a right xorshift matrix with a right shifts. Therefore a xorshift rng is described by :

$$\mathbf{v}_{i+1} = \mathbf{A} \cdot \mathbf{v}_i$$

where **A** is the product of xorshift matrices. It is a particular case of the Multiple Recursive Matrix Method described by Niederreiter in 1995 [8] .

xorshift (Marsaglia 2003) II

Marsaglia's favourite was :

$$A = (I + L^{13})(I + R^{17})(I + L^{5})$$

Period $2^k - 1$ with $k = 32, 64, 96, 128, 160, 192$. Produces integers $\in [0 \ldots 2^{32} - 1]$, by means of the XOR instruction. In C : `y^(y<<a)`, in Fortran : `IEOR(y,ishft(y,a))`. To give an idea of the power of this procedure, given 4 32 bits seeds x, y, z, w the sequence :

```
tmp=(x^(x<<15));
x=y;y=z;z=w;
return w=(w^(w>>21))^(tmp^(tmp>>4));
```

provides $2^{128} - 1$ random 32-bits integers that survive the diehard battery. We have a **seed set** $\in \mathbb{Z}^m$ made up of m-tuples $z = (x_1, x_2, \ldots, x_m)$ and a one-to-one function $f()$ on $\mathbb{Z}^m$. The output of the rng is $f(z), f^2(z), f^3(z), \ldots$.

Numerical Recipes RNG ! !

The authors(Chapter 7 has 100 pages about Random Numbers) claim this to be the ultimate RNG : "this is our suspenders-and-belt, full-body-armor, never any-doubt generator", but this time they don't offer any reward to someone discovering that it is bad. In the first edition of their book both *ran*0() and *ran*1() where flawed.

This to say that the field is mined and even very skilled people can easily stumble.

```
struct Ran
{ Ullong u,v,w;
  Ran(Ullong j):v(4101842887655102017LL),w(1){
    u= j^ v;int64();v=u; int64();w=v;int64();}
  inline Ullong int64()
  { u = u*2862933555777941757LL+7046029254386353O8LL;
    v ^= v >>17; v &= v<<31; v ^= v>>8;
    w = 4294957665U*(w&0xffffffff)+(w>>32);
    Ullong x=u^(u<<21);x ^=x>>35;x ^=x<<4;return (x+v)^w;
  }
```

Numerical Recipes RNG ! II

```
  inline Doub doub(){return 5.42101086242752217E^20*int64()
      ;}
  inline Uint int32(){ return (Uint) int64();}
};
```

P.L' Ecuyer, R.Simard: RngStreams I

An object oriented Random-Number Generator. In C, C++, Java :
http://www.iro.umontreal.ca/~lecuyer/myftp/streams00/
http://statmath.wu.ac.at/software/RngStreams/
To install it :

```
wget http://statmath.wu.ac.at/software/RngStreams/
    rngstreams-1.0.1.tar.gz
tar xz rngstreams-1.0.1.tar.gz
cd rngstreams-1.0.1
./configure --prefix=/usr/local
make
make install
make check
```

GNU Scientific Library (GSL) I

It contains many RNGs. The functions are declared in `gsl_rng.h`.

- `gsl_rng * gsl_rng_alloc (const gsl_rng_type * T)`
  allocates the memory space for a generator of type T. E.g.
  `gsl_rng * r = gsl_rng_alloc (gsl_rng_taus);`
- `void gsl_rng_set (const gsl_rng * r, unsigned long int s)`
  seeds the rng
- `void gsl_rng_free (gsl_rng * r)`
  frees the memory associated with generator r
- `unsigned long int gsl_rng_get (const gsl_rng * r)`
  gets a random integer from generator r
- `double gsl_rng_uniform (const gsl_rng * r)`
  gets a double uniformly distributed in $[0, 1)$
- `unsigned long int gsl_rng_uniform_int (const gsl_rng * r, unsigned long int n)`
  returns an integer between 0 and $n - 1$.

GNU Scientific Library (GSL) II

The library let's you choose different generators and seeds using environment variables: `GSL_RNG_TYPE and GSL_RNG_SEED`

```
#include <stdio.h>
#include <gsl/gsl_rng.h>
gsl_rng * r;  /* global generator */
int main (void)
{
  const gsl_rng_type * T;
  gsl_rng_env_setup();
  T = gsl_rng_default;
  r = gsl_rng_alloc (T);
  printf ("generator type: \%s\n",
      gsl_rng_name (r));
  printf ("seed = \%lu\n",
      gsl_rng_default_seed);
  printf ("first value = \%lu\n",
      gsl_rng_get (r));
  gsl_rng_free (r);
  return 0;
}
```

GNU Scientific Library (GSL) III

and

```
GSL_RNG_TYPE="taus" GSL_RNG_SEED=123 ./a.out
```

Generators/Algorithms :

- `gsl_rng_mt19937`
  mersenne twister
- `gsl_rng_ranlxs0,1,2`
  Luscher Ranlux
- `gsl_rng_cmrg`
  L'Ecuyer CMRG
- `gsl_rng_mrg`
  LEcuyer, Blouin and Coutre
- `gsl_rng_taus`
  Tausworthe generato by L'Ecuyer
- `gsl_rng_gfsr4`
  four taps GFSR (Ziff)

Bad news :
most rng provided by languages and libraries are bad !

These were bad at a certain point in time :

- C-library rand(), random() and drand48()
- Java.util.Random
- standard Perl rand
- Python random()
- Matlab's rand
- Mathematica SWB generator
- ran0() and ran1() in the original *Numerical recipes* book

Good news: good rng can be simple I

KISS99 (proposed by G.Marsaglia) period $\sim 10^{43}$:

```
static unsigned int x = 123456789, y = 362436000,
    z = 521288629, c = 7654321;
unsigned int kiss()
{
    unsigned long long t, a = 698769069ULL;
    x = 69069*x+12345;
    y ^= (y<<13) ; y ^= (y>>17); y ^= (y<<5);
    t = a*z + c; c = (t>>32);
    return x+y+(z=t);
}
```

In *MATLAB*/*Octave* :

Good news: good rng can be simple II

```
% seeds : correct variable types are crucial
A=uint32(12345); B=uint32(65435); Y=12345; Z=uint32(34221);
N=100; % compute N rn
U=zeros(1,N);
for t=1:N
    % 2 MWC generators
    A=36969*bitand(A,uint32(65535))+bitshift(A,-16);
    B=18000*bitand(B,uint32(65535))+bitshift(B,-16);
    % MWC : A and B are low and high 16 bits
    X=bitshift(A,16)+B;
    % CONG : LCG
    Y=mod(69069*Y+1234567,4294967296);
    % SHR3 : 2-shift register generator
    Z=bitxor(Z,bitshift(Z,17));
    Z=bitxor(Z,bitshift(Z,-13));
    Z=bitxor(Z,bitshift(Z,5));
    % combine to form KISS99
    KISS=mod(double(bitxor(X,uint32(Y)))+double(Z)
        ,4294967296);
    U(t)=KISS/4294967296;
end
U(100)
```

Analysis of KISS99 I

A maxima script to show the working of KISS99 :

```
n:32;
u:[b31,b30,b29,b28,b27,b26,b25,b24,b23,b22,b21,b20,
           b19,b18,b17,b16,b15,b14,b13,b12,b11,b10,
           b09,b08,b07,b06,b05,b04,b03,b02,b01,b00];
y:transpose(u);
/* shif left */
shl:genmatrix(lambda([i,j],if i = (j-1) then 1 else 0),n,n);
/* shif right */
shr:genmatrix(lambda([i,j],if i = (j+1) then 1 else 0),n,n);

/* shif right k places */
shrn(k):= shr^^k;
/* shif left k places  */
shln(k):= shl^^k;
/* In GF(2) XOR is the same as + */
y:y + shln(13) . y;
y:y + shrn(17) . y;
y:y + shln(5)  . y;
```

Analysis of KISS99 II

$$
\begin{bmatrix}
b31 + b26 + b18 + b13 \\
b30 + b25 + b17 + b12 \\
b29 + b24 + b16 + b11 \\
b28 + b23 + b15 + b10 \\
b27 + b22 + b14 + b09 \\
b26 + b21 + b13 + b08 \\
b25 + b20 + b12 + b07 \\
b24 + b19 + b11 + b06 \\
b23 + b18 + b10 + b05 \\
b22 + b17 + b09 + b04 \\
b21 + b16 + b08 + b03 \\
b20 + b15 + b07 + b02 \\
b31 + b19 + b18 + b14 + b06 + b01 \\
b30 + b18 + b17 + b13 + b05 + b00 \\
b29 + b17 + b16 + b12 + b04 \\
b28 + b16 + b15 + b11 + b03 \\
b27 + b15 + b14 + b10 + b02 \\
b31 + b26 + b18 + b14 + b13 + b09 + b01 \\
b30 + b25 + b17 + b13 + b12 + b08 + b00 \\
b29 + b24 + b16 + b12 + b11 + b07 \\
b28 + b23 + b15 + b11 + b10 + b06 \\
b27 + b22 + b14 + b10 + b09 + b05 \\
b26 + b21 + b13 + b09 + b08 + b04
\end{bmatrix}
$$

Analysis of KISS99 III

$$
\begin{bmatrix}
b25 + b20 + b12 + b08 + b07 + b03 \\
b24 + b19 + b11 + b07 + b06 + b02 \\
b23 + b18 + b10 + b06 + b05 + b01 \\
b22 + b17 + b09 + b05 + b04 + b00 \\
b21 + b08 + b04 \\
b20 + b07 + b03 \\
b19 + b06 + b02 \\
b18 + b05 + b01 \\
b17 + b04 + b00
\end{bmatrix}
$$

KISS93 and other 6 random number gen by G.Marsaglia I

Main program in C :

```
#include <stdio.h>
int main(void){
int i; UL k;
settable(12345,65435,34221,12345,9983651,95746118);

for(i=1;i<1000001;i++){k=LFIB4;}printf("%u\n",k-1064612766U)
    ;
for(i=1;i<1000001;i++){k=SWB ;}printf("%u\n",k- 627749721U);
for(i=1;i<1000001;i++){k=KISS;}printf("%u\n",k-1372460312U);
for(i=1;i<1000001;i++){k=CONG;}printf("%u\n",k-1529210297U);
for(i=1;i<1000001;i++){k=SHR3;}printf("%u\n",k-2642725982U);
for(i=1;i<1000001;i++){k=MWC ;}printf("%u\n",k- 904977562U);
for(i=1;i<1000001;i++){k=FIB ;}printf("%u\n",k-3519793928U);
}
```

The random number generators using macro definitions :

KISS93 and other 6 random number gen by G.Marsaglia II

```
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC  ((znew<<16)+wnew)
#define SHR3 (jsr^=(jsr<<17), jsr^=(jsr>>13), jsr^=(jsr<<5))
#define CONG (jcong=69069*jcong+1234567)
#define FIB  ((b=a+b),(a=b-a))
#define KISS ((MWC^CONG)+SHR3)
#define LFIB4 (c++,t[c]=t[c]+t[UC(c+58)]+t[UC(c+119)]+t[UC(c
    +178)])
#define SWB  (c++,bro=(x<y),t[c]=(x=t[UC(c+34)])-(y=t[UC(c
    +19)]+bro))
#define UNI  (KISS*2.328306e-10)
#define VNI  ((long) KISS)*4.656613e-10
#define UC   (unsigned char) /*a cast operation*/
typedef unsigned long UL;

/* Global static variables: */
static UL z=362436069,w=521288629,jsr=123456789,jcong
    =380116160;
static UL a=224466889,b=7584631,t[256],x=0,y=0,bro;
static unsigned char c=0;

/* Example procedure to set the table, using KISS: */
```

KISS93 and other 6 random number gen by G.Marsaglia III

```
void settable(UL i1,UL i2,UL i3,UL i4,UL i5, UL i6)
{ int i; z=i1; w=i2; jsr=i3; jcong=i4; a=i5; b=i6;
for(i=0;i<256;i=i+1) t[i]=KISS; }
```

KISS diagram

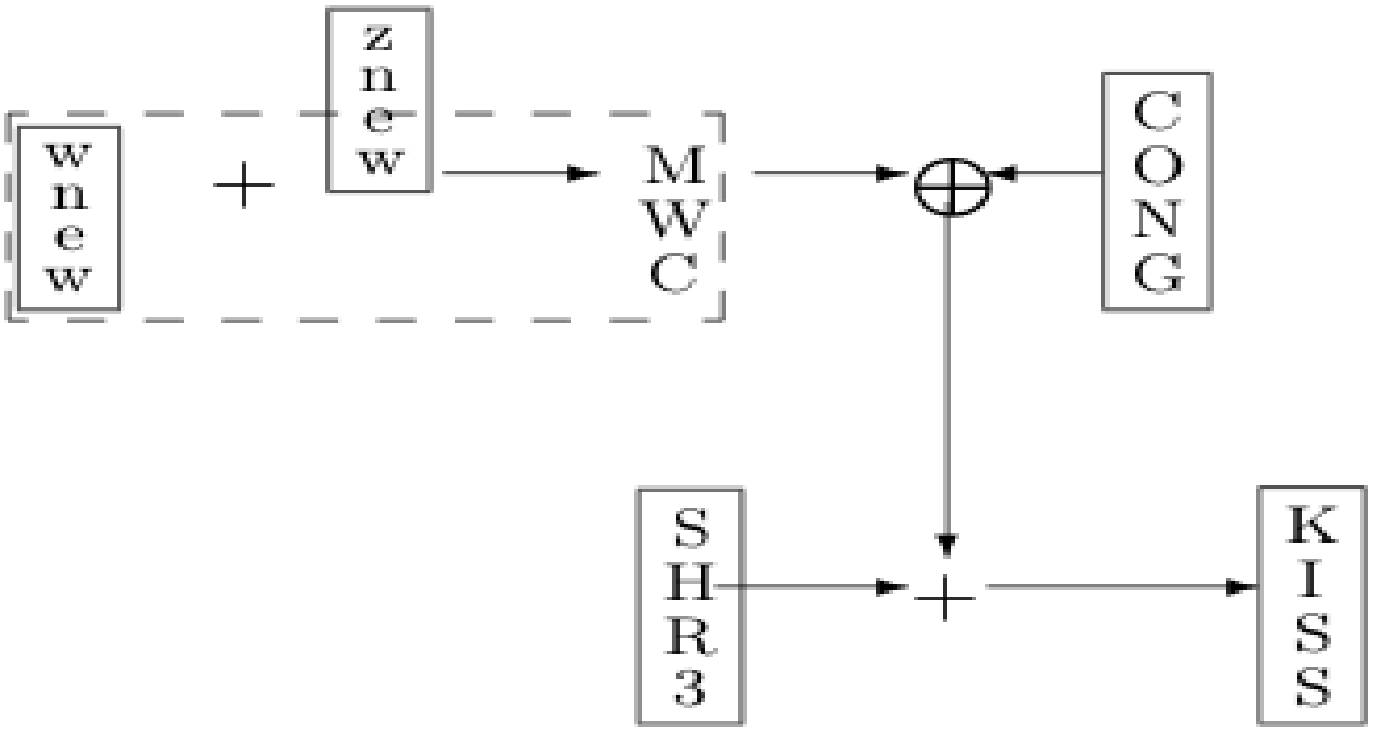


Fig. 2. Diagram of KISS

Statistical Tests I

- First battery of tests described by Knuth in TAOCP in 1969, but no implementation was made available for them
- *diehard* tests by Marsaglia, 1996

  `https://wayback.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/`

  license status of these tests is not clear
- *DieHarder* Robert G.Brown parameterized all diehard tests and put them under GPL. In addition he included some re-programmed STS tests. The declared aim is to add all STS tests and put them under GPL.
- *TestU01* by P.L'Ecuyer and R.Simard, implements most of Knuth tests. Code copyrighted by P.L'Ecuyer (2007).

Statistical Tests II

- *STS* Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptography (National Institute of Standards and Technology), 2001/2010. Consist of 16 tests. These tests were studied for *non-deterministic generators*. In the public domain because made by a gov agency but the situation for the algorithms is not so clear.

DieHarder R.G.Brown I

80 generators (diehard + NIST/STS + RGBrown + Knuth), 31 tests. Revised version : `https://github.com/seehuhn/dieharder` In Ubuntu simply `apt-get install dieharder`.

- `dieharder -g -1` : list internal generators available (80)
- `dieharder -l` : list available tests (31)
- `dieharder -a -g 13` : apply all tests to the generator 13 = mt19937
- produce numbers with `random_write` and then analyze with `cat fn |dieharder -a -g 201`
- `cat /dev/urandom | dieharder -g 200 -a`

The version that is now an Ubuntu package is very user-friendly and it allows to pipe rn to it or simply to give it a file of rn (formatted or not) to digest.
RDieHarder: A DieHarder interface for **R** is now in the R library, developed by Dirk Edelbuttel and Robert G.Brown. Very interesting.
Particular input streams can be provided as raw input (**dieharder -g 200**) on the stdin with a loop like this:

```
#include <stdio.h>
#include <math.h>
#include <unistd.h>
int main(int argc,char* argv[]) {
  unsigned int x = 0x37D1F345; /* 7^5 */
  unsigned int m = 0x3FFFFFFF; /* 2^31 -1 */
  /* park, miller MINSTD */
  while (1) {
        x = (48271 * x) % m;
        write(1,& x, 4); }
}
```

or in a file in raw mode (**dieharder -g 201**) or in ASCII mode (**dieharder -g 202**) using this format :

```
#=============================================
# generator minstd  seed = 3094471044
#=============================================
type: d
count: 1000000
numbit: 32
1015873462
....
```

TestU01 , P.L'Ecuyer and R.Simard I

TestU01 contains over 200 predefined rng for test purposes (*LCG, MRG, combined MRG, lagged-Fibonacci, AWC, SWB, MWC LFSR, combined LFSR, GFSR, twisted GFSR, Mersenne twisters, WELL, . . .*). It divides its tests for $U(0,1)$ variates in 3 batteries (*SmallCrush, Crush[96 tests], BigCrush[106 tests]*) of increasing time and complexity:

e.g. exec time respectively 14 sec*(SmallCrash)*, 1 hour*(Crush)*, 5.5 hours*(BigCrush)*.

Small Crush battery of tests :

1. `smrsa_BirthdaySpacings`
2. `sknuth_Collision`
3. `sknuth_Gap`
4. `sknuth_SimpPoker`
5. `sknuth_CouponCollector`
6. `sknuth_MaxOft`
7. `svaria_WeightDistrib`
8. `smarsa_MatrixRank`
9. `sstring_HammingIndep`
10. `swalk_RandomWalk1`

It's not so friendly to use as *dieharder*. It requires you to write some code. E.g. :

```
#include "ulcg.h"
#include "unif01.h"
#include "bbattery.h"

int main (void)
{
unif01_Gen *gen;
gen = ulcg_CreateLCG (2147483647, 16807, 0, 12345);
bbattery_SmallCrush (gen);
ulcg_DeleteGen (gen);
return 0;
}
```

generates an LCG and submits it to the *SmallCrush* battery of tests. To compile and link it :

```
gcc bat1.c -o bat1 -ltestu01 -lprobdist -lmylib -lm
```

NIST Statistical Test Suite - tests I

1. The Frequency (Monobit) Test,
2. Frequency Test within a Block,
3. The Runs Test,
4. Tests for the Longest-Run-of-Ones in a Block,
5. The Binary Matrix Rank Test,
6. The Discrete Fourier Transform (Spectral) Test,
7. The Non-overlapping Template Matching Test,
8. The Overlapping Template Matching Test,
9. Maurer's "Universal Statistical" Test,
10. The Linear Complexity Test,
11. The Serial Test,
12. The Approximate Entropy Test,
13. The Cumulative Sums (Cusums) Test,
14. The Random Excursions Test, and
15. The Random Excursions Variant Test.

`http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`

Usage : `./assess <length_of_rn>`

Principal tests I

LCG tend to fail run tests, Fibonacci fail the birthday spacing tests, and LFSR fail the matrix rank test.

- Marsaglia - Birthday Spacing.
  m birthdays are extracted in year of n days. They are ordered and distances between adjacents are computed. The distances repetitions should have a Poisson distribution with $\lambda = m^3/(4n)$
- Knuth - Collision Test
  Balls are thrown into urns. A collision occurs when a ball falls in an urn that is already occupied. (Chistiansen 1975,Knuth 1997)
- Knuth - Poker test

Principal tests II

- Marsaglia - Matrix Rank
  A random binary matrix Lxk is created generating [k/s] rn and taking s bits from each. The test then computes the rank of the matrix. This is done for many matrices and the empirical result is compared with the theoretical formulas for the rank of a stochastic matrix. $P[R = 0] = 2^{-Lk}, P[R = x] = \ldots$
- Coveyou - Spectral tests
- Frequency test
  Check the number of zeros in a sequence of random bits (Knuth 1997)
- Knuth - Run Tests
  count the number and length of sequences of non-decreasing numbers or those of non-increasing numbers (*runs up* and *runs down*).

Principal tests III

- Autocorrelation
  Generates an n bits string and measures the autocorrelation between bits d apart :

$$A_d = \sum_{i=1}^{n-d} b_i \oplus b_{i+d}$$

  Under the hypothesis that the b_i are uniform, A_d shoud have the binomial distribution and so for large $n - d$

$$\frac{2A_d - (n-d)}{\sqrt{n-d}}$$

  should be a standard normal
- Self avoiding Random Walk test

Intel MKL Vector Statistics I

The Intel MKL library includes many random number generators:

MCG31m1, R250,MRG32k3a, MCG59,WH,MT19937,SFMT19937,

MT2203,SOBOL,NIEDERREITER,Philox4x32-10,ARS

And it can provide these RNGs already shaped for specific distributions :

Continuous: *Uniform, Gaussian, Exponential, Laplace, Weibull, Cauchy, LogNormal*

Discrete: *Uniform, Bernoulli, Geometric, Binomial, Hypergeometric, Poisson, Negative Binomial*.

The following url sends you to their documentation :

`https://software.intel.com/en-us/mkl-vsnotes`

Test of Hypothesis, p-Values I

1. Make an initial assumption that is usually called $\mathbf{H_0}$ (**null hypothesis**). An alternative is called **alternative hypothesis** and is indicated by $\mathbf{H_a}$
2. Collect evidence
3. Based on data collected reject or not reject the assumption (according if $\mathbf{H_0}$ is unlikely or likely)

In the first Marsaglia's implementation of the tests he used a *confidence interval approach* , e.g $5\% - 95\%$. If the the result was outside the range the generator did'nt pass the test. In fact it has been seen that now there are hundretdths of tests and most generators would not pass some of them. Therefore in the rewriting RG Brown made the tests to report instead a *p-value* giving the possibility to appreciate which test are absolutely not failable and which can have a not so good result.

Test of Hypothesis, p-Values II

In the **p-value** approach, in the case $\mathbf{H_0}$ is true, the probability of observing a more extreme statistic is computed. If the **p-value** is small, say less than α then it is unlikely, if the **p-value** is large it is likely and the null hypothesis is not rejected.
In most of the tests for rng H_0 is the hypothesis that the random numbers are uniform : $\sim \mathscr{U}(0,1)$. This is equivalent to say that for any $t > 0$ the t-uple $(u_0, u_1, \ldots, u_{t-1})$ is uniformly distributed over $[0,1]^t$. We have two kind of statistical test:

- Single level tests
  It observes the value of a statistic Y, say y and rejects H_0 if the p-value

$$p = P[Y \leq y \mid \mathscr{H}_0]$$

  or

$$p = P[Y \geq y \mid \mathscr{H}_0]$$

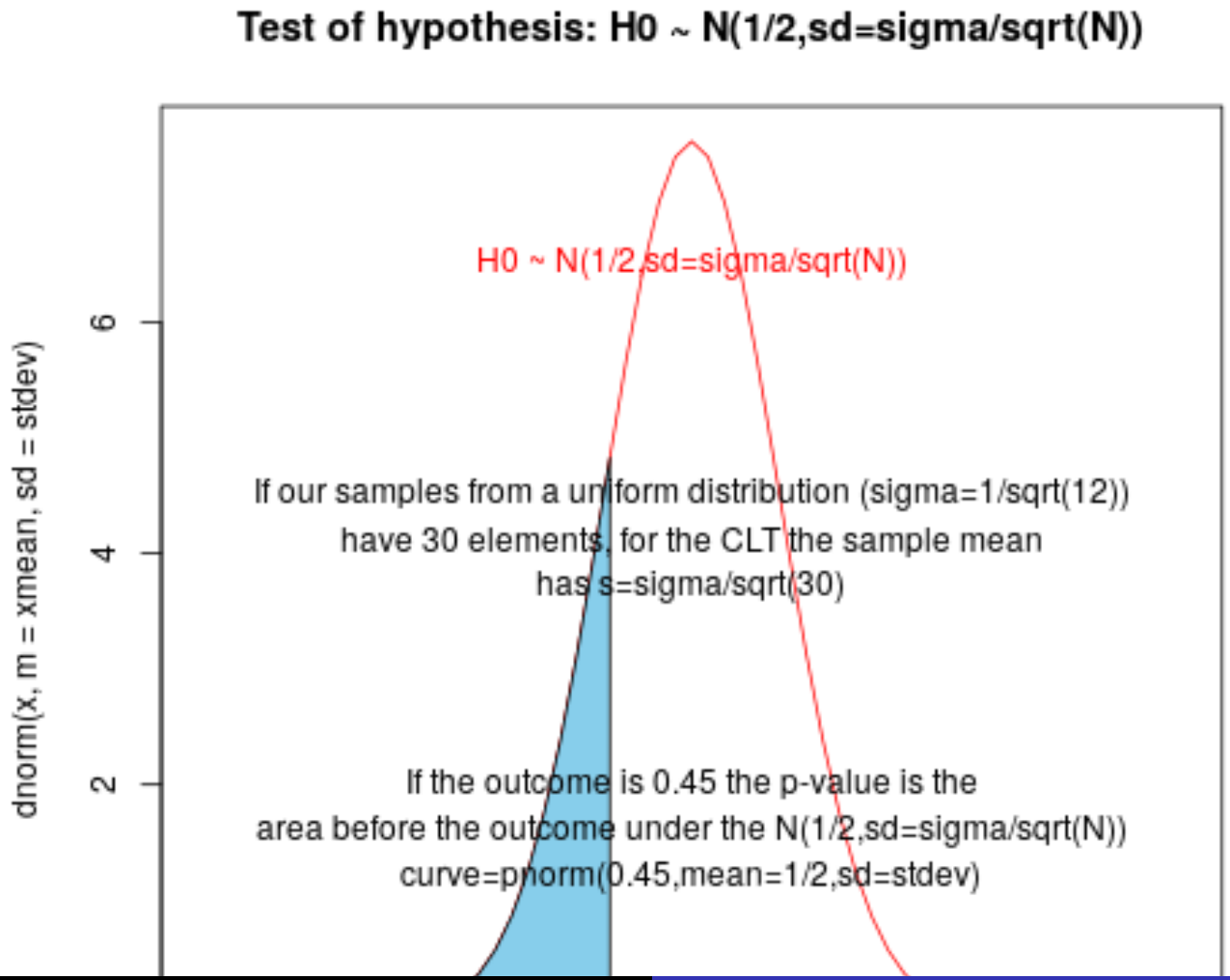
that is, the probability that an outcome, given $\mathscr{H}_0$, is more extreme than the value measured is too much close to 0 or 1.

- Two level tests
  In a second order test one generates N independent copies of Y say $Y_1, Y_2, \ldots, Y_N$ replicating the first order test.

Usually for most of the test the distribution of Y is either χ^2, normal or Poisson. In these 3 case their sum has the same distribution. That is if Y is χ^2 with k degrees of freedom then $\widetilde{Y}$ is χ^2 with Nk degrees of freedom. (χ^2 is the distribution of the sum of normal variates).
if Y is Poisson with k degrees of freedom and mean λ, then $\widetilde{Y}$ is Poisson with Nk deg of freedom and mean λ, if Y is normal with mean μ and variance σ^2 then $\widetilde{Y}$ is normal with mean $N\mu$ and variance $N^2\sigma^2$, etc.

Test of Hypothesis, p-Values IV

E.g. If the statistic $Y \sim \mathcal{N}(1/2, 1/sqrt(12 * N))$ should be normally distributed around $1/2$ with $sd = 1/sqrt(12)/sqrt(N)$ and we measure 0.45 then



Test of Hypothesis, p-Values V

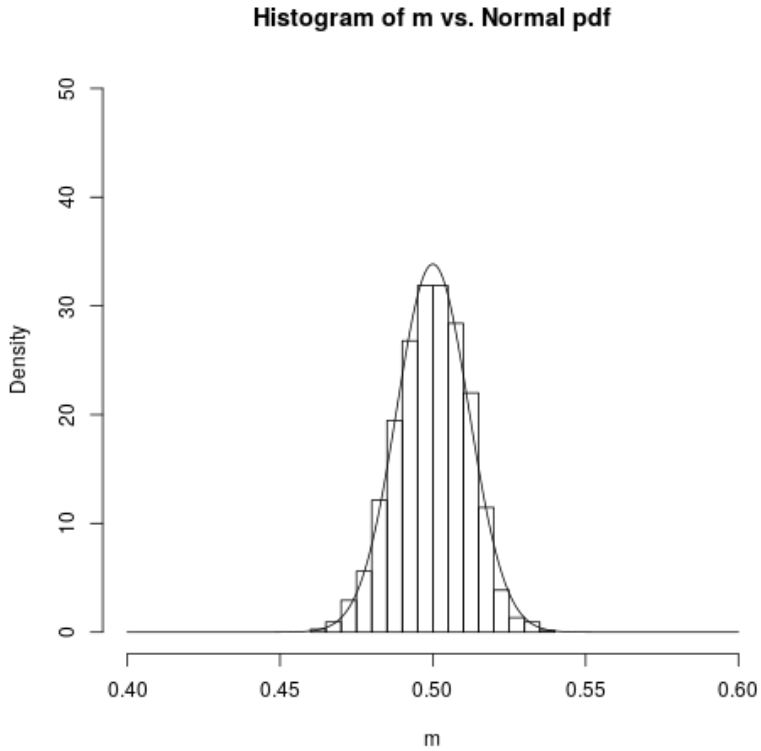
Code in R :

```
# hypothesis testing
xmi=0.2; xma=0.8;xl=0.45
x=seq(xmi,xma,0.01)
xmean=1/2;
N=30
sigma=1/sqrt(12); stdev=sigma/sqrt(N)
plot(x,dnorm(x,m=xmean,sd=stdev),type='n')
title('Test of hypothesis: H0 ~ N(1/2,sd=sigma/sqrt(N)
    )')
str=bquote(H_0 ~ N(1/2,1/sqrt(12*50)))
lines(x,dnorm(x,m=xmean,sd=stdev),col='red')
text(0.5,6.5,'H0 ~ N(1/2,sd=sigma/sqrt(N))',col='red')
xx=c(xmi-0.01,seq(xmi,xl,0.01),xl)
yy=c(0,dnorm(seq(xmi,xl,0.01),m=xmean,sd=stdev),0)
polygon(xx,yy,col='skyblue')
```

Uniformity or goodness-of-fit tests I

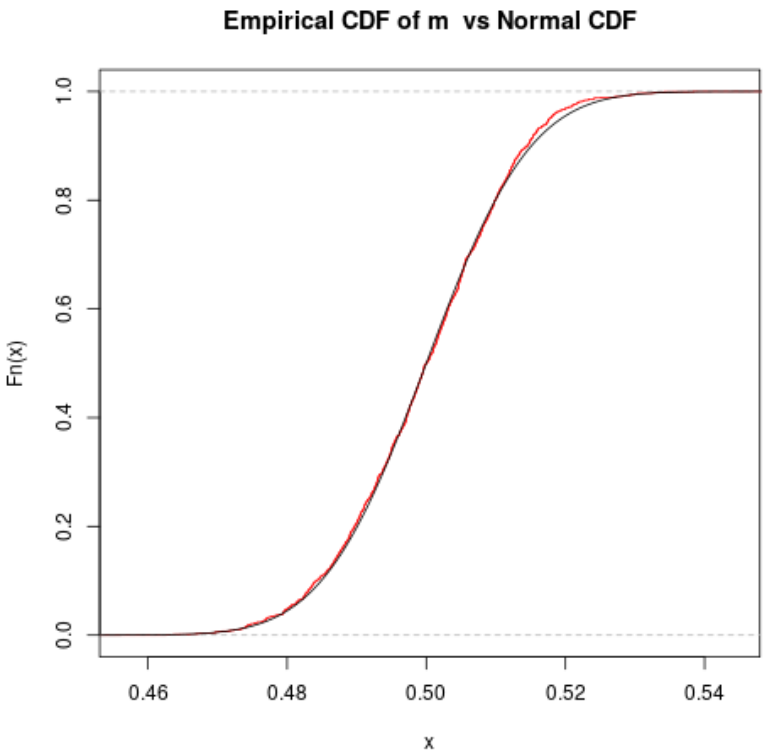
Let $\mathbb{H}_0$ be the hypothesis that our RNG produces samples $\sim \mathscr{U}(0,1)$. For the continuous uniform ditribution we know that the mean $\mu = 1/2$ and the standard deviation is $s = \sqrt{1/12}$. Let us produce some hundredths samples of some size $n > 30$and take their mean. According to the **Central Limit Theorem** the sample mean is distributed like a normal distribuition

Uniformity or goodness-of-fit tests II



$\mathcal{N}(\mu, sd = \sqrt{1/12}/sqrt(n))$.

Uniformity or goodness-of-fit tests III



Uniformity or goodness-of-fit tests IV

```
# R code
# central limit theorem
# average distr is normal(mu,sd/sqrt(n))
nlim=1500; rlim=600
x=seq(0.4,0.6,.0005)
# m[] is an array or nlim means
m<-numeric(nlim)
for (i in (1:nlim)) { m[i]=mean(runif(rlim)) }
png('clt-1.png')
hist(m,xlim=c(0.4,0.6),ylim=c(0,50),freq=FALSE,
        main='Histogram of m vs. Normal pdf')
lines(x,dnorm(x,m=1/2,sd=sqrt(1/12)*sqrt(1/rlim)))
# pause here
png('clt-2.png')
plot.new()
plot(ecdf(m),main=NULL,col='red')
title(main='Empirical CDF of m  vs Normal CDF')
lines(x,pnorm(x,m=1/2,sd=sqrt(1/12)*sqrt(1/rlim)))

ks.test(m,pnorm,mean=1/2,sd=1/sqrt(12)/sqrt(rlim)
   ,alternative=c("less"))
# E.G.:          One-sample Kolmogorov-Smirnov test
#data:  m
```

Uniformity or goodness-of-fit tests V

```
#D^- = 0.00813, p-value = 0.8201
#alternative hypothesis: the CDF of x lies below the null
    hypothesis
```

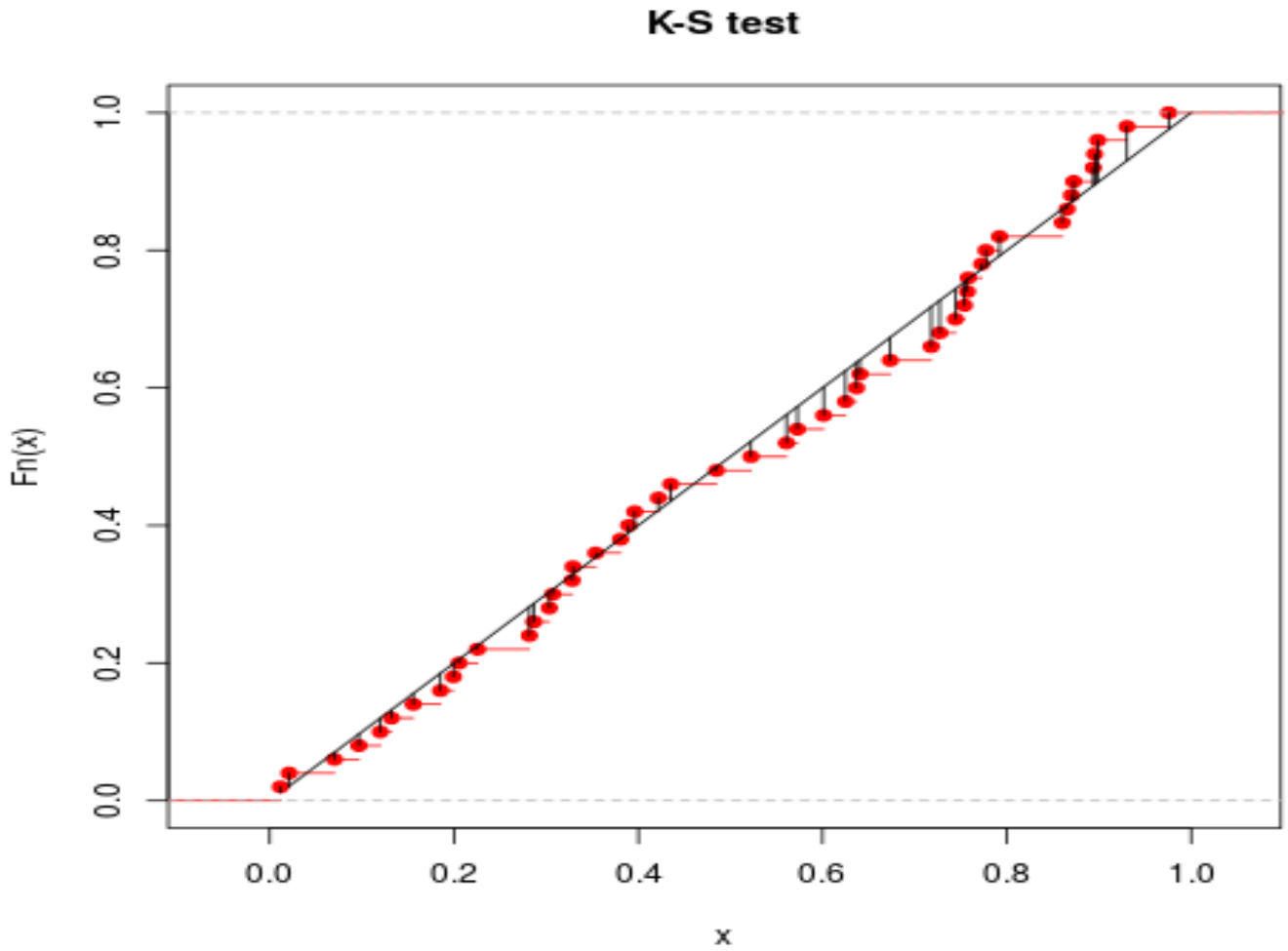
- Kolmogorov-Smirnov K-S test
- χ^2 test

Using R: Kolmogorov-Smirnov I

Kolmogorov-Smirnov measures the maximal deviations between the expected theoretical **cumulative distribution function** (CDF)

Using R: Kolmogorov-Smirnov II

and the **empirical distribution function** (ECDF) obtained from



the data.

Using R: Kolmogorov-Smirnov III

How to compute a p-value from k-s test?

The exact formula for the distribution is :

$$P(k_N^{\pm} \leq s) = \frac{s}{\sqrt{N}} \sum_{0 \leq k \leq \lfloor \sqrt{N}s \rfloor} (-1)^k \binom{N}{k} \left(\frac{\sqrt{N}s - k}{N}\right)^k \left(1 + \frac{\sqrt{N} - k}{N}\right)^{N-k-1}$$

For $N \geq 100$ we can use :

$$P(k_N^{\pm} \leq s) = 1 - e^{-2(s + \frac{1}{6\sqrt{N}})^2}$$

Ferrenberg(1992), Schmid (1995) : 2D Ising model, Blume-Capel model I

A word of caution about use of RNG in large simulations. These generators that performed relatively well on normal tests (at that time : 1992) were reported to fail a **Monte Carlo test** of the 2D Ising model [25]. Some then proposed to use simulation with a theoretical predictable outcome for testing RNG.

CONG

The linear congruential generator: $LCG(16807, 0, 2^{31} - 1)$

2 SHR

2 shift register generators : $F(250, 103, \oplus)$, $F(1279, 1063, \oplus)$

SWC

A subtract with carry generator: $F(1279, 1063, -)$

Ferrenberg(1992), Schmid (1995) : 2D Ising model, Blume-Capel model II

SWCW

A combined subtract with carry and Weyl generator

Schmid [32] reports about large failures using GFSR for the tricritical Blume-Capel model using single spin Metropolis update, due to strong triples correlation of the generator. They found that the average of $\langle X_n \cdot X_{n-p} \cdot X_{n-p+q} \rangle$ for a GFSR is 20% lower than expected $(= 1/8)$.

Closing consideration

RNGs are a swamp in which is easy to become trapped.
To devise a RNG :

- Choose an algorithm already studied and for which theory can predict the period
- Consult publication about seed and parameters for it to get a good rng
- Test it thouroughly with TestU01 and DieHarder.

The modern view about the algorithms is that today probably an *LCG* is not enough for current simulations, but can be conviniently combined with one of the other algorithms like SHR or LFSR.
The Mersenne twister is now considered by many the best rng around.
It can be it will be overtaken by WELL that was created by IEcuyer and the japanese team of MT to solve some issues.

Lex aurea of RNG users

- **Never be first, never be last to use a RNG !**
  Paraphrased from pharmacology :
  Never the first, never the last to prescribe a drug to patients !

Famous Quotes: Von Neumann, Knuth, Marsaglia, . . .

- *Von Neumann* : Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
- *D.Knuth* Random numbers should not be generated with a method chosen at random.
- *Marsaglia* : Random numbers are like sex: Even if they are not very good, they are still pretty good.
- *Robert Coveyou* The generation of random numbers is too important to be left to chance.

quasi-Monte Carlo

A side argument :

- **quasi-random numbers** (or sub-random)
- **quasi-Monte Carlo**

Uses instead of random numbers *low discrepancy* numbers (or quasi-random). Can converge faster if function is smooth.

Exercise buffon

1. Pick a programming language and OS of your choice
2. Choose an available rng
3. Write a program to simulate the Buffon experiment (if you are able show it in graphic mode)
4. Report on a small table the sample size and the results

Exercises

- Transform a uniform variate into one uniform over $[a, b]$
- Transform a uniform variate into an Exponential variate PDF $\lambda e^{-\lambda x}$, CDF $1 - e^{-\lambda x}$
- Using Box-Muller transform a uniform variate in a normal one

Exercise Knuth's Super-random

1. Pick a programming language and OS of your choice
2. Implement Knuth's Super-random
3. Report on the flaws you find generating rn

Exercise NR Ran

Ran.h

Implement Numerical Recipes *Ran* and test it with *dieharder*,

Exercise on MRG

Devise an MRG generator of maximum period and test it with *dieharder*. Report it.

Exercise sprng

1. Choose a OS and a language between C++ and Fortran
2. Install SPRNG as detailed in previous slides
3. Make a program to check the simple interface both in serial and in parallel mode
4. Make a program to check the full interface of SPRNG

Exercise kiss99

1. Pick a programming language and OS of your choice
2. Implement Marsaglia's KISS99 rng
3. Report on the rn it generates
4. Test its randomness with *DieHarder* or *TestU01*

MLCG, LCG exercises

We know from theory how to produce a maximal period generator in these 2 cases, but we should make an heuristic/ experimental search for the good ones.

- implement Lehmer generator and run *dieharder* on its output
- produce an MLCG of maximal period $m - 1$
- produce an LCG of maximal period m
- check both with *dieharder*, eventually change them

Docker container for the exercises

A **docker container** based on ubuntu with everything you need for the exercises and most of the algorithms and tests cited can be run with :

```
docker run -it rinnocente/rng-docker-2018
```

References I

[1] Donald Knuth.
The Art of Computer Programming : vol 2. Seminumerical Algorithms
2nd ed, 1981.

[2] Donald Knuth.
The Art of Computer Programming : vol 2. Seminumerical Algorithms
4th ed, 1999.

[3] Press, Teukolsky, Vetterling, Flannery
Numerical Recipes - The Art of Scientific Computing
Cambridge University Press, 3e, 2007

[4] Gentle,J.E.
Random Number Generation and Monte Carlo methods
Springer, 2e, 2003

References II

[5] A.Rukhin, J. Soto, et al.

A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications

National Institute of Standards and Technology (D.Of Commerce)

April 2010.

[6] H.Niederreiter

Pseudorandom Vector Generation

Random Number Generation and Quasi-montecarlo methods (CBMS-NSF regional conference series in applied mathematics 63), 1992

References III

[7] H.Niederreiter

Factorization of polynomials and some linear-algebra problems over finite fields

Linear Algebra and its Applications

Volume 192, 1993

[8] H.Niederreiter

The Multiple Recursive Matrix Method

Finite Fields and Their Applications

Vol 1,issue 1, 1995

[9] H.Niederreiter, A.Winterhof

Applied Number Theory

Springer, 2015

References IV

[10] Crandall R., Pomerance C.
Prime Numbers - A computational Perspective
Springer, 2015

[11] Solomon Wolf Golomb
Shift register sequences
Holden Day, 1967

[12] Solomon Wolf Golomb, Lloyd R. Welch, Richard M. Goldstein, Alfred W. Hales
Shift Register Sequences
Aegean Park Press, 1982

[13] Menezes, van Oorschot, Vanstone
CRC Handbook of Applied Cryptography
CRC Press, 1996

References V

[14] Lehmer, D. H.

Mathematical methods in large-scale computing units

Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery: 141146. MR 0044899. 1949. journal : Annals of the Computation Laboratory of Harvard University, Vol. 26 (1951)).

[15] G. Marsaglia.

Random numbers fall mainly in the planes.

PNAS, 61 (1): 2528. 1968.

[16] A new class of Random Number generators (AWC,SWB)

G.Marsaglia, Zaman

The Annals of Applied Probability, 1991, n. 3

References VI

[17] Tables of Linear Conguential Generators of different sizes and good lattice structure
P. L'Ecuyer
Mathematics of Computation
Volume 68, Number 225, Januiary 1999

[18] Pierre L'Ecuyer , Richard Simard
TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators
ACM Transactions on Mathematical Software
33: 22. 2007.

[19] Pierre L'Ecuyer
Uniform random number generation.
Handbooks in Operations Research and Management Science
13:55-81, 2006

References VII

[20] Matsumoto, Kurita

Twisted GFSR Generators

ACM Transactions on Modeling and Computer Simulation

Year:1992 Month:7 Day:1 Volume:2 Issue:3

[21] Matsumoto, Kurita

Twisted GFSR Generators II

ACM Transactions on Modeling and Computer Simulation

Volume 4 issue 3 1994

[22] Matsumoto, M.; Nishimura, T.

Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator

ACM Transactions on Modeling and Computer Simulation

8 (1): 330. doi:10.1145/272991.272995. 1998.

References VIII

[23] Mutsuo Saito, Makoto Matsumoto
A variant of mersenne twister suitable for graphic processors.
CoRR, abs/1005.4972, 2010

[24] Robert G. Brown
Dieharder: A random number test suite
http://www.phy.duke.edu/~rgb/General/dieharder.php , 2009

[25] Alan M.Ferrenberg, D.P.Landau and Y.Joanna Wong
Monte Carlo simulations: Hidden errors from "good" random number generators
Phys.Rev.Lett. 69,3382(1992)

References IX

[26] Panneton, F. O., I'Ecuyer P., Matsumoto M.
Improved long-period generators based on linear recurrences modulo 2 (WELL)
ACM Transactions on Mathematical Software. 32 (1): 116. doi:10.1145/1132973.1132974

[27] G. E. P. Box and Mervin E. Muller
A Note on the Generation of Random Normal Deviates
The Annals of Mathematical Statistics (1958)
Vol. 29, No. 2 pp. 610611

[28] S.Park, K.Miller
Random numbers generators : good ones are hard to find
Communications of the ACM CACM Volume 31 Issue 10, Oct. 1988 Pages 1192-1201

References X

[29] Richard Brent

Search for primitive trinomials (mod 2)

http://maths-people.anu.edu.au/~brent/trinom.html , 2008

[30] Richard Brent, Paul Zimmermann

TWELVE NEW PRIMITIVE BINARY TRINOMIALS

https://arxiv.org/pdf/1605.09213.pdf

[31] Lewis, Payne

Generalized Feedback Shift Register Pseudorandom Number Algorithm

Journal of the ACM Volume 20 Issue 3, July 1973 Pages 456-468

References XI

[32] Errors in Monte Carlo simulations using shift register rand om number generators
F. Schmid and N.B. Wilding
https://arxiv.org/pdf/cond-mat/9512135.pdf

[33] Coveyou, M; R.Macpherson
Fourier Analysis of Uniform Random Number Generators
Journal of the Association for Computing Machinery
1967

[34] Kirckaptrick, Stoll
A Very Fast Shift-Register Sequence Random Number Generator

Journal of Computational Physics 40
1981

References XII

[35] J.D.Parker
The period of the Fibonacci random number generator
Discrete Applied Mathematics 20
(1988), 145-164
North-Holland

E N D