# Generation of Random Numbers

Credits: HILL David - Wikipedia
WRIGHT Nicolas,
WOLMER Julien, CRAWFIS Roger

## Outline:

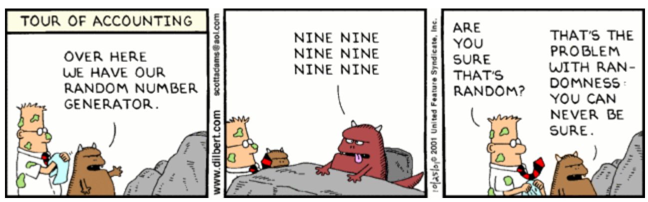
**« Generation of random numbers is too important to be left by chance » :**
Robert R. Coveyou
Oak Ridge National Laboratory

- Pseudo Random Numbers
- Quasi Random Numbers
- True Random numbers

**Generation of non-uniform variates**

# Introduction
## Random Number Generation Basics

- ✓ Generate a sequence of numbers uniformly distributed between 0 and 1
- ✓ This property helps : Transforming this uniform distribution to... any kind of distributions!




## What is an random number ?

- Is 7 a random number ?
  - 7 is a prime number, but there is no such thing as single random number
- A random number comes with its friends...
  - A set of numbers that have "no statistical relation" with the other numbers in the sequence
- In a Uniform distribution of random numbers the range retained is [0,1]. Within this range, every number has the same chance of turning up.
  - 0.0000001 is just as likely as 0.5

## Random number vs Pseudo random numbers

- **True Random Numbers** (TRN) have no defined sequence or formulation. Thus, for any *n* random numbers, each appears with equal probability. They come from : Radioactive decay, Thermal noise, Cosmic ray arrival… (They could be interesting for cryptography…)
- If we restrict ourselves to computer algorithms generating numbers that (tries) to have no statistical correlations, we call them **Pseudo-Random Numbers.**
- Pseudo-random numbers have an advantage for numerical experiments, they are **reproducible.**
- Not being able to reproduce experiments is what differs a Science from Pseudo-science (and a scientist from a pseudo-scientist).

## Standard system libraries – not for Science

- **Pseudo-Random Numbers** produced by a basic pseudo-random generator (PRNG) are interesting for the operating system, for hardware purposes, for game development, for prototyping, but not for scientific experiments.
- Standard C Library :
  - See "man rand" on your Unix environment, Windows or Mac documentation
  - Rather poor pseudo-random number generator with small period
  - Often results in 16-bit integers from 0 up to RAND_MAX (32767)
  - Some have better performance with periods up to $2^{32}$, but they are still very week. (only 4 billions possibilities).

## Initialization of a PRNG

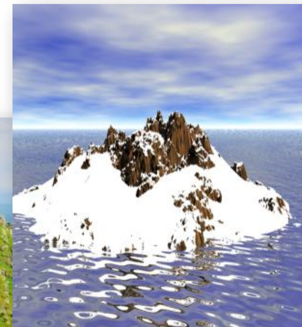
- Pseudo-Random Numbers Generator algorithms have some state that can be initialized.
- For basic algorithms, the sate is only the last generated number. With old generators the state is often called a seed, and the process of initializing is called seeding.
- For fine generators, the state can be complex and up to a few kilobytes (6KB for the Mersenne Twister for instance) and we speak about "PRNG status initialization".
- We can set this state using the initialization methods given by the generator API like srand(), srand48()...
  - But why would you want to do this?

## Initialization of a PRNG (continued)

There are 2 main reasons to setup PRNG statuses

- **For Science and for debugging**,
  - You need a deterministic an repeatable process (How do you debug if you program is driven by TRNs – at each run you have a different program behavior...)
- **To run different independent experiments :**
  - Since we need a deterministic PRNG, the default (same) initialization state will always generates the same sequence of random numbers and the same program behavior (euh... not really random isn't it ?).
  - Common solutions:
    - **Run loops of experiments without re-initializing the generator between two experiments (need a period long enough).**
    - **Call the initialization method for each independent experiment with 'independent' statuses (complex to determine – used for parallel computing with care).**

## Ex: Generating the same image at each run



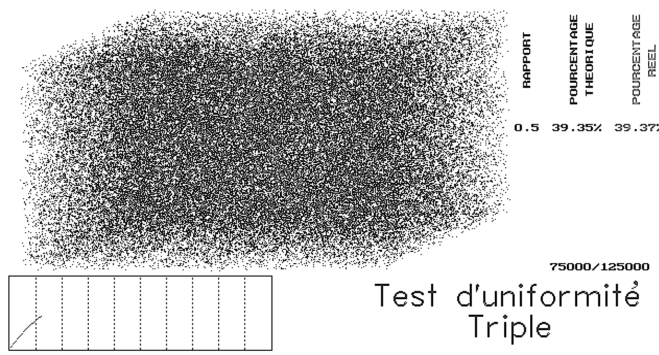
# Part I

# Deterministic Generation

(1) Pseudo Random Numbers
(2) Quasi Random Numbers

## We need a uniform reproducible Pseudo-Random Number Generator (PRNG)

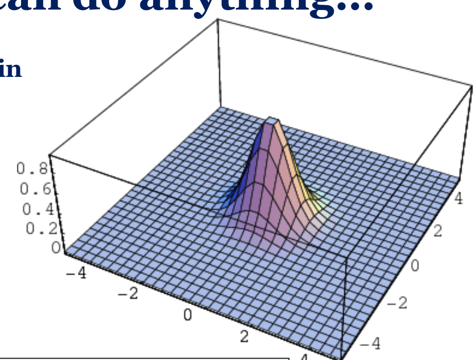
- How do we test for randomness ?
  - Statistical tests
  - Empirical tests
- How do we test for Uniformity ? Spectral tests…

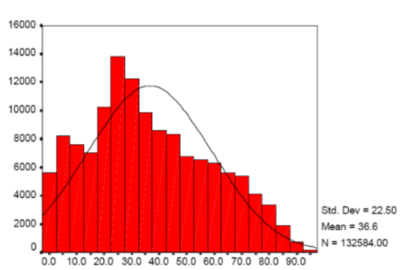


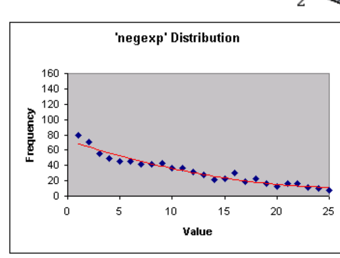

## With uniformity we can do anything…

***« God does not play dice…»* A. Einstein**

1. Anamorphosis (inverting a distribution law) x = F$^{-1}$ (RandomNumber)
2. Rejection Method
3. Reproducing Histograms



## Linear Congruential Generators (LCGs)

- Based on a linear recurrence formula
- For instance: $x_n = (5x_{n-1} + 1)\bmod 16$
- With $x_0 = 5$ we get:

$$x_1 = (5.(5) + 1)\bmod 16 = 26 \bmod 16 = 10$$

- The 32 first pseudo-random numbers generated are:

10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5.

## How do we get real numbers between 0 and 1 ?

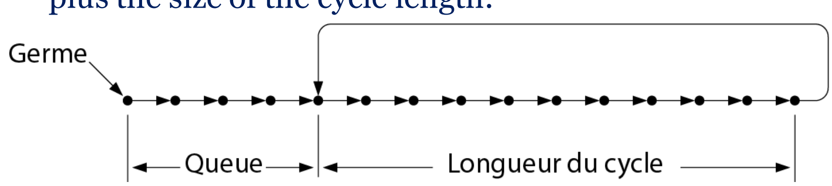
- We divide the obtained number by the maximum value.
- In this toy case, 16 is the modulus (maximum) we get numbers between [0..1[
- For the LCG presented we get: 0.625, 0.1875, 0, 0.0625, 0.375, 0.9375, 0.75, 0.8125, 0.125, 0.6875, 0.5, 0.5625, 0.875, 0.4375, 0.25, 0.3125, 0.625, 0.1875, 0, 0.0625, 0.375, 0.9375, 0.75, 0.8125, 0.125, 0.6875, 0.5, 0.5625, 0.875, 0.4375, 0.25, 0.3125

## LCGs main characteristics

- When the generator is known, it is possible to reproduce sequences from the *xo* initial value.
- We deterministically produce a sequence that mimicks randomness
- Reproducibility is the essence of Science
- This value is used to initialise a generator to a different state, thus providing a new sequence. In this simple case of generator the value is called a **seed.**
- In the basic exemple, we have a cyclic repetition of the 16 first numbers. The length of the cycle (also called period) is 16

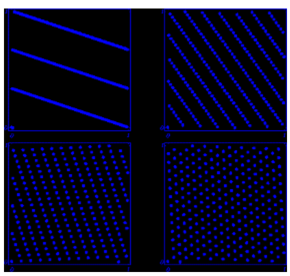
## Some particularities…

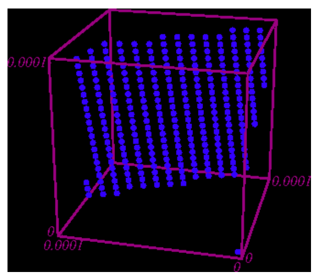
- Depending on the initial state, the cycle can be found after an initial queue or warming period.
- In this case the maximum number of different pseudorandom values is equal to the size of the queue plus the size of the cycle length.

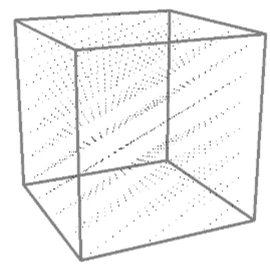


## Known problems with LCGs

- **They have to be avoided for scientific applications !**
- Their mathematical stucture presents weaknesses that prevent them to be successful for spectral tests (possible bad uniformity in more than 1 dimension)
- LCGs are fast and can be used for game and other software applications







## Tausworthe Generators

- Proposed by Tausworthe in 1965
- Application in cryptography
- Generation of long random streams
- Random sequences of binary numbers that can be divided in substrings of a given size

## Tausworthe Generators

- The recurrence formula is given by:

$$b_n = c_{q-1}b_{n-1} \oplus c_{q-2}b_{n-2} \oplus c_{q-3}b_{n-3} \oplus \ldots \oplus c_0 b_{n-q}$$

Where $c_i$ and $b_i$ are binary variables

- When this kind of generator uses the last q bits of a sequence. It is named an AutoRegressive sequence of order q or AR(q).
- An AR(q) generator can have a maximum period of $2^{q-1.}$

19

## Tausworthe Generators

- Supposing that we have a « Delay » operator D such as $Db_n=b_{n+1}$ then:

$$D^q b(i-q) = c_{q-1}D^{q-1}b(i-q) + c_{q-2}D^{q-2}b(i-q) + \ldots + c_0 b(i-q) \bmod 2$$

*where*

$$D^q - c_{q-1}D^{q-1} - c_{q-2}D_{q-2} - \ldots - c_0 = 0 \bmod 2$$

*or*

$$D^q + c_{q-1}D^{q-1} + c_{q-2}D_{q-2} + \ldots + c_0 = 0 \bmod 2$$

20

## Tausworthe Generators

- Such an operator is a polynom named **caracteristic polynom**, and this becomes more readable by replace D by x :

$$x^q + c_{q-1}x^{q-1} + c_{q-2}x^{q-2} + \ldots + c_0$$

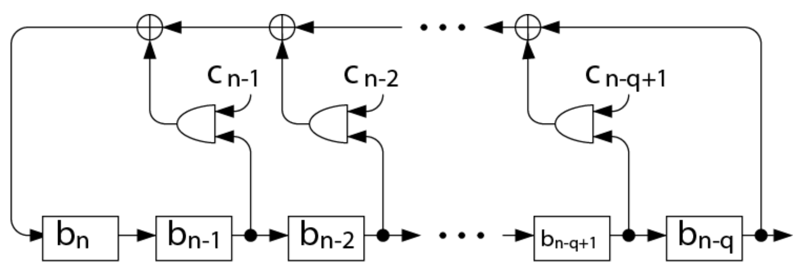


## Tausworthe Generators

- The generator period depends on this characteristic polynomial
- For a polynom of order q its maxium period is equal to $2^{q-1}$.
- The polynom giving the maximum period is the named a **primitive polynomial**.

## Tausworthe Generators

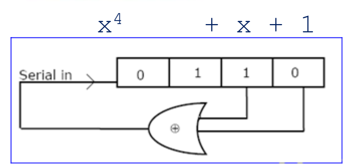
- Tausworthe sequences can be easily generated with shift registers.




General polynom of order q

23

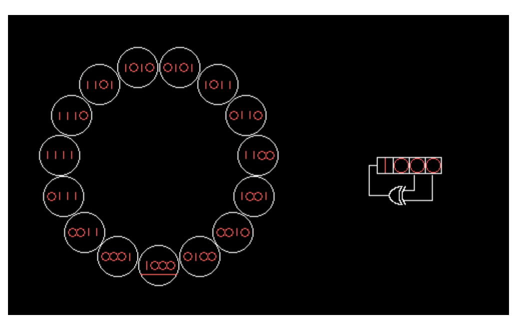
## This leads to what we call: LFSR for Linear Feed Back Shift Register generators




This (wikipaedia) animated GIF presents the functioning of a 4-bits Linear Feed Back Shift Register (Fibonacci like) with its complete state diagram.

$x^4 + x + 1$

A simple XOR gate is used for the feedback and the bit is reinjected on the left after a rigtht shift of the register.



The maximum sequence is obtained with all the possible values except 0 ( $2^4$-1 states)
(See Wikipedia for more details of this short tutorial)

24

## Bitwise operators in C

- & Binary AND
- | Binary OR
- ^ Binary XOR
- << Binary left shit
  - yourVariable << 1 ; One bit left shift ( x 2 )
  - yourVariable << 4 ; One bit left shift ( x 16 )
- >> Binary right shit
  - yourVariable >> 1 ; One bit right shift ( / 2 )
  - yourVariable << 3 ; One bit right shift ( / 8 )
- ~ Binary NOT One's Complement - has the effect of 'flipping' bits.

## Decimal, Hexadecimal and binary equivalence

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Dec** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Hex** | 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 |
| **Bin** | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Dec** | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **Hex** | 0x8 | 0x9 | 0xA | 0xB | 0xC | 0xD | 0xE | 0xF |
| **Bin** | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Dec** | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| **Hex** | 0x10 | 0x11 | 0x12 | 0x13 | 0x14 | 0x15 | 0x16 | 0x17 |
| **Bin** | 1 0000 | 1 0001 | 1 0010 | 1 0011 | 1 0100 | 1 0101 | 1 0110 | 1 0111 |

| | | | | | |
|---|---|---|---|---|---|
| **Dec** | 31 | 32 | 63 | 128 | 255 |
| **Hex** | 0x1F | 0x20 | 0x3F | 0x80 | 0xFF |
| **Bin** | 0001 1111 | 0010 0000 | 0011 1111 | 1000 0000 | 1111 1111 |

## Examples of value shifting (left & right)

| SYNTAX | BINARY FORM | VALUE |
|---|---|---|
| x = 7; | 00000111 | 7 |
| x=x<<1; | 00001110 | 14 |
| x=x<<3; | 01110000 | 112 |
| x=x<<2; | 11000000 | 192 |

| SYNTAX | BINARY FORM | VALUE |
|---|---|---|
| x = 192; | 11000000 | 192 |
| x=x>>1; | 01100000 | 96 |
| x=x>>2; | 00011000 | 24 |
| x=x>>3; | 00000011 | 3 |

## Examples of bitwise operations

```
A      0 1 1 0 1 0 0 1   0x69
B      1 1 0 0 1 0 1 0   0xCA
A & B  0 1 0 0 1 0 0 0   0x48

A      0 1 1 0 1 0 0 1   0x69
B      1 1 0 0 1 0 1 0   0xCA
A | B  1 1 1 0 1 0 1 1   0xEB

A      0 1 1 0 1 0 0 1   0x69
B      1 1 0 0 1 0 1 0   0xCA
A ^ B  1 0 1 0 0 0 1 1   0xA3

A      0 1 1 0 1 0 0 1   0x69
~A     1 0 0 1 0 1 1 0   0x96
```

## Sample C code

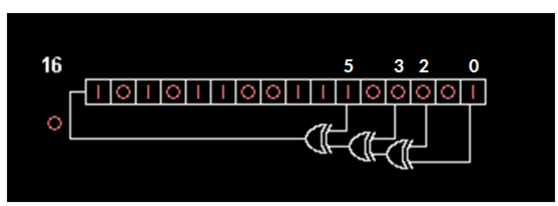
```c
#include <stdio.h>

int main(void) {
        unsigned char a = 0x69;  // 01101001
        unsigned char b = 0xca;  // 11001010

        printf("a & b = %02x\n", a & b);

        unsigned char c = a | b;

        printf("a | b = %02x\n", c);

        b ^= a;  // b = b ^ a
        printf("a ^ b = %02x\n", b);

        printf("~a = %x\n", ~a);
        printf("~a & 0xff = %02x\n", ~a & 0xff);
}
```

## Output

```
a & b = 48
a | b = eb
a ^ b = a3
~a = ffffff96
~a & 0xff = 96
```

## Principle of characteristic polynomials in LFSR

- Example of LFSR of Fibonacci kind with 3 lags on 16 bits




- The 4 feedback tap numbers in white correspond to a **primitive polynomial** selected to maximize the number of states: 65535 states (excluding the all-zeroes state.

$$x^{16} + x^5 + x^3 + x^2 + 1$$

- The hexadecimal state "ACE1 hex" shown on this image will be followed by "5670 hex"

30

## Tausworthe Generators

- Given an AR(q) sequence, Tausworthe proposes the construction of $x_n$ numbers of *l* bits
- The $b_n$ bit sequence is split in successive groups of 's' bits
- The first *l* bits of each group are given by:

$$x_n = 0.b_{sn}b_{sn+1}b_{sn+2}b_{sn+3}\ldots b_{sn+l-1}$$

*or*

$$x_n = \sum_{j=1}^{l} 2^{-j} b_{sn+j-1}$$



## Tausworthe Generators : properties

- *s*, is a constant s ≥ *l*
  - This ensures that the generated numbers do not have overlapping bits.
- *s* is prime to $2^q$-1
  - This ensures that the numbers of *l* bits are drawn within an integer period.
- The *l* bits numbers generated by the preceeding equations have the following properties:
  - The average of the sequence of numbers is: ½
  - The variance is of: 1/12
  - The coorrelation of the whole serie is: 0

## Tausworthe Generators : an example

- Considering the following primitive polynomial:

$$x^7+x^3+1$$

- Then we have:

$$b_{n+7} \oplus b_{n+3} \oplus b_n = 0, \mathrm{n} = 0,1,2,...$$
$$where$$
$$b_{n+7} = b_{n+3} \oplus b_n, \mathrm{n} = 0,1,2,...$$
$$substituting \; \mathrm{n} \; by \; \mathrm{n} - 7,$$
$$b_n = b_{n-4} \oplus b_{n-7}, \mathrm{n} = 7,8,9...$$

33

## Tausworthe Generators : an example

- With $b_0=b_1=...=b_6=1$, we obtain the following sequence of bits:

$$b_7 = b_3 \oplus b_0 = 1 \oplus 1 = 0$$
$$b_8 = b_4 \oplus b_1 = 1 \oplus 1 = 0$$
$$b_9 = b_5 \oplus b_2 = 1 \oplus 1 = 0$$
$$b_{10} = b_6 \oplus b_3 = 1 \oplus 1 = 0$$
$$b_{11} = b_7 \oplus b_4 = 0 \oplus 1 = 1$$
$$...$$

34

## Tausworthe Generators : an example

- The complete sequence is:

1111111 0000111 0111100 1011001 0010000

0010001 0011000 1011101 0110110 0000110

0110101 0011100 1111011 0100001 0101011

1110100 1010001 0111111 1000011 1000000

- The first 7 bits constitute the seed of this generator.
- This sequence is repeating itself every 127 bits.
- $2^7-1=127$, the $x^7+x^3+1$ is a **prime polynomial**.

35

## Most stochastic simulations use PRNGs (Pseudo Random Number Generators)

**Quick survey of the main PRNG types: Green recommended**

- LCG (Linear Congruential Generator)
  $x_i = (a*x_{i-1} + c) \bmod m$
- LCGPM (Linear Congruential Generator with Prime Modulus – could be Mersenne or Sophie Germain primes)
- **MRG** (Multiple Recursive Generator)
  $x_i = (a_1*x_{i-1} + a_2*x_{i-2} + \ldots + a_k*x_{i-k} + c) \bmod m$ – with k>1
  (Ex: **MRG32k3a** & MRG32kp – of l'Ecuyer and Panneton)
- LFG (Lagged Fibonacci Generator)
  $x_i = x_{i-p} \square x_{i-q}$
- L & GFSR (Generalised FeedBack Shift Register…) Mod 2
  (Ex: **Mersenne Twisters** (MT19237, SFMT, MTGP) – **WELLs**
- **MLFG** (Multiple Lagged Fibonacci Generator) - Mascagni
- Counter Based Generators (SC'11) some based on cryptographic standards (AES,Threefish) and some completely new (Philox).

## Requirements for PRNGs

- *prop. 1 :* numbers are uniformly generated
- *prop. 2 :* the sequence is uncorrelated
- ***prop. 3 :*** **the sequence is reproducible**
- *prop. 4 :* the generator is portable on any computer
- *prop. 5 :* the sequence can be changed by adjusting an intial status (seed on primitive generators)
- *prop. 6 :* the period is as large as possible
- *prop. 7 :* the generator satisfies any random test
- *prop. 8 :* a quick generation is obtained
- *prop. 9 :* the generator uses a limited computer memory

37

## Properties and structures of PRNGs

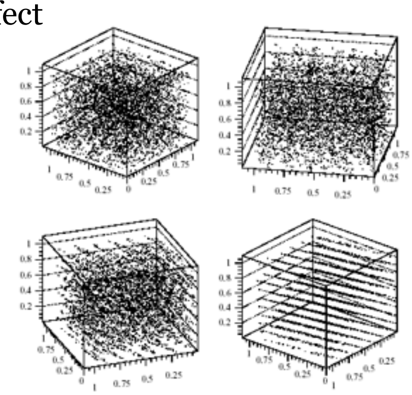
- Due to their structures (linear generators) and characteristics (limited period) basic generators fail to give satisfying results when used in high dimensions, they fail to what is called the "spectral test".
- The restriction to quantized numbers (a finite-set), leads indeed to problems in high-dimensional space. Many points end up to be co-planar. For ten-dimensions, and 32-bit random numbers, this leads to only 126 hyper-planes in 10-dimensional space.

## Example of 3D distribution of “RANDU”

- RANDU was introduced in the 60’s by IBM for its famous 370 computer.

$$X_{n+1} \equiv (65539 \times X_n) \quad \text{mod } 2^{31}$$

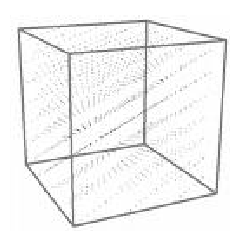
- It was the perfect example of bad PRNG that could not be used for scientific applications



Problems seen when observed at the right angle

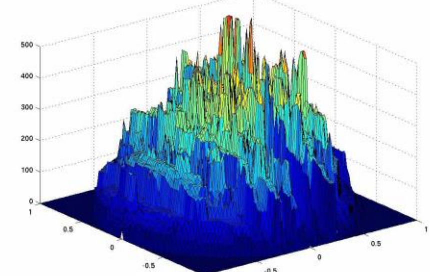
## Batteries of tests for “random” numbers

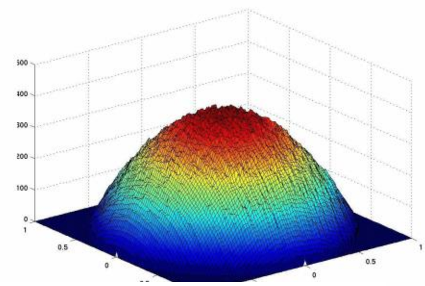
- Some tests initially proposed by Knuth
- DieHard (G. Marsaglia) – 1990s
- NIST – STS (mainly for cryptography…)
- Tests form L’Ecuyer, Matsumoto, Kurita…
- DieHarder… (R.G. Brown)
- **The most complete test battery is currently**: **TestU01**
  - L’Ecuyer et al. 2002-2006
  - Small crush, crush and big crush with more than a hundred statistical tests



40

## Let's « see » some real impact of the generator quality…



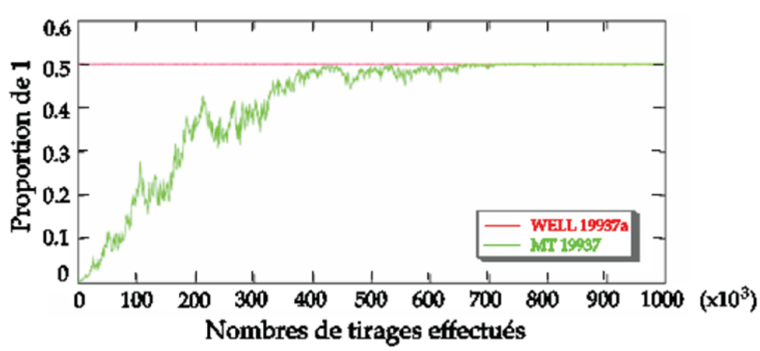


Here we see two results of the same PDE simulation (sequential).
On the left the image is obtained with Linux default rand in 2007 (which is already far better than the old std UNIX rand on 15bits)
**On the right** – same simulation with Mastumoto Mersenne Twister

41

## There is no perfect Generator (1/2) First Mersenne Twister (1997): a known default now corrected…

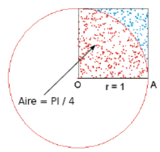
Between 1997 and 2002: very long recovery of **zero-excess initial state** for MT19237 (700 000 drawings…)

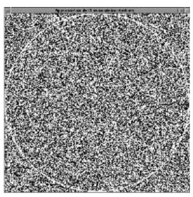


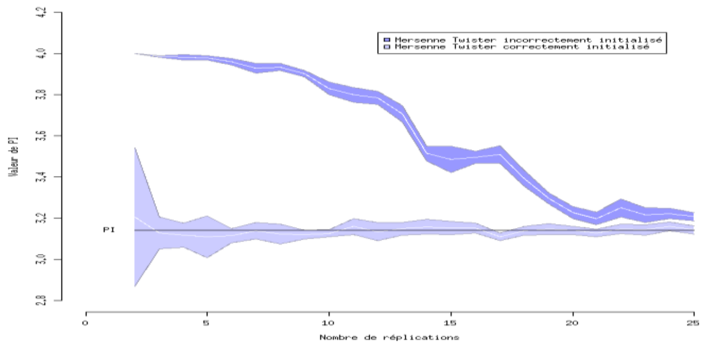

42

## There is no perfect Generator… (2/2)

*“We are here dealing with mere cooking recipes for making digits”* (1951 Von Neuman)






If it is not correctly initialized, even a very good generator can lead to biased results. Here, we obtain 3.25 for π after 25 replications each drawing 50 000 points (100000 pseudo-random numbers). Darker blue : 3.14 is not even in the confidence interval after the 25 replicates… Lighter blue : a correct simulation.