MICS-3: Computational Statistics

Lesson 1: Random number generators for simulations

Raymond Bisdorff

University of Luxembourg

3 octobre 2019

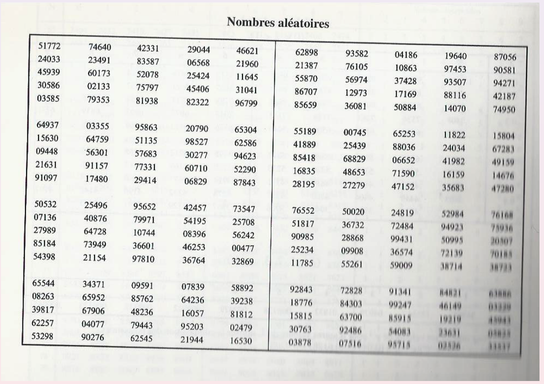
Numbers "chosen at random"

Numbers "*chosen at random*" are mostly required in order to :

a) Simulate natural phenomena or operational systems in a realistic manner ;
b) Sample potential cases in order to uncover typical behaviour when it is impractical to observe all cases ;
c) Test effectiveness and performance of algorithms and software components ;
d) Cipher messages for secure, trustful and reliable communications ;
e) Hash the access to data structures and storage areas.

Manually pick numbers from a random table

You are requested to draw two independent samples of 100 random integers in the range [0; 500] from the table of random numbers provided in the resources of the lecture.



Nombres aléatoires

51772	74640	42331	29044	46621	62898	93582	04186	19640	87056
24033	23491	83587	06568	21960	21387	76105	10863	97453	90581
45939	60173	52078	25424	11645	55870	56974	37428	93507	94271
30586	02133	75797	45406	31041	86707	12973	17169	88116	42187
03585	79353	81938	82322	96799	85659	36081	50884	14070	74950
64937	03355	95863	20790	65304	55189	00745	65253	11822	15804
15630	64759	51135	98527	62586	41889	25439	88036	24034	67283
09448	56301	57683	30277	94623	85418	68829	06652	41982	49159
21631	91157	77331	60710	52290	16835	48653	71590	16159	14676
91097	17480	29414	06829	87843	28195	27279	47152	35683	47280
50532	25496	95652	42457	73547	76552	50020	24819	52984	76168
07136	40876	79971	54195	25708	51817	36732	72484	94923	75936
27989	64728	10744	08396	56242	90985	28868	99431	50995	20507
85184	73949	36601	46253	00477	25234	09908	36574	72139	70185
54398	21154	97810	36764	32869	11785	55261	59009	38714	38723
65544	34371	09591	07839	58892	92843	72828	91341	84821	[illegible]
08263	65952	85762	64236	39238	18776	84303	99247	46149	[illegible]
39817	67906	48236	16057	81812	15815	63700	85915	19219	[illegible]
62257	04077	79443	95203	02479	30763	92486	54083	23631	[illegible]
53298	90276	62545	21944	16530	03878	07516	95715	02536	[illegible]

1. How would you proceed ?
2. How many such random numbers may one pick from this table ?

Choose bytes from 650 random Megabytes

In 1995 *George Marsaglia* prepared a **CDROM** with 650 Megabytes of "*white and black*" noise, generated by combining the output of a noise-diode circuit with deterministically scrambled rap music.

```
..$ hexdump bits.01
0000000 1d3f 7cc4 1330 16b2 fd2d 1c01 7963 5f10
0000010 f813 c907 27cd f625 af78 25e7 17c0 6a05
0000020 593c 1ce4 293a 86af 1109 cee3 f39b 429b
0000030 2b62 d0fc 2482 0eb1 a3d8 d677 3e9f 0931
0000040 fe0c 403d 9799 4640 4951 3f6e 4697 4ed0
0000050 984f 6f3a 6ef4 3510 bfb6 2cca 6601 cda7
0000060 a063 b7af 2ba7 28b7 3563 544f 5ced 9a69
0000070 aa41 7a14 c2be c3d8 a92c 0f20 c218 3471
0000080 d67c 49db e59e aae1 5fc2 fdf9 ba18 f877
0000090 2ffe 1601 1165 62c1 9f16 d24e 3104 ded0
00000a0 24ca da7a 7b39 1561 d5d9 34b5 2b3f dd13
00000b0 6adb 058d 059a c0c9 9c10 5057 7017 84a8
00000c0 6257 f049 0b0e c912 cb59 4087 1a34 a2be
00000d0 bbd0 bcfa 0135 c0d5 e74f bdc9 a07a beaa
...
...|
```

See http://stat.fsu.edu/pub/diehard/

Early computing techniques

John von Neumann suggests in 1949 to recursively extract the middle digits from the square of a random number.
For instance, to generate 10-digit numbers and the previous value was 5772156649, we square it to get :

$$33317792380594909201;$$

the next number is hence 7923805949.

The sequence is evidently not random, but it appears to be so. Depending on the starting point it may, however, quickly end in a short cycle of repeating the same sequence of numbers.

Exercise

Generate in Python a sequence of random 4 middle digits from squared numbers of length 8, starting with the seed = 2608.

The linear congruential generator LCG

By far the most popular random number generator in use until recently, is based on a linear congruential recursion $\langle X_n \rangle$ (D. H. Lehmer 1949) :

$$\boxed{X_{n+1} \;=\; (aX_n + c) \bmod m, \quad n \geq 0}$$

with four magic numbers :

m,	the modulus ;	$0 < m$.
a,	the multiplier ;	$0 \leq a < m$.
c,	the increment ;	$0 \leq c < m$.
X_0,	the starting value ;	$0 \leq X_0 < m$.

Exercise

i) *Generate $\langle X_n \rangle$ with $m = 10$ and $X_0 = a = c = 7$.*

ii) *Generate $\langle X_n \rangle$ with $m = 256$, $X_0 = 0$, $a = 137$, and $c = 187$. Scatterplot X_n versus X_{n-1} for $n = 2, ..., 256$.*

Maximal period of the MLCG

With increment $c = 0$, the maximal period of the multiplicative linear congruential generator (MLCG) is $m - 1$ when m is prime and $a > 1$ is a primitive element modulo m.
In this case, a is a generator of the cyclic group $(\mathbb{Z}_m^+, \cdot)$, where $\mathbb{Z}_m^+$ represents $\mathbb{Z}$ - $\{0\}$ and $\cdot$ represents arithmetic multiplication modulo m.
Indeed, consider the random sequence output by an LCG in this case :

$$\langle X_m \rangle = \left[\frac{x_0}{m}, \frac{a^1 x_0}{m}, \frac{a^2 x_0}{m}, ..., \frac{a^{m-1} x_0}{m}, \frac{a^m x_0}{m} \right]$$

Since $a^i \neq a^j$ for all $1 \leqslant i \neq j \leqslant m - 1$, the first $m - 1$ elements of $\langle X_m \rangle$ are all different, and $(a^m x_0)/m = (a^0 x_0)/m = x_0/m$. The sequence starts repeating itself from that point on.

LCG with maximum period length m

Theorem (Greenberger 1961, Hull and Dobell 1962)

The linear congruential sequence defined by m, a, c, and X_0 has period length m if and only if :

(i) *c is relatively prime to m ;*

(ii) *$a - 1$ is a multiple of p, for every prime p dividing m ;*

(iii) *$a - 1$ is a multiple of 4, if m is a multiple of 4.*

Comment

LCGs are obsolete today. And better generators based on register shifts and xor operations, like the Mersenne Twister – based on a matrix linear recurrence over a finite binary field $\mathbb{F}_2$ – which produces 53-bit precision floats and has a period of $2^{19937} - 1$, have replaced them in most softwares.

Generic RNG structure

Witout loss of generality, all Random Number Generators (RNGs) can be described as structure of the form (S, T, τ, ξ, x_0), where

$$\begin{aligned} S &= \text{state space} \\ T &= \text{output space} \\ \tau : S \to S &= \text{transition function} \\ \xi : S \to T &= \text{output function} \\ x_0 &= \text{seed.} \end{aligned}$$

The "random" sequence $[u_0, u_1, ...]$ generated in T is defined as $u_i = \xi(x_i)$, for $i \geqslant 0$, where $x_i = \tau(x_{i-1})$ for $i > 0$.

Example (Linear Congruential Generator)

For instance, in the case of the previous LCG, $S = \mathbb{Z}_m$, $T = [0, 1)$, $\tau(x) = (ax + c) \pmod m$, and $\xi(x) = x/m$. The magic numbers : a (the multiplier) and c (the increment) are in $\mathbb{Z} - \{0\}$, whereas m (the modulus) is in $\mathbb{N} - \{0\}$.

Multiple recursive generators

Constructing RNGs with longer periods than the linear congruential generators is possible when using a recursion of higher order.
Let $k \geqslant 1$ and m be prime. A multiple recursive generator (MRG) is an RNG with $S = \mathbb{Z}_m^k$, and state $\mathbf{y}_i = (x_i, ..., x_{i-k+1})$ at step i evolves through the recurrence :

$$x_i = \tau(\mathbf{y}_{i-1}) = (a_1 x_{i-1} + ... + a_k x_{i-k}) \mod m$$

where $a_i \in \mathbb{Z}$ for $j = 1, ..., k$ with $a \neq 0$ and the output $\xi(y_i)$ is given by x_i/m.
The potential period of an MRG is $m^k - 1$, obtained when the characteristic polynomial $P(z)$ of the recurrence is primitive over $\mathbb{F}_m$. That is, the smallest integer r for which $z^r \equiv 1 \mod P(z)$ is $m^k - 1$.

Examples (Simple MRGs)

1. The multiplicative congruential generator (MRG) where $k = 1$.
2. The additive lagged-Fibonacci generator, where the transition function is given by : $x_i = (x_{i-r} + x_{i-k}) \mod m$. Proposed magic numbers are $r = 24$, $k = 55$ and $m = 2^{24}$ (Mitchell&Moore 1958).

Multiple recurrences modulo 2

Because of the binary nature of all data and computations, it is opportune to use a binary state space $\{0, 1\}^k$ and implement transition functions on $\mathbb{F}_2$, where multiplication and division are implemented with register shifts ($<<, >>$) and addition modulo 2 is implemented with the `xor` operator.

Example (Tausworthe generator)

The linear feedback shift register (LFSR), proposed by Tausworthe (1965), is a MRG with $S = \mathbb{Z}_2^k$ and transition function :

$$x_i = (a_1 x_{i-1} + ... + a_k x_{i-k}) \mod 2$$

where $a_i \in \{0, 1\}$ for $i = 1, ..., k$, and the output value $u_i = \sum_{j=1}^{L} [(x_{i\nu+j-1}) 2^{-j}$ with ν (the step size) and L (the word length) being positive integers.
The Tausworthe generator has a maximal period $\rho = 2^k - 1$ if the transition function has this period ρ and $\gcd(\rho, \nu = 1)$.

Generalized feedback shift register (GFSR)

The Tausworthe generator has been generalized by replacing the "bits" x_i by vectors $\mathbf{x}_i$ of L bits. The state $\mathbf{y}_i$ is then defined as kL bits vectors $(\mathbf{x}_i, ..., \mathbf{x}_{i-k+1})$
The transition function is replaced by a recurrence of the form :

$$\mathbf{x}_i = (a_1\mathbf{x}_{i-1} + ... + a_k\mathbf{x}_{i-k}) \mod 2$$

where $\mathbf{x}_i = (x_{i,1}, ..., x_{i,L})$ and the output value is :

$$u_i \ = \ \sum_{j=1}^{L}(x_{i,j}) \cdot 2^{-j}$$

The maximal period of this generator is still $2^k - 1$, while the period could potentially be of the size of the state space $(|S| - 1 \ = \ 2^{kL} - 1)$.

Twisted generalized feedback shift register (TGFSR)

A way to further increase the potentially maximal period of a GFSR goes by generalizing the recurrence defining the transition function. To illustrate this construction it is useful to reformulate the GFSR transition function in matrix notation : $\mathbf{x}_i \ = \ A\mathbf{x}_{i-1}$, where $\mathbf{x}_i$ are vectors of kL bits and A is a $kL \times kL$ matrix of the form :

$$A = \begin{pmatrix} 0 & I_L & 0 & \dots & 0 \\ 0 & 0 & I_L & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & \dots & I_L \\ a_k I_L & a_{k-1} I_L & a_{k-2} I_L & \dots & a_1 I_L \end{pmatrix}$$

where I_L is the $L \times L$ identity matrix.
The TGFSR replaces the I_L matrix in the last row by more general matrices. Furthermore the output value u_i of the GFSR is *tempered* with a series of left and right register shifts.

The Mersenne Twister generator

The actually most popular and used TGFSR is the *Mersenne Twister* MT19937 generator with magic numbers $k = 624$ (19×64) and word size $L = 32$. It attains a period of $2^{kL-L+1} - 1$ $= \ 2^{19937} - 1$; very close to the theoretical maximal period of $2^{kL} - 1 \ = \ 2^{19968} - 1$.
All these multiple recursive RNG designs with state and output space $\mathbb{F}_2^{k \times L}$, for some positive integer k and word size L, can be easily described in a generic matrix notation :

$$\begin{aligned} \mathbf{x}_i &= \tau(\mathbf{x}_{i-1}) \ = \ A\mathbf{x}_{i-1}, \\ \mathbf{y}_i &= \xi(\mathbf{x}_i) \ = \ \mathbf{x}_i B, \end{aligned}$$

where $0 \leqslant i \leqslant k - 1$; A is the $kL \times kL$ transition matrix ; B is the $kL \times L$ output matrix, both with elements in $\mathbb{F}_2$. The output value $u_i \in [0, 1]$ is computed as follows :

$$u_i \ = \ \sum_{j=1}^{L}(y_{i,j-1}) \cdot 2^{-j}.$$

Definition (From Numerical Recipes p.340)

A home-made generator of random numbers should ideally verify the following methodological principles :

1. The procedure – deterministic or not – that produces a random sequence of numbers should be different from, and – in all measurable respects – statistically uncorrelated with, the procedure that uses its output ;
2. Any two different random generating procedures ought to produce statistically the same results if used similarly in a scientific investigation or application ;
3. A same sequence of random numbers may be regenerated Ad libitum for testing and debugging purposes.

Traps to watch for

Many out-of-date and inferior methods for generating random numbers remain in general use. Therefore :

- Never use a generator principally based on a linear congruential generator (LCG) or a multiplicative linear congruential generator (MLCG).
- Never use a generator with a period less than $\sim 2^{64} \approx 2 \times 10^{19}$, or any generator whose period is unknown to you.
- Note that in your scientific reports, when using random numbers, you should always mention the generator and its period.
- Never use a generator that warns against using its low-order bits. This indicates an obsolete algorithm (usually a LCG).
- Never use the built-in generators in the C and C++ language, especially `rand` and `srand`. They have no standard implementation and are often of bad quality.

Best practice

Recommendations for constructing a random number generator :

- An acceptable random generator must combine at least two ideally unrelated methods.
- The methods combined should evolve independently and share no state.
- The combination should be by simple operations that do not produce results less random than their operands.

  Reference : *Numerical Recipes : The Art of Scientific Computing (3rd Ed.), W H Press, S A Teukolsky, W T Vetterling & B P Flannery, Cambridge University Press 2007, Chapter 7, Random Numbers, pp. 340 – 418.*

Combining bitwise operators

64-bit Xor ($\oplus$) and bit shifts ($<<$, $>>$)

state :	x (unsigned 64-bit)
initialize :	$x \neq 0$
update :	$x \leftarrow x \oplus (x >> a_1)$
	$x \leftarrow x \oplus (x << a_2)$
	$x \leftarrow x \oplus (x >> a_3)$
can use as random :	x (all bits)
period :	$2^{64} - 1 \;=\; 1.8446744073709551615 \times 10^{19}$

Triples of magic numbers (a_1, a_2, a_3), that deliver a full period are a.o. $(21, 35, 4)$, $(20, 41, 5)$, and $(17, 31, 8)$. The MT19937 generator uses, for instance, this approach as tempering functions, with a quadruple of magic numbers $(11, 7, 15, 18)$.

Example of simple and fast combined generator

Combining Xor, shifts and an LCG

state :	x (unsigned 64-bit)
initialize :	$x \neq 0$ (default : 4101842887655102017)
update :	$x \leftarrow x \oplus (x >> 21)$
	$x \leftarrow x \oplus (x << 35)$
	$x \leftarrow x \oplus (x >> 4)$
	$x \leftarrow 2685821657736338717 \cdot x \pmod{2^{64}}$
can use as random :	x (all bits)
period :	1.8×10^{19}

Source : Numerical recipes, `Ranq1`, p.351.

Testing equidistribution

Let $\langle U_n \rangle = [u_0, u_1, u_2, ...]$ be a sequence of random numbers from the float interval $[0.0; 1.0)$ apparently generated in a uniformly manner.
To test the quality of the random generator, we consider the auxiliary sequence $\langle Y_n \rangle = [y_0, y_1, y_2, ...]$ defined by the rule $y_n = \lfloor d \times u_n \rfloor$, where d is a positive integer – usually 64, 100, or 128 – also called the *discrete grain* of the generator.
When sequence $\langle U_n \rangle$ is indeed uniformly distributed, we will observe a sequence $\langle Y_n \rangle$ of equidistributed integers between 0 and $d - 1$.

> A generator produces a good uniform random sequence $\langle U_n \rangle$ if, for a large grain d and $n \rightarrow \infty$, the relative frequency $f(i)$ of each integer i from 0 to $d - 1$ in $\langle Y_n \rangle$ converges (but not suspiciously fast) to $1/d$.

The quality of a given random generator may now be assessed with a two-tailed Chi-square test of difference between the empirical $f(i)$ distribution and the theoretical uniform $1/d$ distribution. Below 5% or above 95% differences indicate the likeliness of a suspicious non-randomness in $\langle U_n \rangle$.

Serial test

- We reconsider the auxilliary $\langle Y_n \rangle$ sequence with discrete grain d and count the number of times the pair $(y_{2j}, y_{2j+1}) = (q, r)$ occurs, for $0 \leq j < n/2$, $q \neq r$ and $0 \leq q, r \leq d$.
- These counts are to be made for each pair of integers (q, r) with $0 \leq q, r \leq d$, and the Chi-square test is applied to these $k = d^2$ categories with theoretical uniform relative frequency $1/d^2$ in each category.
- To keep the length n of the random sequence large compared to k, d will be chosen of smaller value than for the equidistributional test.

Gap test

- Another test is to examine the length of "gaps" between occurences of u_j in a certain range. If α and β are two real numbers with $0 \leq \alpha < \beta \leq 1$, we want to consider the lengths of consecutive subsequences $[u_j, u_{j+1}, ..., u_{j+r}]$ in which the consecutive r values u_{j+k}, for $k = 1, ...r$, remain between α and β. This situation will be counted as a gap of length r.
- With given values α and β and a maximal gap length t, let C_r for $r = 0, ..., t-1$ count the occurences of gaps of length $0, ..., t-1$, and C_t the gaps of length $r \geq t$. If $p = \beta - \alpha$, the theoretical counts for each gap length r, is $p_r = p(1-p)^r$ for $0 \leq r < t - 1$ and $p_t = (1-p)^t$.
- Again, a Chi-square test, comparing the C_r with the p_r distribution may be used in order to assess the likeliness of a suspicious non-randomness of the gap lengths observed in the sequence $\langle U_n \rangle$.

Coupon collector's test

- This test relates the frequency test to the previous gap test. We use the auxiliary sequence $\langle Y_n \rangle$ and we observe the lengths of subsequences $y_{j+1}, y_{j+2}, \ldots, y_{j+r}$ that are required to get a complete set of integers – a coupon collector seqment – from 0 to $d-1$.
- With a given maximal subsequence length t, let C_r for $r = d, \ldots, t-1$ count the occurences of coupon collector segments of length $d, d+1, \ldots, t-1$, and C_t the segments of length $r \geq t$.
- The theoretical count for each coupon collector segment of length r, is

$$p_r = \frac{d!}{d^r}\left\{ {r-1 \atop d-1} \right\}, \quad d \leq r < t-1; \quad p_t \; = 1 - \frac{d!}{d^r}\left\{ {r \atop d} \right\}.$$

- Similarly, a Chi-square test, comparing the empirical C_r with the theoretical p_r distribution, may be used in order to assess the likeliness of a suspicious non-randomness of the coupon collector segments.

Up and down runs test

- A sequence $\langle U_n \rangle$ of uniform random numbers may also be tested for "*runs up*" and "*runs down*" segments, by examining the length of monotone portions of it. Let $[u_{j+0}, u_{j+1}, \ldots, u_{j+r}]$ be a subsequence of length r such that either $u_{j+0} \geq u_{j+1} \geq \ldots \geq u_{j+r}$, or, $u_{j+0} \leq u_{j+1} \leq \ldots \leq u_{j+r}$.
- Given a maximal subsequence length t, let C_r for $r = 1, \ldots, t-1$ count the occurences of separated monotone, either up, or, down runs of length $1, 2, \ldots, t-1$, and C_t the same runs of length $r \geq t$.
- Assuming that a monotone run of length r occurs with probability $1/r! - 1/(r+1)!$, the theoretical relative count for each length r, gives $p_r = 1/r! - 1/(r+1)!$ for $r < t$ and $p_t = 1/t!$.
- And, again, we may use a Chi-square test, comparing the empirical C_r with the theoretical p_r distribution, for assessing the likeliness of a suspicious non-randomness of "*runs up*" or "*runs down*" segments.

Generating random floats with Python3

`random` is the basic module for generating random numbers in Python. Python3 uses the *Mersenne Twister* as the core generator.
Some code for generating random floats :

```
Python 3.2.3 (default, Oct 19 2012, 19:53:57)
Type "help", "copyright", "credits" or "license"
    for more information.
>>> from random import seed, random, uniform
>>> seed(100) # Setting X_0
>>> print('random number on (0,1):', random())
random number on (0,1): 0.1456692551041303
>>> print('random number on (-1,1):',\
            uniform(-1,1))
random number on (-1,1): -0.0901459909719573
```

▷ http://docs.python.org/library/random.html

Random integers and choice

- randrange([start], stop[, step])
  Returns a randomly selected element from range(start, stop, step).

```
print([randrange(-100,100,5) for i in range(10)])
[-85, 90, -25, 95, 60, -10, 85, 5, 65, 5]
```

- randint(a, b)
  Returns a random integer N such that $a \leqslant N \leqslant b$.

```
print([randint(0,5) for i in range(10)])
[2, 0, 5, 5, 1, 2, 1, 2, 1, 1]
```

- choice(seq)
  Returns a random element from the non-empty sequence seq. If seq is empty, raises IndexError.

```
seq = ['a','b','c','d','e','f']
print([choice(seq) for i in range(7)])
['e', 'c', 'd', 'd', 'a', 'a', 'c']
```


Shuffling sequences and drawing samples

- shuffle(x[, random])
  Shuffles the sequence x in place. The optional argument random is a 0-argument function returning a random float in $[0.0, 1.0)$; by default, this is the function random().Note that for even rather small len(x), the total number of permutations of x is larger than the period of most random number generators ; this implies that most permutations of a long sequence can never be generated.
- sample(population, k)
  Returns a k length list of unique elements chosen from the population sequence. Used for random sampling without replacement. Example : sample(xrange(10000000), 60).

Comparing real random and pseudo-random number sequences

Exercise

The Python3 random module provides the class SystemRandom that uses the os.urandom() function for generating 'real' random numbers from electronic sources provided by the operating system.

1. *Generate two sequences of 10 000 random floats in the range [0.0, 1.0), one, from the systemRandom generator and, the other, from the standard Mersenne Twister generator.*
2. *Does there appear a noticeable difference in randomness quality between both sequences ?*

Random numbers from a MLCG

Exercise

1. *Develop in Python a linear congruential generator for random floats between 0 an 1 of the following type :*

$$x_0 = seed \quad (1)$$
$$x_n \equiv a \cdot x_{n-1} + c \pmod{m} \quad (2)$$

where a, c, m and seed may be given at run time.

2. *Generate a csv data file containing a sample of 10000 random numbers obtained with your generator when using each one of the following sets of magic numbers :*
   - i. $a = 3141592653$, $c = 2718281829$, $m = 2^{35}$, $seed = 0$
   - ii. $a = 2^7 + 1$, $c = 1$, $m = 2^{35}$, $seed = 0$
   - iii. $a = 23$, $c = 0$, $m = 10^8 + 1$, $seed = 47594118$
   - iv. $a = 2^{18} + 1$, $c = 1$, $m = 2^{35}$, $seed = 314159265$
3. *Test the quality of the randomness of the random sequences obtained with the different settings of the magic numbers above.*

Discrete empirical random laws

Exercise

You are requested to draw a sample of 1000 random integers in the range $[0;9]$ *along the following empirical probability distribution :*

0	*1*	*2*	*3*	*4*
0.0478	*0.3349*	*0.2392*	*0.1435*	*0.0957*

5	*6*	*7*	*8*	*9*
0.0670	*0.0478*	*0.0096*	*0.0096*	*0.048*

1. *Write a Python program for generating this sample.*
2. *Compare the sample distribution with the empirical one.*

Random triangular floats

Exercise (Triangular law)

You are requested to draw a sample of 1000 random floats in $[-1.0;1.0]$ *from a triangular distribution with mode* $= 0.0$ *:*

1. *Write a Python program for generating this sample using a uniform random float generator.*
2. *Verify the triangular aspect of the sample distribution.*

Bibliography

Christiane Lemieux, *Monte Carlo and Quasi Monte Carlo Sampling.* Springer series in Statistics, New York 2009, Chapter 3 : Pseudorandom Number Generators pp 57–86.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes : The Art of Scientific Computing.* Third Edition, Cambridge University Press, Cambridge UK 2007, Chapter 7 Random Numbers pp 340–418.

Donald E. Knuth, *The Art of Computer Programming : Seminumerical Algorithms.* Vol. 2, Third Edition, Addison-Wesley, Boston, 1998, Chapter 3 Random Numbers pp 1–193.

M. Matsumoto and T. Nishimura (1998), "Mersenne twister : a 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Transactions on Modeling and Computer Simulation* 8 (1) : 3–30