

# Computational Statistics with Python

## Topic 4: Random number and random variable generators

**Expected lecture time: 2 hours**

Giancarlo Manzi

Department of Economics, Management and Quantitative Methods

University of Milan, Milan, Italy

## Random numbers

- In statistics randomness is vital.
- The importance of the availability of random values generated from random variables is fundamental especially in simulation methods.
- Computational statistics is based on random numbers and variables.
- Two definitions of random numbers:
  - Classic;
  - Modern.

## Classical definition of a random number

- Ideally, in the classic definition a random number should come from a Discrete Uniform distribution (also called Rectangular) with parameter 10.

$$X \sim \text{Unif}(10),$$

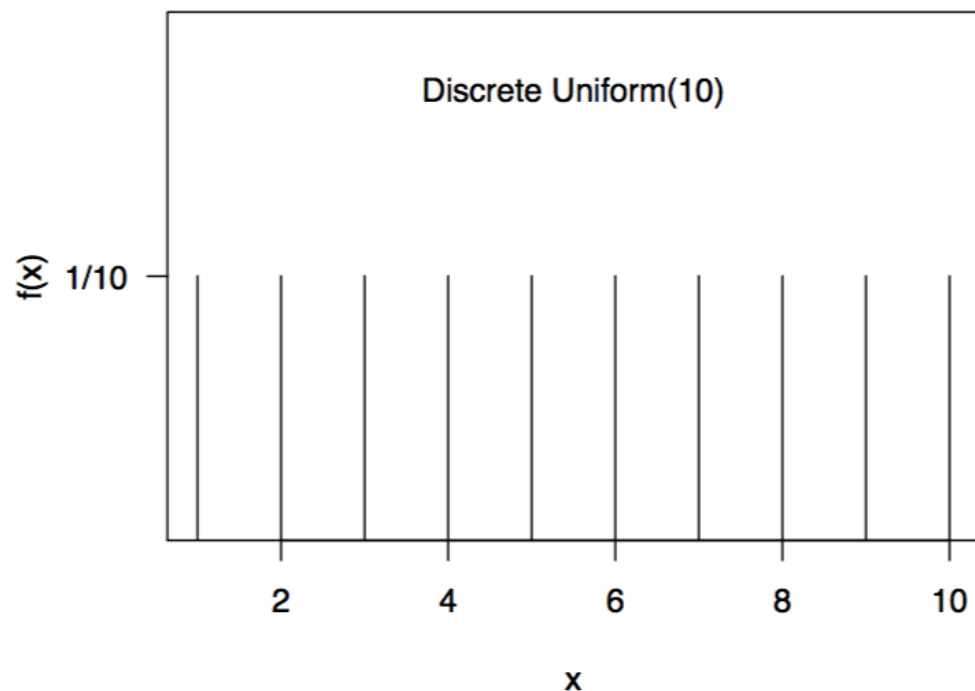
with range  $\{0, 1, \dots, 9\}$ .

- Recall that a random variable  $X$  is Discrete Uniform if its pdf  $f(x)$  is:

$$f(x) = \begin{cases} \frac{1}{N}, & \text{if } x = 1, 2, \dots, N \\ 0, & \text{otherwise} \end{cases}$$

and we write  $X \sim \text{Unif}(N)$ .

- In the following plot a  $\text{Unif}(10)$  is represented:



- **Classic definition of random numbers.**
- A sequence of random numbers is formed by integer digits:  $\{0, 1, \dots, 9\}$  generated in sequence independently and with the same probability\*.

## Modern definition of a random number

- Ideally, in the modern definition a random number should come from a Continuous Uniform distribution instead, with parameters 0,1:

$$X \sim Unif(0, 1).$$

- **Modern definition of random numbers.**
- A sequence of random numbers is formed by real numbers in the interval  $(0, 1)$  generated in sequence, independently and with the same probability\*.

## Traditional RNG systems

- Historically, mechanical devices have always been used to generate true random numbers.
- Lotto games
- Dice.
- Coins.
- Cards.
- Etc.
- For example the long sequence of lotto results are good example of classic random numbers (they have been recorded in random number tables for decades).

56692	63426	94166	37271	60586
72809	94634	27233	79840	26188
36861	21975	62251	36443	48684
12925	56772	24233	39203	64140
33918	71118	72411	82767	88607
57520	79459	10040	25613	17279
33464	95868	80075	15628	70150
35542	69109	29990	46032	56826
54073	71383	18666	56308	35904
37850	20616	79818	29048	68501


# Advantages and disadvantages of traditional RNG systems

- Advantages:
  - Produce *true* random numbers.
- Disadvantages:
  - Sensitivity to events external to the random event (how about frauds in Italian lotto?).
  - Long amount of time needed to produce a sufficiently long sequence.
  - Storage procedure in a computer system: long time needed and computational power not completely exploited.

## "Natural" RNG systems

- Recording of some natural process (in physics, for example) postulated by theory.
- Example: Rand corporation (1955):
  - Code of an electronic disturbance.
  - One million random number produced.
  - Recorded in a random number tables.

## A million random digits....



**WIKIPEDIA**  
The Free Encyclopedia

[Main page](#)  
[Contents](#)  
[Featured content](#)  
[Current events](#)  
[Random article](#)  
[Donate to Wikipedia](#)  
[Wikipedia store](#)

Interaction

[Help](#)  
[About Wikipedia](#)  
[Community portal](#)  
[Recent changes](#)  
[Contact page](#)

Tools

[What links here](#)  
[Related changes](#)  
[Upload file](#)  
[Special pages](#)  
[Permanent link](#)  
[Page information](#)  
[Wikidata item](#)  
[Cite this page](#)

Print/export

[Create a book](#)

### *A Million Random Digits with 100,000 Normal Deviates*

From Wikipedia, the free encyclopedia

***A Million Random Digits with 100,000 Normal Deviates*** is a [random number book](#) by the [RAND Corporation](#), originally published in 1955. The book, consisting primarily of a [random number table](#), was an important 20th century work in the field of [statistics](#) and [random numbers](#). It was produced starting in 1947 by an electronic simulation of a [roulette wheel](#) attached to a [computer](#), the results of which were then carefully filtered and tested before being used to generate the table. The RAND table was an important breakthrough in delivering random numbers, because such a large and carefully prepared table had never before been available. In addition to being available in book form, one could also order the digits on a series of [punched cards](#).

The table is formatted as 400 pages, each containing 50 lines of 50 digits. Columns and lines are grouped in fives, and the lines are numbered 00000 through 19999. The [standard normal deviates](#) are another 200 pages (10 per line, lines 0000 through 9999), with each deviate given to three decimal places. There are 28 additional pages of [front matter](#).

The main use of the tables was in statistics and the [experimental design](#) of [scientific experiments](#), especially those that used the [Monte Carlo method](#); in [cryptography](#), they have also been used as *nothing up my sleeve numbers*, for example in the design of the [Khafre cipher](#). The book was one of the last of a series of random number tables produced from the mid-1920s to the 1950s, after which the development of high-speed computers allowed faster operation through the generation of [pseudorandom](#) numbers rather than reading them from tables.

The book was reissued in 2001 ([ISBN 0-8330-3047-7](#)) with a new foreword by RAND Executive Vice President [Michael D. Rich](#). It has generated many humorous user reviews on [Amazon.com](#).<sup>[1][2]</sup>

The digits (sequence [A002205](#) in the [OEIS](#)) begin:

10097 32533 76520 13586 34673 54876 80959 09117 39292 74945

73735	45963	78134	63873
02965	58303	90708	20025
98859	23851	27965	62394
33666	62570	64775	78428
81666	26440	20422	05720
15838	47174	76866	14330
89793	34378	08730	56522
78155	22466	81978	57323
16381	66207	11698	99314
75002	80827	53867	37797
99982	27601	62686	44711
84543	87442	50033	14021
77757	54043	46176	42391
80871	32792	87989	72248
30500	28220	12444	71840

Lines 10580–10594, columns 21–40, <sup>53</sup> from *A Million Random Digits with 100,000 Normal Deviates*

## Switching to empirically *simulate* true random numbers

- With the increasing power of computers, it has been evident that something different was needed.
- Exploiting the power of computers, a new way of obtaining random numbers has been introduced.
- The simulation consisted on creating as much randomness as possible *artificially*.
- This leads towards **pseudo-random number generation**.
- Advantages:
  - Solve the problem of long time needed to generate a sufficiently long sequence of numbers.
  - Not constrained on using pre-formatted tables.
- Disadvantages:
  - Can be more or less reliable in terms of pure randomness.
  - Need to test for randomness (see below).

## Main shared characteristics of Pseudo RNG systems

- The term '*pseudo*' stands for indicating that the generated values are indeed **not random at all** (should be called *fake* random numbers, actually)!
- Numbers produced via computation from such generators are the result of an ordered and finite series of **deterministic** operations.
- These algorithms are often recursive (iterative), so inducing sometimes an autocorrelation in the series of the produced values, *in contrast with the desired independence*, which is very, very important.

## What characteristics a PRNG must have to be considered OK

- A good PRNG must have the following characteristics:
  - To be able to produce numbers that appear to be *uniformly distributed* in the interval  $(0, 1)$ , with as *low dependence structure as possible*.
  - To be *fast* enough for simulation purposes.
  - To have *reliable* computational memory requirements.
  - To *grant reproducibility* of any generated series (this allows, for example, to validate simulation results and to apply the simulation study itself on different models/systems under the same conditions).

## Statistical tests used to evaluate PRNG performance

- We can implement a test on the uniform randomness of the values  $y_i$  generated by the PRNG using the following null hypothesis:

$$H_0: Y \sim U(0, 1),$$

or, equivalently:

$$H_0: F(Y) = y, 0 < y < 1.$$

- In this way, we test an hypothesis on the *functional form* of the cdf of  $Y$ .
- Many sophisticated tests available.
- But also many basic tests.
- Among basic tests, two tests are very popular: the *Chi-Squared* test and the *Kolmogorov-Smirnov* test.

## The Chi-squared test (cont'd)

- The observed frequencies in  $I_j$  are given by:

$$n_j = \#y_i \in I_j, i = 1, \dots, n.$$

- From basic statistics, we know that the test statistic:

$$x^2 = \sum_{j=1}^k \frac{(n_j - n_j^*)^2}{n_j^*}$$

is distributed  $\chi^2$  with  $k - 1$  degrees of freedom.

- Therefore, the larger  $x^2$ , the higher the probability of rejecting  $H_0$ .
- Note that if the size of intervals  $I_j$  is  $\frac{1}{k}$ , then the test statistic is:

$$x^2 = \frac{k}{n} \sum_{j=1}^k (n_j - \frac{n}{k})^2,$$

because in this case  $n^* = \frac{n}{k}$ .

## The Kolmogorov-Smirnov (KS) test

- The KS is a *nonparametric* test that compares the distributions of two unmatched groups.
- It can also be used as a 'goodness of fit' test to assess the quality of a PRNG.
- Being non-parametric, this test makes no assumptions about the distribution of the sample being tested.
- The one-sample KS test tests the following null hypothesis against the general alternative hypothesis:

$H_0$ : the values were drawn from the  $U(0, 1)$  distribution;

$H_1$ : the values were drawn from some other distribution.

## The Kolmogorov-Smirnov (KS) test (cont'd)

- The KS test constructs an observed *empirical* CDF (again!) and contrasts this distribution against the  $U(0, 1)$  cdf.

- The empirical CDF is:

$$\hat{F}(x) = \frac{\sum_{i=1}^n \mathcal{I}[y_i \leq x]}{n},$$

where  $\mathcal{I}[\cdot]$  is the indicator function equal to one when  $y_i \leq x$ .

- We aim at finding the largest distance  $d$  between the empirical CDF  $\hat{F}(x)$  and the  $U(0, 1)$  CDF and then comparing this value  $d$  to the values in a Kolmogorov-Smirnov table: the larger this  $d$  value, the higher the probability of rejecting the null hypothesis.

## The almost infinite world of PRNG...

The currently available RNG kinds are given below. `kind` is partially matched to this list. The default is "Mersenne-Twister".

"Wichmann-Hill"

The seed, `.Random.seed[-1] == x[1:3]` is an integer vector of length 3, where each `x[i]` is in  $1:(p[i] - 1)$ , where `p` is the length 3 vector of primes, `p = (30269, 30307, 30323)`. The Wichmann-Hill generator has a cycle length of  $6.9536e12 (= \text{prod}(p-1)/4)$ , see *Applied Statistics* (1984) **33**, 123 which corrects the original article).

"Marsaglia-Multicarry":

A *multiply-with-carry* RNG is used, as recommended by George Marsaglia in his post to the mailing list 'sci.stat.math'. It has a period of more than  $2^{60}$  and has passed all tests (according to Marsaglia). The seed is two integers (all values allowed).

"Super-Duper":

Marsaglia's famous Super-Duper from the 70's. This is the original version which does *not* pass the MTUPLE test of the Diehard battery. It has a period of *about*  $4.6 \cdot 10^{18}$  for most initial seeds. The seed is two integers (all values allowed for the first seed: the second must be odd).

We use the implementation by Reeds *et al* (1982-84).

The two seeds are the Tausworthe and congruence long integers, respectively. A one-to-one mapping to S's `.Random.seed[1:12]` is possible but we will not publish one, not least as this generator is **not** exactly the same as that in recent versions of S-PLUS.

"Mersenne-Twister":

From Matsumoto and Nishimura (1998). A twisted GFSR with period  $2^{19937} - 1$  and equidistribution in 623 consecutive dimensions (over the whole period). The 'seed' is a 624-dimensional set of 32-bit integers plus a current position in that set.



# The almost infinite world of PRNG... (cont'd)

"Knuth-TAOCP-2002":

A 32-bit integer GFSR using lagged Fibonacci sequences with subtraction. That is, the recurrence used is

$$X[j] = (X[j-100] - X[j-37]) \bmod 2^{30}$$

and the 'seed' is the set of the 100 last numbers (actually recorded as 101 numbers, the last being a cyclic shift of the buffer). The period is around  $2^{129}$ .

"Knuth-TAOCP":

An earlier version from Knuth (1997).

The 2002 version was not backwards compatible with the earlier version: the initialization of the GFSR from the seed was altered. R did not allow you to choose consecutive seeds, the reported 'weakness', and already scrambled the seeds.

Initialization of this generator is done in interpreted R code and so takes a short but noticeable time.

"L'Ecuyer-CMRG":

A 'combined multiple-recursive generator' from L'Ecuyer (1999), each element of which is a feedback multiplicative generator with three integer elements: thus the seed is a (signed) integer vector of length 6. The period is around  $2^{191}$ .

The 6 elements of the seed are internally regarded as 32-bit unsigned integers. Neither the first three nor the last three should be all zero, and they are limited to less than 4294967087 and 4294944443 respectively.

This is not particularly interesting of itself, but provides the basis for the multiple streams used in package **parallel**.

"user-supplied":

Use a user-supplied generator. See [Random.user](#) for details.

## Congruential generators

- Mathematically, given a positive integer  $n$ , a congruence relationship  $\rho_n$  modulus  $n$  on  $\mathbb{Z}$  is a relationship defined as following:

$$a \rho_n b, \text{ i.e. } a \equiv b \pmod{m}.$$

- This means that, for some  $h \in \mathbb{Z}$ :

$$a - b = mh.$$

- Modern PRNG methods are often based on *recursive algebraic algorithms* which, starting from a so-called *seed* (i.e. an arbitrary chosen value or a vector of values), recursively generate new values from previously generated ones.

- Congruential generators* are based on the *modulus operator*:

$$X_{n+1} = X_n \pmod{m}.$$

- This means that, with  $m$  positive integer,  $X_{n+1}$  is the remainder of  $\frac{X_n}{m}$ .
- Modern PRNGs that use linear congruential methods (linear because we have  $(aX_n + c)$  instead of  $X_n$ ) are based on the idea that the remainder of a division - the modulo - is *irregular enough to emulate randomness*.

## An initial sub-family of congruential generators

- The generic linear congruential generator gives a sequence of pseudo-random numbers  $X_1, X_2, \dots, X_n, X_{n+1}, \dots$ , according to the following recurrent formula:  

$$X_{n+1} = (aX_n + c)(\text{mod } m)$$
 where  $a$  and  $c$  are integer constants.
- $X_{n+1}$  gives the reminder of  $\frac{aX_n + c}{m}$ ;  $m$  is the modulus.
- In order to get a sequence of linear congruential pseudo-random numbers we need a "seed"  $X_0$ , the modulus  $m$ , and the two constants  $a$  and  $c$ .
- The number of iterations needed to obtain a value equal to the initial seed is called the *period* of the linear congruential generator.
- When  $c \neq 0$ , the formula above is a *mixed-congruential generator*.
- When  $c = 0$ , the formula above is a *multiplicative-congruential generator* (Lehmer, 1951).

## An example

- For convenience, write  $x_n \equiv_m (ax_{n-1} + c)$ , so that the module  $m$  is explicitly visible in the expression.
- Parameters:  $m = 9$ ;  $c = 7$ ;  $a = 10$ .
- Seed:  $x_0 = 2$ .
- We have:  $x_1 \equiv_9 (10 \times 2 + 7) \equiv_9 0$ ;
  - $x_2 \equiv_9 (10 \times 0 + 7) \equiv_9 7$ ;
  - $x_3 \equiv_9 5$ ;  $x_4 \equiv_9 3$ ;  $x_5 \equiv_9 1$ ;  $x_6 \equiv_9 8$ ;  $x_7 \equiv_9 6$ ;  $x_8 \equiv_9 4$ ;  $x_9 \equiv_9 2$ .
- At the 9th iteration we obtain the seed value, so the period is 9.

## Another example

- If  $X_0 = 1$ ,  $a = 5$ ,  $c = 1$ , and  $m = 16$ , we get:

**Draw**

---

$$X_0 = 1$$

$$X_1 = 6$$

$$X_2 = 15$$

$$X_3 = 12$$

$$X_4 = 13$$

$$X_5 = 2$$

$$X_6 = 11$$

$$X_7 = 8$$

$$X_8 = 9$$

$$X_9 = 14$$

$$X_{10} = 7$$

$$X_{11} = 4$$

$$X_{12} = 5$$

$$X_{13} = 10$$

$$X_{14} = 3$$

$$X_{15} = 0$$

$$X_{16} = 1$$

- ...and therefore the period is 16.

## Properties and rules of thumb

A congruential generator has a period  $m$  if and only if:

- $c$  is prime with respect to  $m$  (in the first example above  $c = 7$  is relatively prime to  $m = 9$ ).
- $a \equiv_p 1$  for every prime factor  $p$  of  $m$ .
- $a \equiv_4 1$  if 4 divides  $m$  (in the second example above  $a = 5 \equiv_4 1$  and 4 divides  $m = 16$ ).

Rules of thumb:

- A congruential generator should have period **as large as possible**, at least  $2^{30}$  (!!).
- People choose the multiplier  $a$  in order to get the desired  $m$  or  $m - 1$ .

## A bad example: the unfortunate RANDU

- A generator used widely on IBM 360/370 and PDP-11 machines (late 1960s and 1970s), defined with the following recurrence:

$$X_n = 65539X_{n-1} \pmod{2^{31}}$$

with the seed,  $X_0$ , an odd number.

- Knuth (1997) considers it as *one of the most ill-conceived random number generators ever designed*.
- For example, starting with the seed  $X_0 = 1$ , the sequence is the following:  
65539 393225 1769499 7077969 26542323 ...  
(<https://oeis.org/A096555> (<https://oeis.org/A096555>)).

# Multiply-with-carry

by George Marsaglia (called *Marsaglia multicarry* in R)

- In linear congruential generators,  $c$  is the so-called *carry*, i.e. it is a constant such that, with an initial seed  $x_0$ , the generic number  $x_n$ , subsequent to  $x_{n-1}$ , is generated considering the modulus  $m$ .
- A *lag-1* Multiply-with-carry (MWC) is one that modifies the carry *at each step*:
  - Start with an initial seed  $x_0$  and an initial carry  $c_0$ .
  - Subsequent number  $x_n$  is calculated as before but....
  - The carry is modified at each step by the following:
 
$$c_n = \left\lfloor \frac{ax_{n-1} + c_{n-1}}{m} \right\rfloor, n = 1, 2, \dots$$
  - Marsaglia suggested  $m = 2^{32}$  (Gentle, 2003).

## A quick example

- Suppose we have  $a = 6$ , a modulus  $m = 10$  and a seed  $x_0 = 4$  and an initial carry  $c_0 = 4$ .
- The generic number  $x_n$  is generated as  $x_n = (6x_{n-1} + c_n) \bmod 10$ , where  $c_n = \left\lfloor \frac{ax_{n-1} + c_{n-1}}{b} \right\rfloor, n = 1, 2, \dots$ .
- We have:

$X_n$	$c_n$
$X_0 = 4$	$c_0 = 4$
$X_1 = (6 \times 4 + 4) \bmod 10 = 8$	$c_1 = 2$
$X_2 = (6 \times 8 + 2) \bmod 10 = 0$	$c_2 = 5$
$X_3 = (6 \times 0 + 5) \bmod 10 = 5$	$c_3 = 0$
$X_4 = (6 \times 5 + 0) \bmod 10 = 0$	$c_4 = 3$
$X_5 = (6 \times 0 + 3) \bmod 10 = 3$	$c_5 = 0$
$X_6 = (6 \times 3 + 0) \bmod 10 = 8$	$c_6 = 1$

## Lag- $r$ MWC, base $b$ , with $b = 2^{32}$

- [Initial Step:] Start with the following set of random seed values:

$$x_0, x_1, \dots, x_{r-1} \text{ and an initial carry } c_{r-1} < a$$

- [Subsequent Step]: the lag- $r$  MWC sequence is formed by:

$$x_n \equiv_b (ax_{n-r} + c_{n-r}), \quad c_n = \left\lfloor \frac{ax_{n-r} + c_{n-r}}{b} \right\rfloor, \quad n \geq r.$$

- [Output sequence]:  $x_r, x_{r+1}, x_{r+2}, \dots$
- Usually,  $m = 2^{32} - 1$  is also used.

## Complementary-multiply-with-carry generators

- The carry uses the same computation, but the number is replaced by taking the complement  $(m - 1) - x_n$ :

$$x_n \equiv_m (m - 1) - (ax_{n-r} + c_{n-r}).$$

- The MWC/CMWC is said to have period ranging from  $2^{60}$  to  $2^{2000000}$ .
- Marsaglia-Multicarry is used as an option in RNG in R.

## Combined LCGs, Wichman & Hill 1982, 1984

- [Initial Step]: Length-3 vector as seed values:  $(x_0, y_0, z_0)$ , each element of the vector should be between 1 and 30,000.

- [Subsequent Step]:

$$\begin{aligned}x_n &\equiv_{30269} 171x_{n-1} \\y_n &\equiv_{30307} 172x_{n-1} \\z_n &\equiv_{30323} 171x_{n-1}\end{aligned}$$

- [Output]:

$$u_n \equiv_1 \left( \frac{x_n}{30269} + \frac{y_n}{30307} + \frac{z_n}{30323} \right)$$

- It turns out that:  $u_n \in (0, 1)$ .
- Such generator has period approximately  $7 \times 2^{12}$  (corrected in 1984) not  $2.8 \times 2^{13}$  as claimed originally in 1982.
- Wichmann-Hill is used as an option in RNG in R.

## Combined LCGs, Wichman & Hill 2006

- 64-bit integer arithmetic:

$$\begin{aligned}x_n &\equiv_{2147483579} 11600x_{n-1} \\y_n &\equiv_{2147483543} 47003x_{n-1} \\z_n &\equiv_{2147483423} 23000x_{n-1} \\t_n &\equiv_{2147483123} 33000x_{n-1}\end{aligned}$$

- Output function:

$$u_n \equiv_1 \left( \frac{x_n}{2147483579} + \frac{y_n}{2147483543} + \frac{z_n}{2147483423} + \frac{t_n}{2147483123} \right)$$

## Multiple Recursive Generators (MRGs) and Combined MRGs

- An MRG is defined by the recurrence

$$x_i \equiv_m (a_1 x_{i-1} + \dots + a_k x_{i-k})$$

with output function  $u_n = \frac{x_n}{m}$ .

- Knuth(39) (Knuth, 1998, Eq. (39)):  $a_1 = 271828183$ ,  $a_2 = 314159269$ ,  $k = 2$  and modulus  $(2^{31} - 1)$ .

- Marsa-LFIB4 (Marsaglia, 1999):\

- Recurrence:

$$x_i \equiv_{2^{32}} (x_{i-55} + x_{i-119} + x_{i-179} + x_{i-256})$$

- MRG32k3a proposed by L'Ecuyer (1996):\

- Recurrence:

$$x_i \equiv_{(2^{31}-1)} (107374182x_{i-1} + 104480x_{i-5})$$

(used by SAS).

## Some useful references on the internet for PRNGs

- Biebighauser, D. (2000). Testing Random Number Generators. University of Minnesota.
- Kennedy, T. (?) Monte Carlo Methods, Course Notes: Chapter 3, Pseudo random number generators. University of Arizona.



## References on PRNG so far

- Gentle, J.E. (2003). *Random Number Generation and Monte Carlo Methods*, Springer.
- Knuth, D. E. (1998). *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley.
- L'Ecuyer, P. (1996). Combined multiple recursive random number generators. *Operations Research*, 44(5): 816-822.
- Lehmer, D. H. (1951), Mathematical methods in large-scale computing units, *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*, Harvard University Press, Cambridge, Massachusetts. 141-146.
- Marsaglia, G. (1999). Random numbers for C: The END? Posted to the electronic billboard sci. crypt.random-numbers.
- Wichmann, B. A., Hill, I. D. (1982). Algorithm AS 183. An Efficient and Portable Pseudo-random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2): 188-190.
- Wichmann, B. A., Hill, I. D. (1984). Correction: algorithm AS 183: an efficient and portable pseudo-random number generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 33(1): 123-123.
- Wichmann, B. A., Hill, I. D. (2006). Generating good pseudo-random numbers. *Computational Statistics & Data Analysis*, 51: 1614-1622.

```
In [ ]: 1 import numpy as np
        2 n=1000
        3 # Set the seed
        4 x0=1
        5 c=1
        6 a=125
        7 module1 = 2**12
        8 module2 = 2**11
        9 alpha = 0.90
       10 vector_1 = np.ones(n)
       11 vector_1[0] = x0
       12 #import pdb; pdb.set_trace()
       13 #First type of congruential generator x_n=(125x_(n-1) + 1) mod
       14 for i in range(1,n):
       15     vector_1[i] = (a * vector_1[i-1] + c) % module1
       16 np.set_printoptions(suppress=True)
       17 print(vector_1)
```

```
In [ ]: 1 import numpy as np
        2 n=1000
        3 # Set the seed
        4 x0=1
        5 c=1
        6 a=125
        7 module1 = 2**12
        8 module2 = 2**11
        9 #Second type of congruential generator  $x_n=(125x_{n-1} + 1) \bmod$ 
       10 vector_2 = np.ones(n)
       11 vector_2[0] = x0
       12 for i in range(1,n):
       13     vector_2[i] = (a * vector_2[i-1] + c) % module2
       14 np.set_printoptions(suppress=True)
       15 print(vector_2)
```

```

In [ ]: 1 import pandas as pd
        2 import numpy as np
        3 from scipy import stats
        4 #Chi-square test (computed by hand) on first RNG
        5 # No. of intervals (also called "cells" in CS jargon)
        6 k = 10
        7 #Expected n*
        8
        9 #import pdb; pdb.set_trace()
       10 n_star = n/k
       11 n_star
       12 int_length = module1/k
       13 table1 = dict(pd.cut(vector_1,np.linspace(start = 0, stop = mod
       14 #Table1_list = list(table1.values())
       15 Table1_array = np.array(list(table1.values()))
       16
       17 n_star_vect = np.repeat(n_star, k)
       18
       19 chi1 = np.sum((Table1_array - n_star_vect)**2/n_star_vect)
       20
       21 teo_chi1 = stats.chi2.ppf(alpha, k-1)
       22
       23 if teo_chi1 > chi1:
       24     print("ACCEPT FIRST RNG")
       25 else:
       26     print("REJECT FIRST RNG")
       27 table2 = dict(pd.cut(vector_1,np.linspace(start = 0, stop = mod
       28 #Table1_list = list(table1.values())
       29 Table2_array = np.array(list(table2.values()))
       30
       31 n_star_vect = np.repeat(n_star, k)
       32
       33 chi2 = np.sum((Table2_array - n_star_vect)**2/n_star_vect)
       34
       35 teo_chi2 = stats.chi2.ppf(alpha, k-1)
       36
       37 if teo_chi2 > chi2:
       38     print("ACCEPT SECOND RNG")
       39 else:
       40     print("REJECT SECOND RNG")

```

## Lagged-Fibonacci generators

- A Fibonacci sequence is:

$$x_{i+2} = x_{i+1} + x_i.$$

- The first terms of the Fibonacci sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$$

- An idea to get a RNG from the Fibonacci numbers could be to use the modulus to reduce the sequence into a set of somewhat random numbers.
- Also, instead of combining consecutive terms, we combine terms at some greater distance apart, so the *lagged Fibonacci congruential generator* is:

$$x_n \equiv_m (x_{n-j} + x_{n-k}).$$

- Let  $x_0 = 3, x_1 = 5, x_2 = 17, m = 7, j = 2, k = 3$ , we have:  
 $x_3 \equiv_7 (5 + 3) = 1; x_4 \equiv_7 (17 + 5) = 1; x_5 \equiv_7 (1 + 17) = 4; x_6 \equiv_7 (1 + 1) = 2$

## Lagged-Fibonacci Generators, LFib( $m, r, k, \circ$ )

- It has the following recurrence:

$$x_n \equiv_m (x_{n-j} \circ x_{n-k})$$

where  $\circ$  can be one of  $+, -, \times$ , XOR (bitwise exclusive OR).

- An XOR operator takes two bit patterns of equal length and performs the logical inclusive OR on the following rules: if both bits in the same position are 0 then the result in that position is 0, if both bits in the same position are 1, then the result in that position is 0, otherwise is 1. Example: 0101 XOR 0011 = 0110.
- An alternative notation is LFib( $m, j, k, \circ$ ).
- Unix/Linux's random(): LFib( $2^{32}, 7, 3, +$ ), LFib( $2^{32}, 15, 1, +$ ), LFib( $2^{32}, 31, 3, +$ ), and LFib( $2^{32}, 63, 1, +$ ) with the least significant bit of each random number dropped.
- Knuth-TAOCP-2002: LFib( $2^{30}, 100, 37, +$ ), used as an option in the RNG used by R.

## Mersenne numbers and Mersenne primes

- The Mersenne numbers (or for our purposes the Mersenne prime numbers), named after Marin Mersenne (a French monk who began the study of these numbers in the early 17th century), is a positive integer that is one less than a power of two:

$$M_p = 2^p - 1$$

- Let's see a sequence for  $p = 1, 2, \dots$ :

$$M_1 = 2^1 - 1 = 1$$

$$M_2 = 2^2 - 1 = 3$$

$$M_3 = 2^3 - 1 = 7$$

$$M_4 = 2^4 - 1 = 15$$

$$M_5 = 2^5 - 1 = 31$$

$$M_6 = 2^6 - 1 = 63$$

$$M_7 = 2^7 - 1 = 127$$

...

- The first Mersenne *prime numbers* are: 1, 3, 7, 31, 127, ...

## Mersenne numbers: binary notation

- In binary notation Mersenne numbers are:

$$M_1 = 2^1 - 1 = 1$$

$$M_2 = 2^2 - 1 = 11$$

$$M_3 = 2^3 - 1 = 111$$

$$M_4 = 2^4 - 1 = 1111$$

$$M_5 = 2^5 - 1 = 11111$$

$$M_6 = 2^6 - 1 = 111111$$

$$M_7 = 2^7 - 1 = 1111111$$

...

## Largest Mersenne Prime to Date: The GIMPS Project

- The website <https://www.mersenne.org> (<https://www.mersenne.org>) is a distributed computing is devoted to finding Mersenne primes. On December 21st, 2018, the largest Mersenne prime so far was found:  $M_{82,589,933}$  (half million larger than the previous Mersenne prime which had 23,249,425 digits).

## The Mersenne Twister algorithm

- It's a quite complicated pseudo uniform random number generator developed in 1997 by Matsumoto and Nishimura based on the twisting principle and with a period of  $2^{19937} - 1$  (no explanation given here).
- Mersenne-Twister is the default RNG in R and is the core of the random Numpy module in Python.
- Its name derives from the fact that its period length is chosen to be a Mersenne prime.

## The middle-square algorithm

- A simple recursive algebraic algorithm is the *middle-square algorithm* proposed by Von Neumann:
- Choose  $k$  (the no. of digits of the generated values),  $X_0$  (seed) and  $N$  the number of values to be generated.
- For (i:1 to N): compute  $X_{i-1}^2$  (with  $2k$  digits) and take  $X_i$  as the central  $k$  digits of  $X_{i-1}^2$ .
- $X_0, X_1, \dots, X_N$  is the sequence of the random generated values.
- Example:  $X_0 = 54321$ ;  $X_0^2 = 2950771041$ ;  $X_1 = 50771$ ;  
 $X_1^2 = 2577694441$ ;  $X_2 = 77694$ ; etc.

## References

- Lewis, T. G., Payne, W. H. (1973). Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20: 456-468.
- Marsaglia, G., Anathanarayanan, K., PAUL, N. (1973). How to use the McGill random number package SUPER-DUPER. Tech. rep., School of Computer Science, McGill University, Montreal, Canada.
- Matsumoto, M., Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random generator, *ACM Transactions on Modeling and Computer Simulation*, 8: 3?30.
- Tausworthe, R. C. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*. 19: 201-209.

## Random Variable Generation (RVG)

- RVG relies on the possibility of producing (via computation) a supposedly endless flow of random variables (usually iid) from specific densities.
- Here we are concerned with the part of computational statistics which deals with producing random variable values using the characteristics of the uniform (0,1).
- We look at some basic methodology that can, starting from these simulated uniform random variables, produce random variables from both standard and non-standard distributions.

## RVG: The uniform distribution

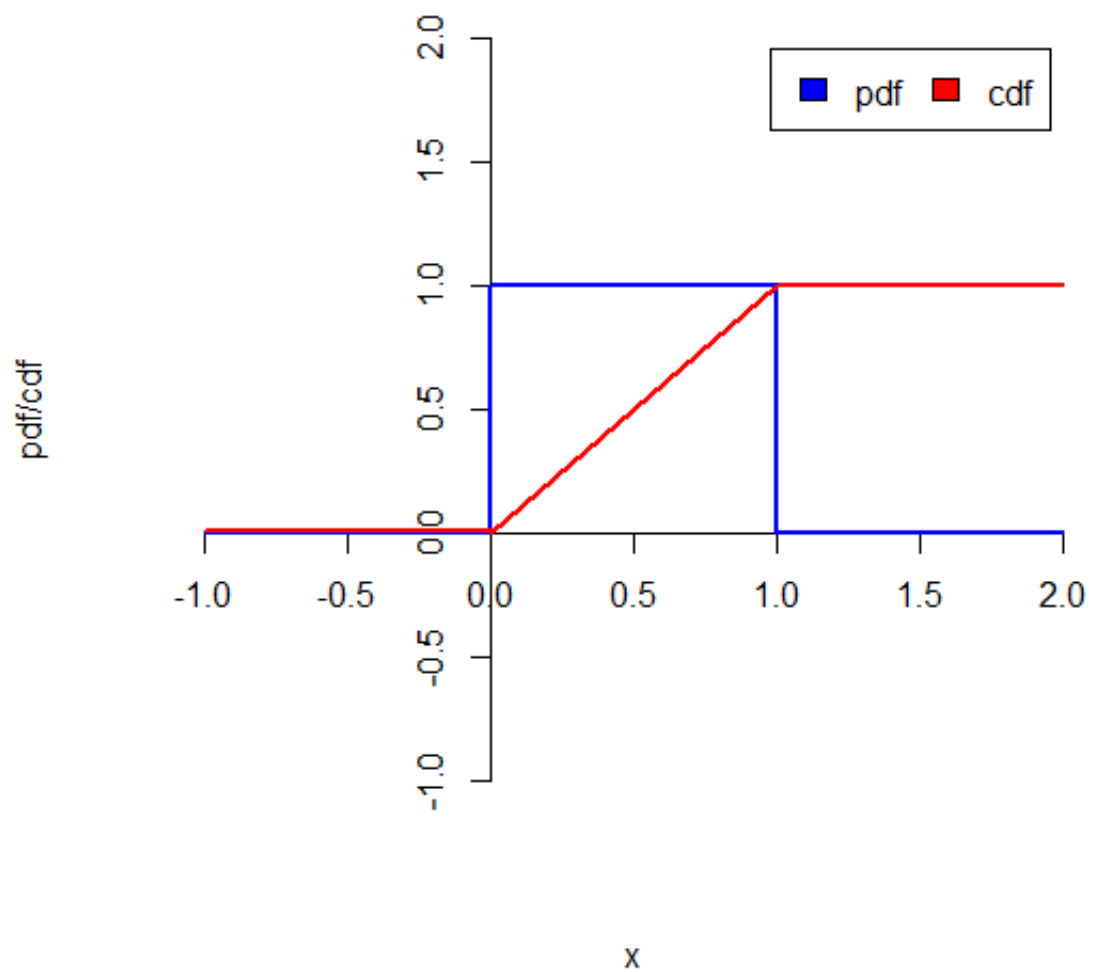
- Recall now the definition of a continuous uniform distribution  $U(a, b)$ .
- A r.v.  $X$  is defined as a **continuous uniform r.v.** with parameters  $a$  and  $b$  if its probability density function is:

$$f(x; a, b) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b]; \\ 0 & \text{otherwise.} \end{cases}$$

- Its cdf is:

$$F(x; a, b) = \begin{cases} 0 & \text{for } x < a; \\ \frac{x-a}{b-a} & \text{for } x \in [a, b]; \\ 1 & \text{for } x \geq b. \end{cases}$$

## RVG: The uniform distribution (cont'd)





## RVG: the universality of the uniform distribution and the inverse transform

- Let  $U$  be a Uniform  $(0, 1)$  r.v.
- Let  $X = F^{-1}(U)$ .  $X$  has cdf  $F(\cdot)$ , and to see this we want to show that:  

$$P(X \leq x) = F(x). \quad (1)$$
- By our definition above,  

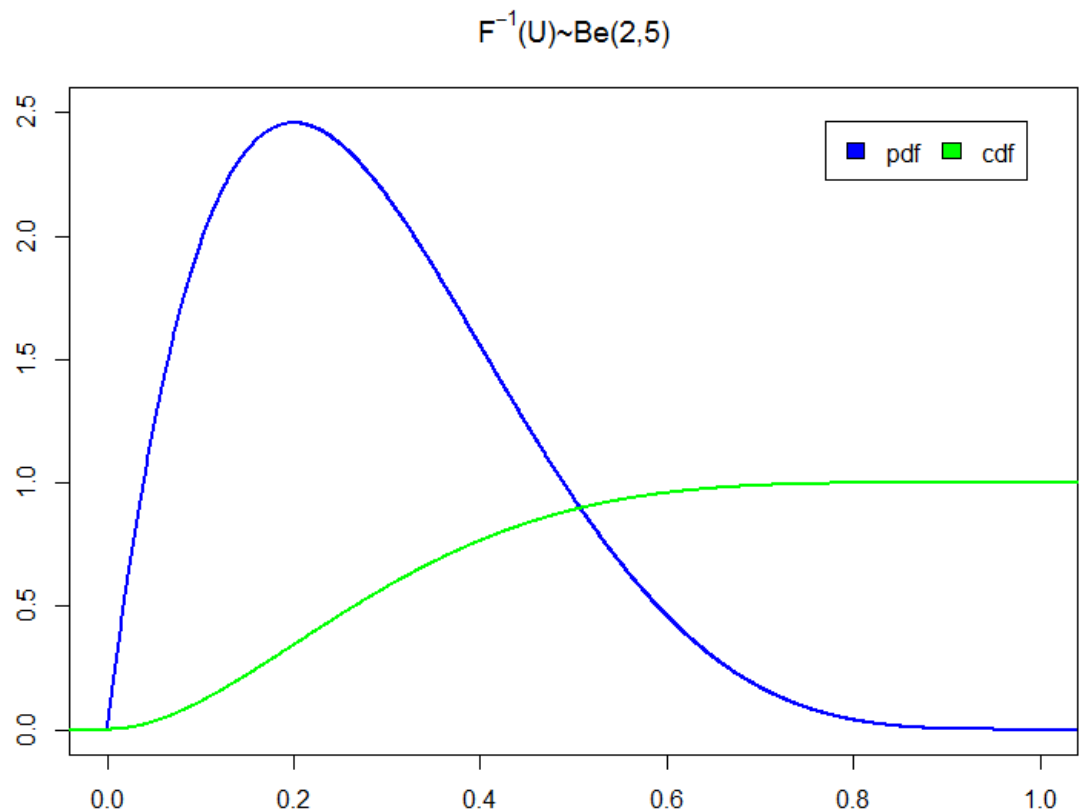
$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)). \quad (2)$$
- This means that if  $X$  has the cdf  $F(x)$ , then the random variable  $F(X)$  has the  $U(0, 1)$  distribution.
- Thus, formally, in order to generate a r.v.  $X \sim F$ , it suffices to generate  $U$  according to  $U(0, 1)$  and then make the transformation  $x = F^{-1}(u)$ .
- The key fact in this 'magic formula' is in applying  $F$  to both sides of inequality (2), which is allowed since cdfs are increasing functions.
- An equivalent version of (2) is the following:  

$$\begin{aligned} P(U \leq u) &= P[F(X) \leq F(x)] = P[F^{-1}(F(X)) \leq F^{-1}(F(x))] \\ &= P(X \leq x). \end{aligned}$$

## The universality of the uniform distribution: what is the message?

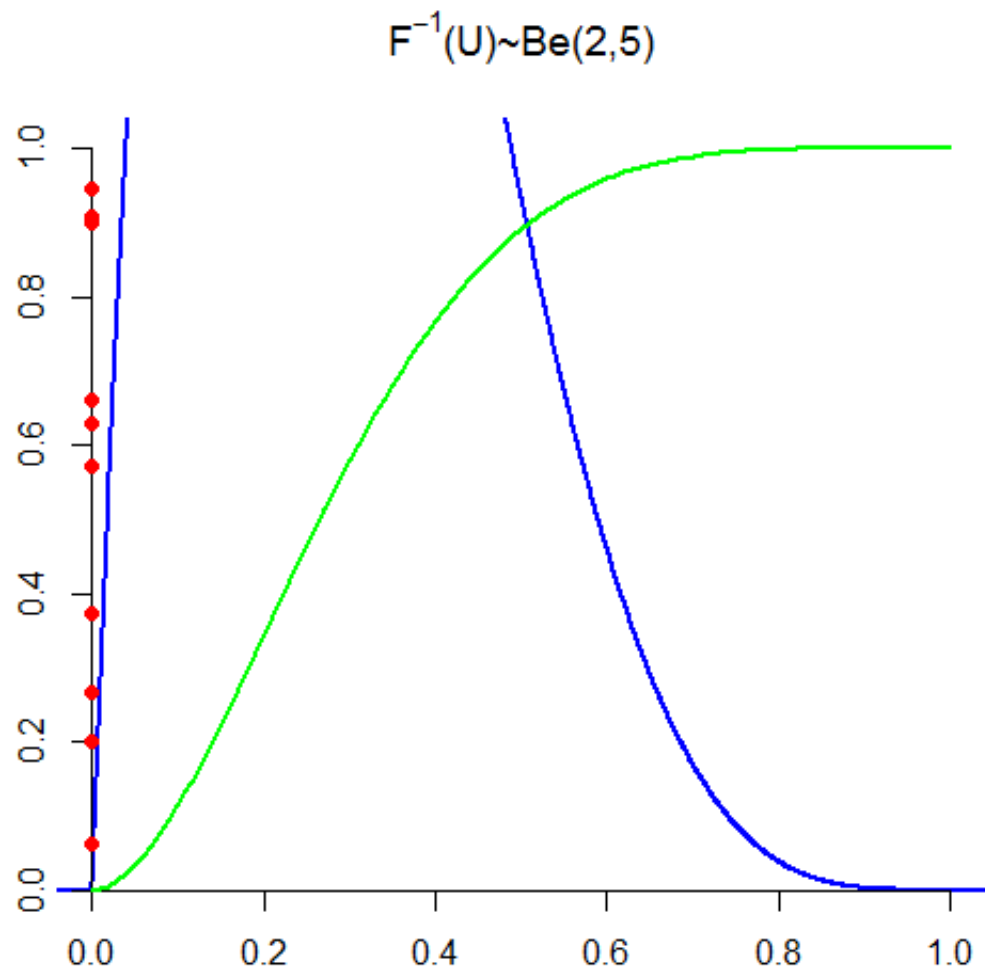
- Essentially, here we are talking about the possibility of going back and forth between the cdf of the uniform distribution and any other cdf.
- This is rather important since we are given the possibility of drawing random variables from many cdf using the uniform.
- To sum up before a useful visual explanation, this possibility of using the uniform distribution grounds on the fact that it is possible to transform any random variable into a  $U(0, 1)$  and, more importantly, to transform the uniform into any other random variable by means of a transformation known as the *probability integral transform*.
- Note that this gives us the possibility of *exactly* generating realizations of a r.v.

## RVG: Visual explanation of the universality of the uniform distribution

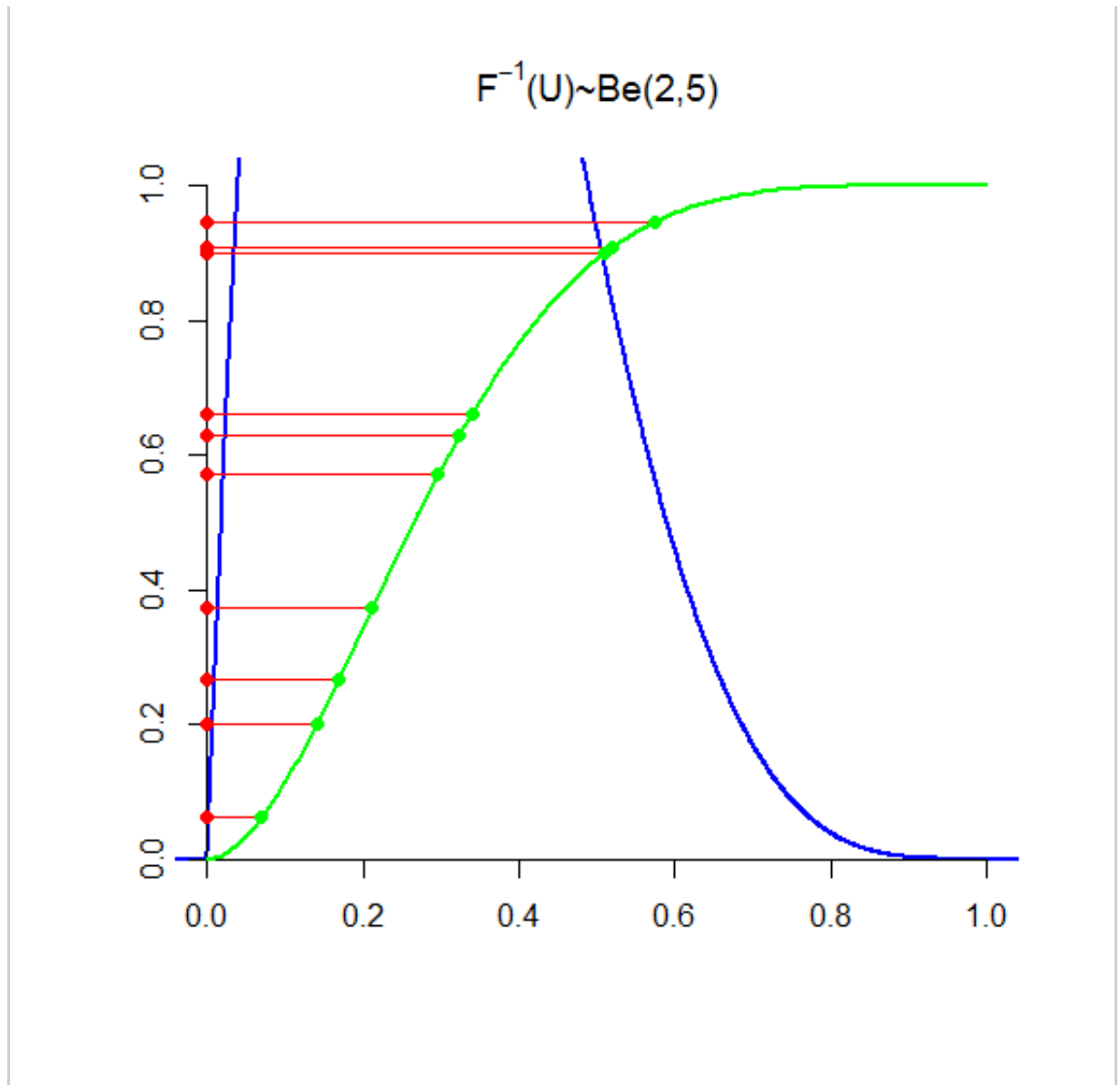


## RVG: Visual explanation of the universality of the uniform distribution (cont'd)

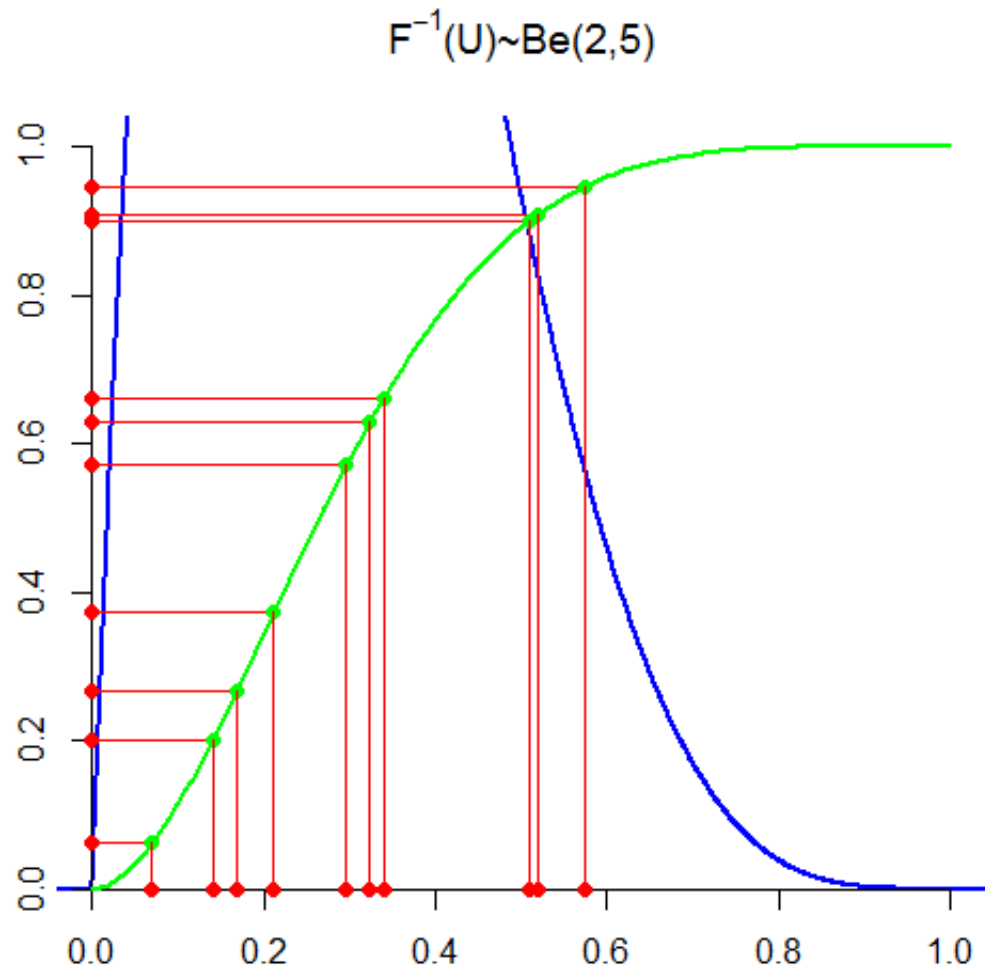
- Draw 10 values from a  $Unif(0, 1)$  and mark them on the y axis:



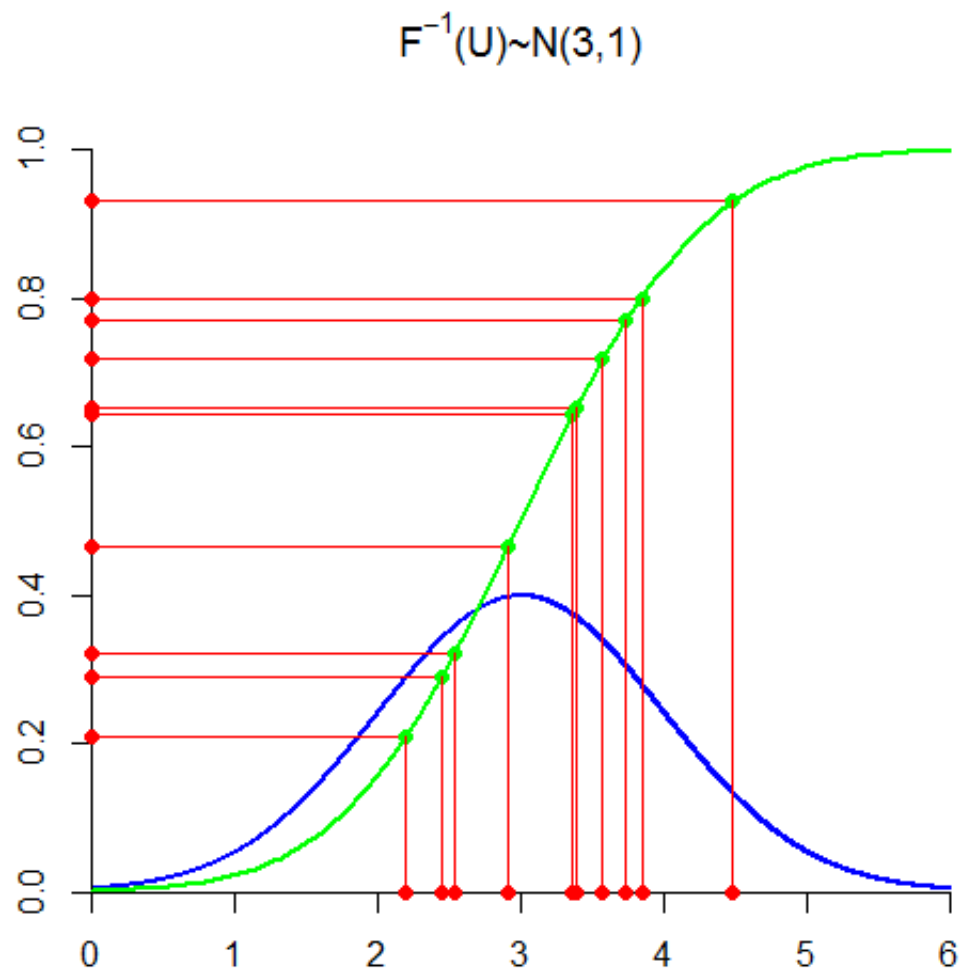
- Then, take the values of these 10 points on the cdf of the  $\text{Be}(2, 5)$  :



## RVG: Visual explanation of the universality of the uniform distribution (cont'd)



## RVG: Visual explanation of the universality of the uniform distribution (cont'd)



## Recap of the inverse transform method

- The inverse transform method.
- From Uniform  $(0, 1)$  to any distribution.

Let  $F(x)$  be any cumulative distribution function (cdf) of a r.v.  $X$ , i.e.

$F(x) = \int_{-\infty}^x f(t)dt$ , with  $f$  the density of  $X$ . Assume that  $F$  has an inverse and define  $X = F^{-1}(U)$ , where  $U$  has the continuous uniform distribution over the interval  $(0, 1)$ . Then  $X$  is distributed as  $F$ .

- **From any distribution to Uniform  $(0, 1)$ .** Let  $F(x)$  be the cumulative distribution function (cdf) of a r.v.  $X$ . Define  $U = F(X)$ . Then  $U$  is Uniform  $(0, 1)$ .
- **Sketch of proof.** The Uniform  $(0, 1)$  cdf is:

$$P(U \leq u) = P(F(X) \leq F(x)). \quad (1)$$

By taking the inverse of both sides of  $F(X) \leq F(x)$  we obtain

$F^{-1}(F(X)) \leq F^{-1}(F(x))$ . Therefore, the right-hand side of (1) becomes

$$P(F(X) \leq F(x)) = P(F^{-1}(F(X)) \leq F^{-1}(F(x))) = P(X \leq x) = F(x) \quad .$$

## Recap of the inverse transform algorithm

- In general we use the following steps in applications:

### Inverse transform algorithm

1. Generate  $U$  from  $Unif(0, 1)$ , i.e. set  $U \sim Unif(0, 1)$ .
2. Set  $U = F(X)$ .
3. Solve the inverse  $X = F^{-1}(U)$  for  $X$ , provided that the inverse of  $F$  exists.
4.  $X$  is the simulated value from  $F(x)$ .

# Inverse transform examples

## Example 1: the exponential r.v.

- The general exponential r.v. with parameter  $\lambda = \frac{1}{\theta}$  has the following pdf:

$$f(x) = \frac{1}{\theta} e^{-\frac{x}{\theta}} I_{(x>0)},$$

and cdf:

$$F(x) = \left(1 - e^{-\frac{x}{\theta}}\right) I_{(x>0)}.$$

- We set

$$U = F(X) = 1 - e^{-\frac{X}{\theta}} \sim \text{Unif}(0, 1),$$

and solve for  $X$ :

$$U = 1 - e^{-\frac{X}{\theta}}$$

$$U - 1 = -e^{-\frac{X}{\theta}}$$

$$1 - U = e^{-\frac{X}{\theta}}$$

$$\log(1 - U) = \log(e^{-\frac{X}{\theta}})$$

$$\log(1 - U) = -\frac{X}{\theta}$$

$$\theta \log(1 - U) = -X$$

$$X = -\theta \log(1 - U).$$

- That is, first generate  $U \sim \text{Unif}(0, 1)$  and use this value in:

$$(*) \quad X = -\theta \log(1 - U).$$

- Note: (\*) can be replaced by  $X = -\theta \log U$  since  $U \sim \text{Unif}(0, 1) \iff 1 - U \sim \text{Unif}(0, 1)$ .



## Inverse transform examples (cont'd)

### Example 2: the Weibull r.v.

- The Weibull r.v.  $X \sim \mathcal{W}(\alpha, \beta)$  with parameters  $(\alpha, \beta)$  has the following pdf:

$$f(x) = \alpha \beta x^{\beta-1} e^{-\alpha x^\beta} I_{(x>0)}.$$

- With  $\alpha = 1$ , it becomes:

$$f(x) = \beta x^{\beta-1} e^{-x^\beta} I_{(x>0)},$$

with cdf:

$$F(x) = (1 - e^{-x^\beta}) I_{(x>0)}.$$

- Then:

$$U = F(X) = 1 - e^{-X^\beta} \sim \text{Unif}(0, 1),$$

and hence:

$$X = [\log(1 - U)]^{\frac{1}{\beta}} = (-\log W)^{1/\beta}, \quad W = 1 - U \sim \text{Unif}(0, 1).$$

## Inverse transform examples (cont'd)

### Discrete distributions

- Let  $X$  be a discrete r.v. with  $P(X = x_i) = p_i$ ,  $i = 1, \dots, n$ , and  $x_i < x_{i+1}$ .
- In this case, the transform method has the following form:
  1. generate  $U \sim U(0, 1)$ ;
  2. find the smallest integer  $k$  such that  $U \leq F(x_k)$ ;
  3. return  $x_k$ .

## Inverse transform examples (cont'd)

### The Binomial r.v.

- Let  $X$  be a Binomial( $n, p$ ) r.v. with  $P(X = x; n, p) = \binom{n}{x} p^x (1 - p)^{n-x}$ ,  $x = 1, \dots, n$ , and  $x_i < x_{i+1}$ .
- In this case, the transform method can have the following form:
  - generate  $n$  iid r.v.s  $U_1, U_2, \dots, U_n \sim U(0, 1)$ ;
  - for each  $1 \leq i \leq n$  set  $Y_i = 0$  if  $U_i \leq (1 - p)$ ;  $Y_i = 1$  if  $U_i > (1 - p)$  (this yields  $n$  iid Bernoulli( $p$ ) values);
  - set  $X = \sum_{i=1}^n Y_i$ .

```
In [35]: 1 #Generating a single value for a binomial
          2 import numpy as np
          3 import scipy as sp
          4
          5 p = 0.5
          6 n = 50
          7 Y = np.zeros(n)
          8 for i in range(0,n):
          9     u = sp.random.uniform(low=0,high=1,size=1)
         10     if u < (1-p):
         11         Y[i] = 0
         12     else:
         13         Y[i] = 1
         14 X = np.sum(Y)
         15 X
```

Out [35]: 24.0

```
In [36]: 1 def binomial(n,p):
          2     p = 0.5
          3     n = 50
          4     Y = np.ones(n)
          5     for i in range(0,n):
          6         u = sp.random.uniform(low=0,high=1,size=1)
          7         if u < (1-p):
          8             Y[i] = 0
          9     X = np.sum(Y)
         10     return X
```

```
In [37]: 1 binomial(4,2)
```

Out [37]: 27.0

## Using Inversion Method

- Evaluation of  $F^{-1}$  could be costly.
- Or, its closed form does not exist  $\implies$  approximation of  $F^{-1}$  must be employed.
- For instance, `rnorm` in **R** for generating normal random numbers does not use standard inverse transform.

## Assignment 1

Adapt the code above to construct a general Python function to generate  $m$  values from a  $Bin(n, p)$ .

```
In [3]: 1 import numpy as np
        2 import scipy as sp
        3
        4 p = 0.1
        5 n = 50
        6 m = int(input("Enter the number of values to generate: "))
        7 for i in range(0,m):
        8     print(binomial(n,p))
```

Enter the number of values to generate: 5

```
4.0
7.0
6.0
10.0
7.0
```

## Assignment 2

Implement example 1 above in Python.

```
In [38]: 1 # Exponential r.v.
        2 teta = int(input("Enter teta: "))
        3 lb = 1/teta
        4 u = sp.random.uniform(low=0,high=1,size=1)
        5 print(-teta* np.log(1-u))
```

Enter teta: 3  
[0.1491853]

## The Box-Muller method: The Gaussian integral

- Suppose you want to calculate the following integral:

$$I = \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx,$$

which is quite tricky.

- We can solve this integral, called *Gaussian integral*, by taking its square value:

$$\begin{aligned} I^2 &= \left( \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx \right)^2 = \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx \int_{-\infty}^{+\infty} e^{-\frac{y^2}{2}} dy \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{-\frac{(x^2+y^2)}{2}} dx dy, (3) \end{aligned}$$

and using polar coordinates:

$$\begin{cases} x = r \cos \theta; \\ y = r \sin \theta. \end{cases}$$

## The Box-Muller method: The Gaussian integral (cont'd)

- The "substitution rule" for multiple integrals states that, when using other coordinates, the Jacobian determinant of the coordinate conversion formula should be used:

$$\begin{aligned} J = \det \frac{\delta(x, y)}{\delta(r, \theta)} &= \begin{vmatrix} \frac{\delta(x)}{\delta(r)} & \frac{\delta(x)}{\delta(\theta)} \\ \frac{\delta(y)}{\delta(r)} & \frac{\delta(y)}{\delta(\theta)} \end{vmatrix} = \begin{vmatrix} \cos \theta & -r \sin \theta \\ \sin \theta & r \cos \theta \end{vmatrix} \\ &= r \cos^2 \theta + r \sin^2 \theta = r. \end{aligned}$$

- So the area element in integral (3) is given in polar coordinates by  $dx dy = r dr d\theta$ .
- In integral (3),  $x$  and  $y$  vary over the whole plane. In polar variables this is given by:

$$0 \leq r < \infty;$$

$$0 \leq \theta \leq 2\pi.$$

## The Box-Muller method: The Gaussian integral (cont'd)

- Therefore, (3) can be transformed as follows:

$$I^2 = \int_0^{2\pi} \int_0^{+\infty} e^{-\frac{r^2}{2}} r dr d\theta$$

$$\int_0^{2\pi} \left[ -e^{-\frac{r^2}{2}} \right]_0^{\infty} d\theta = \int_0^{2\pi} d\theta = [\theta]_0^{2\pi} = 2\pi.$$

- The *Box-Muller* algorithm is a probabilistic interpretation of this type of integration.
- Let  $(X, Y)$  be a pair of independent standard normal random variables. Then, their joint probability density is:

$$f(x, y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x^2+y^2)}{2}}.$$

- We can express the r.v.  $(X, Y)$  using the polar coordinate random variables  $(R, \Theta)$  with  $0 \leq \Theta < 2\pi$ ,  $X = R \cos \Theta$  and  $Y = R \sin \Theta$ .

## The Box-Muller method: The Gaussian integral (cont'd)

- Let's start with stating that the random variable  $\Theta$  is  $\text{Unif}(0, 2\pi)$ .
- Therefore, we can use the following expression to sample from it:

$$\Theta = 2\pi U_1,$$

where  $U_1 \sim \text{Unif}(0, 1)$ .

- Now let's derive the cumulative distribution function for  $R$  by getting its marginal distribution via integration:

$$G(r) = P(R \leq r) = \int_0^r \int_0^{2\pi} \frac{1}{2\pi} e^{-\frac{r^2}{2}} r dr d\theta = \int_0^r e^{-\frac{r^2}{2}} r dr.$$

- Now, using the substitution  $\frac{r^2}{2} = s$  so that  $r dr = ds$ , we get:

$$G(r) = \int_0^s e^{-s} ds = 1 - e^{-\frac{r^2}{2}},$$

which is a cdf of an exponential r.v.

## The Box-Muller method: The Gaussian integral (cont'd)

- Therefore, we may sample from  $R$  by solving for  $R$  the following cumulative distribution function equation ( $(1 - U_2) \sim \text{Unif}(0, 1)$ ):

$$G(R) = 1 - e^{-\frac{R^2}{2}} = 1 - U_2.$$

- This gives:

$$R = \sqrt{-2 \log U_2}.$$

- So, the Box-Muller algorithm takes two  $\text{Unif}(0, 1)$ ,  $U_1$  and  $U_2$  and produces independent standard normal using the formulas:

$$\Theta = 2\pi U_1; \quad R = \sqrt{-2 \log U_2}; \quad X = R \cos \Theta \quad \text{and} \quad Y = R \sin \Theta.$$

## The Box-Muller method: Quick recap

- To turn a uniformly distributed random variable into a normally distributed variable, one might use the Box-Muller transform.
- Given two continuous uniform random variables  $U_1 \sim \text{Unif}(0, 1)$  and  $U_2 \sim \text{Unif}(0, 1)$ , then:

$$Z_0 = \sqrt{-2 \log U_2} \cos(2\pi U_1)$$

$$Z_1 = \sqrt{-2 \log U_2} \sin(2\pi U_1),$$

where  $Z_0$  and  $Z_1$  are independent random variables with normal distribution and standard deviation 1.

- Alternatively, given  $u \sim U(-1, 1)$  and  $v \sim U(-1, 1)$  with  $s = u^2 + v^2$  and  $0 \leq s < 1$ , then:

$$Z_0 = u \sqrt{\frac{-2 \log s}{s}}$$

$$Z_1 = v \sqrt{\frac{-2 \log s}{s}}$$

# The Box-Muller algorithm

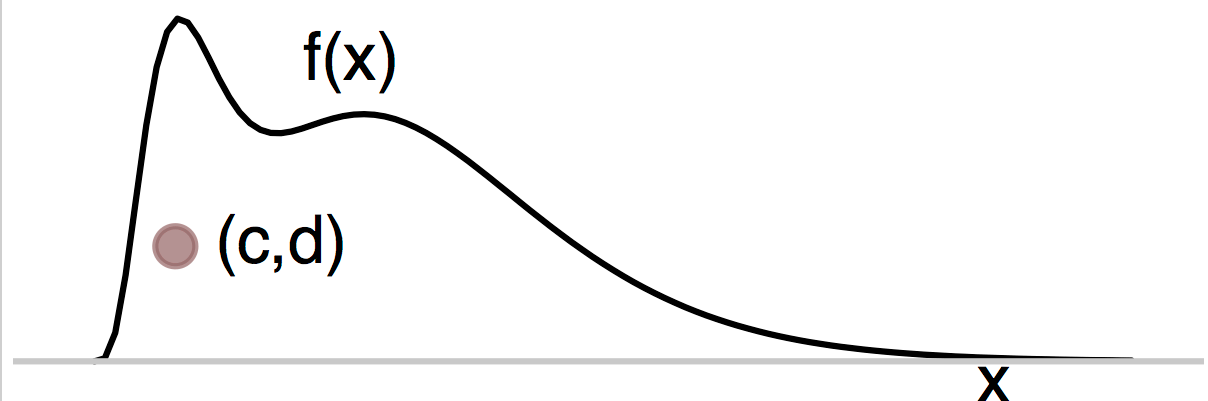
## The Box-Muller algorithm

- For practical use, one can use the following algorithm:
  - Generate  $U_1, U_2$  independently from  $Unif(0, 1)$ .
  - Set  $R = \sqrt{-2 \log U_2}$ .
  - Set  $\Theta = 2\pi U_1$ .
  - Then,  $Z_1 = R \cos \Theta$  and  $Z_2 = R \sin \Theta$  are the two simulated values from a  $N(0, 1)$ .

## Assignment 3

Implement the Box-Muller method in Python and compare it with the standard normal by plotting the histograms of the generated values.

## RVG indirect methods - The 'Accept-Reject' method



- Let's  $f(x)$  a *target density*, i.e. a density we want to draw from, for which we have problems in getting the values.
- Key point 1: we should know the mathematical functional form of a density  $f$  of interest up to a multiplicative constant.
- Key point 2: we use a simpler (to simulate) density  $g$  to generate the random variable for which the simulation is actually done.

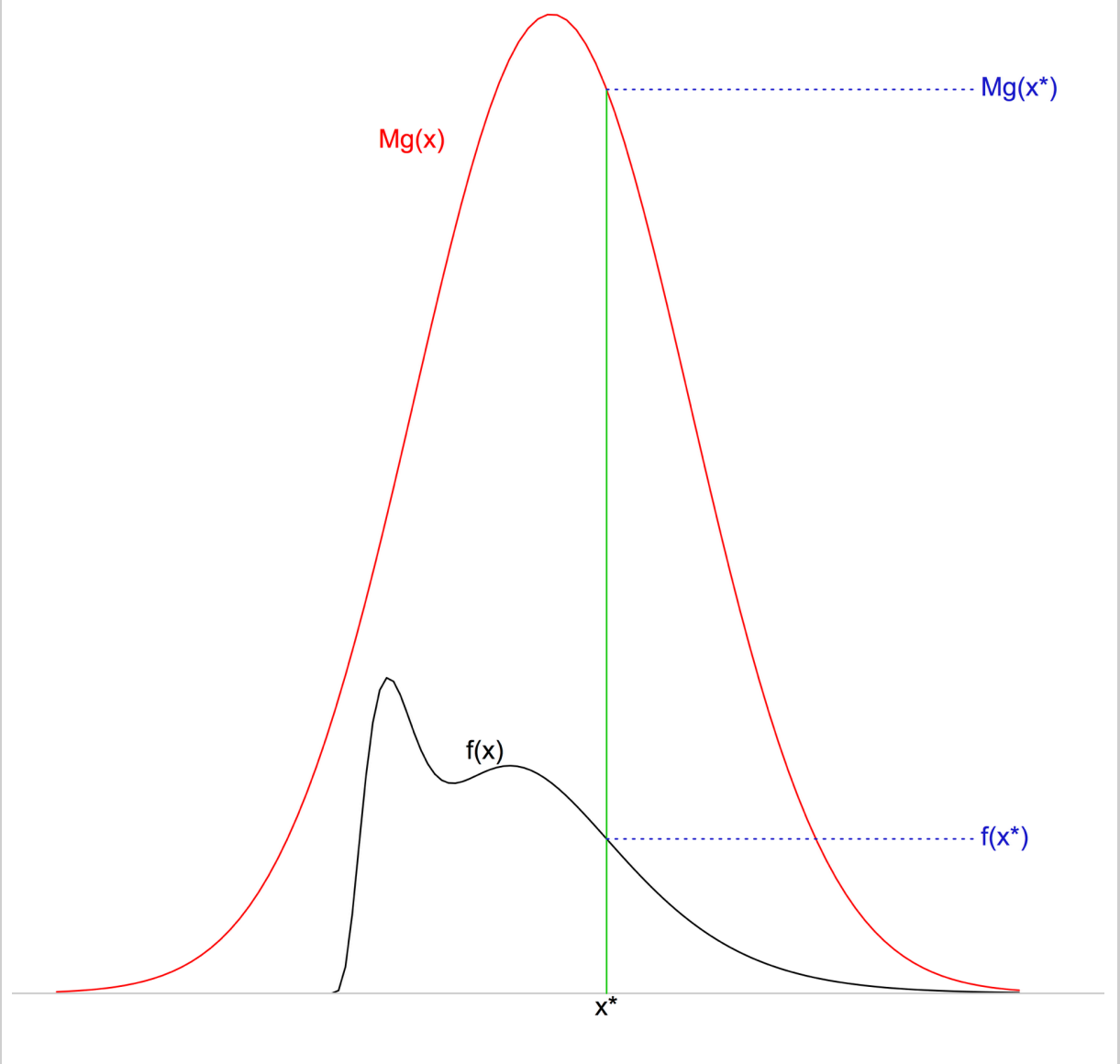
# The 'Accept-Reject' method: ingredients and requirements

- Ingredients:
  1. Two densities,  $f$  and  $g$ .
  2. The functional analytical form of the density  $f$  we wish to simulate from (that's why we call this density the target density) is known up to a multiplicative constant.
  3. The density  $g$  is simple to simulate from and is called \textit{the candidate density} to generate from the random variable for which the simulation is intended.
- Requirements:
  1.  $f$  and  $g$  are on the same support (or, at least, their supports are compatible; for example, if  $g(x) > 0$  then  $f(x) > 0$ ).
  2. There exists a constant  $M$  such that  $Mg(x) \geq f(x)$ .



## The 'Accept-Reject' method: algorithm (version 1)

- Have  $g(x)$  for which  $x$  is easily generated,  $f(x)$ , whose functional form is known, and a constant  $M$  such that  $Mg(x) \geq f(x)$ .
- $\forall x$ , and  $c \in (0, 1)$ ,
  1. generate  $x^*$  from  $g(x)$
  2. generate  $b \sim U(0, Mg(x^*))$ ,  $(x^*, b)$  is now random in the area under  $Mg(x)$
  3. If  $b \leq f(x^*)$ , return  $x = x^*$ . Otherwise, start all over from step 1.



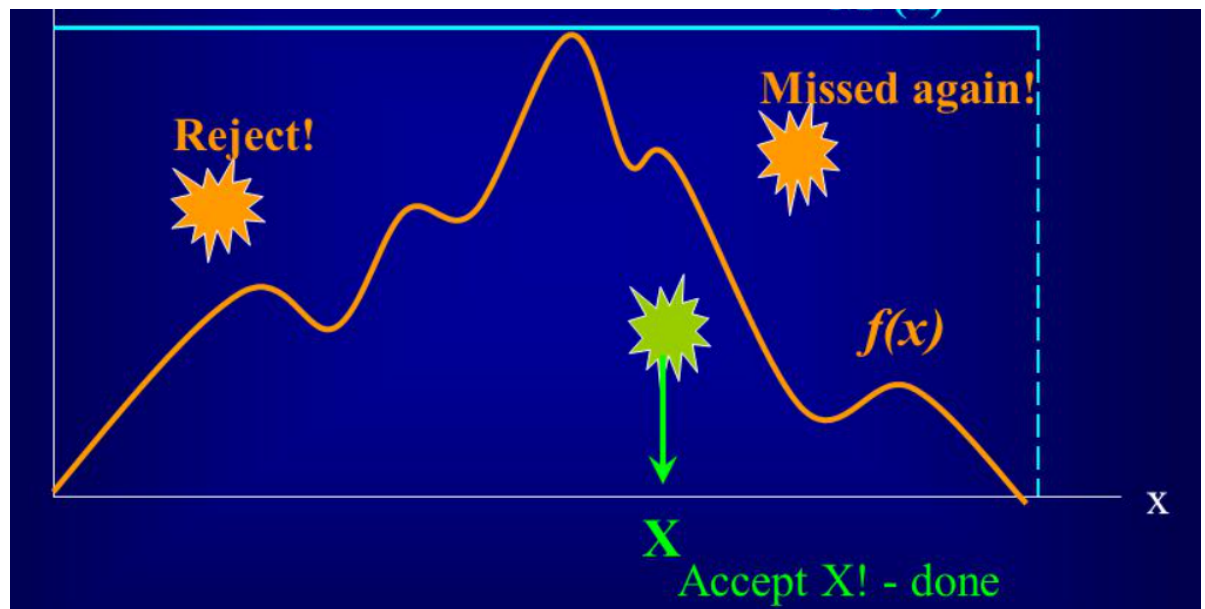
## The 'Accept-Reject' method: algorithm (version 2)

- **Accept-Reject algorithm:**
  1. Generate  $Y \sim g, U \sim U(0, 1)$ .
  2. Accept  $X = Y$  if  $U \leq \frac{f(Y)}{Mg(Y)}$ .
  3. Return to step 1 otherwise.

## The 'Accept-Reject' method: the 'dart similitude'

To better understand the idea underlying accept-reject method the following practical example could be useful.

- Suppose you have a board where you throw lots of darts. Suppose this board is a 1 meter by 1 meter square board.
- We have seen how to approximate the value  $\pi$  using the uniform (and in a certain sense we used the uniform like the darts hitting the unit circle area). Now we want to use the uniform again to approximate the value generation from a density  $f(x)$ .
- The 'accept-reject' method uses the same underlying idea as the darts: throw many darts at a board that is  $c$  high and with base  $[0, 1]$  on which we have drawn our desired density  $f(x)$ .
- Reject the darts that land above the curve, and accept those that land under it.
- There will be more darts near values of  $x$  for which  $f(x)$  is large than where  $f(x)$  is small, so with enough darts you will approximate the density function!



## Homework in Python

1. Check that the following function:

$$f(x) = \begin{cases} \frac{x^2}{18}, & -3 < x < 3; \\ 0, & \text{otherwise} \end{cases}$$

is a proper density on  $(-3, 3)$ .

- Plot its graph.
- Use accept-reject sampling to generate a set of random values for this density on  $(-3, 3)$ , using an appropriate density for  $g$ .

## The Box-Muller method: The Gaussian integral

- Suppose you want to calculate the following integral:

$$I = \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx,$$

which is quite tricky.

- We can solve this integral, called *Gaussian integral*, by taking its square value:

$$\begin{aligned} I^2 &= \left( \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx \right)^2 = \int_{-\infty}^{+\infty} e^{-\frac{x^2}{2}} dx \int_{-\infty}^{+\infty} e^{-\frac{y^2}{2}} dy \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} e^{-\frac{(x^2+y^2)}{2}} dx dy, (3) \end{aligned}$$

and using polar coordinates:

$$\begin{cases} x = r \cos \theta; \\ y = r \sin \theta. \end{cases}$$

## The Box-Muller method: The Gaussian integral (cont'd)

- The "substitution rule" for multiple integrals states that, when using other coordinates, the Jacobian determinant of the coordinate conversion formula should be used:

$$J = \det \frac{\delta(x, y)}{\delta(r, \theta)} = \begin{vmatrix} \frac{\delta(x)}{\delta(r)} & \frac{\delta(x)}{\delta(\theta)} \\ \frac{\delta(y)}{\delta(r)} & \frac{\delta(y)}{\delta(\theta)} \end{vmatrix} = \begin{vmatrix} \cos \theta & -r \sin \theta \\ \sin \theta & r \cos \theta \end{vmatrix} \\ = r \cos^2 \theta + r \sin^2 \theta = r.$$

- So the area element in integral (3) is given in polar coordinates by  $dx dy = r dr d\theta$ .
- In integral (3),  $x$  and  $y$  vary over the whole plane. In polar variables this is given by:

$$0 \leq r < \infty;$$

$$0 \leq \theta \leq 2\pi.$$

## The Box-Muller method: The Gaussian integral (cont'd)

- Therefore, (3) can be transformed as follows:

$$I^2 = \int_0^{2\pi} \int_0^{+\infty} e^{-\frac{r^2}{2}} r dr d\theta \\ \int_0^{2\pi} \left[ -e^{-\frac{r^2}{2}} \right]_0^{\infty} d\theta = \int_0^{2\pi} d\theta = [\theta]_0^{2\pi} = 2\pi.$$

- The *Box-Muller* algorithm is a probabilistic interpretation of this type of integration.
- Let  $(X, Y)$  be a pair of independent standard normal random variables. Then, their joint probability density is:

$$f(x, y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x^2+y^2)}{2}}.$$

- We can express the r.v.  $(X, Y)$  using the polar coordinate random variables  $(R, \Theta)$  with  $0 \leq \Theta < 2\pi$ ,  $X = R \cos \Theta$  and  $Y = R \sin \Theta$ .

## The Box-Muller method: The Gaussian integral (cont'd)

- Let's start with stating that the random variable  $\Theta$  is  $\text{Unif}(0, 2\pi)$ .
- Therefore, we can use the following expression to sample from it:

$$\Theta = 2\pi U_1,$$

where  $U_1 \sim \text{Unif}(0, 1)$ .

- Now let's derive the cumulative distribution function for  $R$  by getting its marginal distribution via integration:

$$G(r) = P(R \leq r) = \int_0^r \int_0^{2\pi} \frac{1}{2\pi} e^{-\frac{r^2}{2}} r dr d\theta = \int_0^r e^{-\frac{r^2}{2}} r dr.$$

- Now, using the substitution  $\frac{r^2}{2} = s$  so that  $r dr = ds$ , we get:

$$G(r) = \int_0^s e^{-s} ds = 1 - e^{-\frac{r^2}{2}},$$

which is a cdf of an exponential r.v.

## The Box-Muller method: The Gaussian integral (cont'd)

- Therefore, we may sample from  $R$  by solving for  $R$  the following cumulative distribution function equation ( $(1 - U_2) \sim \text{Unif}(0, 1)$ ):

$$G(R) = 1 - e^{-\frac{R^2}{2}} = 1 - U_2.$$

- This gives:

$$R = \sqrt{-2 \log U_2}.$$

- So, the Box-Muller algorithm takes two  $\text{Unif}(0, 1)$ ,  $U_1$  and  $U_2$  and produces independent standard normal using the formulas:

$$\Theta = 2\pi U_1; \quad R = \sqrt{-2 \log U_2}; \quad X = R \cos \Theta \quad \text{and} \quad Y = R \sin \Theta.$$

## The Box-Muller method: Quick recap

- To turn a uniformly distributed random variable into a normally distributed variable, one might use the Box-Muller transform.
- Given two continuous uniform random variables  $U_1 \sim \text{Unif}(0, 1)$  and  $U_2 \sim \text{Unif}(0, 1)$ , then:

$$Z_0 = \sqrt{-2 \log U_2} \cos(2\pi U_1)$$

$$Z_1 = \sqrt{-2 \log U_2} \sin(2\pi U_1),$$

where  $Z_0$  and  $Z_1$  are independent random variables with normal distribution and standard deviation 1.

- Alternatively, given  $u \sim U(-1, 1)$  and  $v \sim U(-1, 1)$  with  $s = u^2 + v^2$  and  $0 \leq s < 1$ , then:

$$Z_0 = u \sqrt{\frac{-2 \log s}{s}}$$

$$Z_1 = v \sqrt{\frac{-2 \log s}{s}}$$

## The Box-Muller algorithm

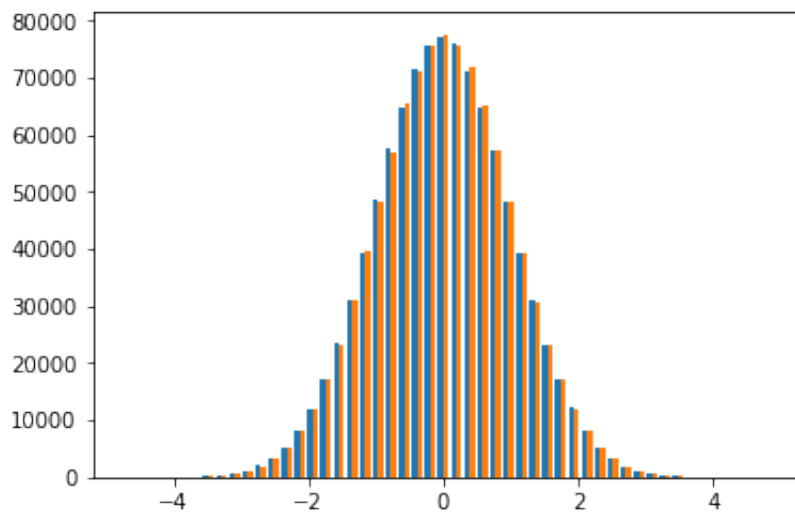
### The Box-Muller algorithm

- For practical use, one can use the following algorithm:
  - Generate  $U_1, U_2$  independently from  $\text{Unif}(0, 1)$ .
  - Set  $R = \sqrt{-2 \log U_2}$ .
  - Set  $\Theta = 2\pi U_1$ .
  - Then,  $Z_1 = R \cos \Theta$  and  $Z_2 = R \sin \Theta$  are the two simulated values from a  $N(0, 1)$ .

**Implement the Box-Muller method in Python and compare it with the standard normal by plotting the histograms of the generated values.**

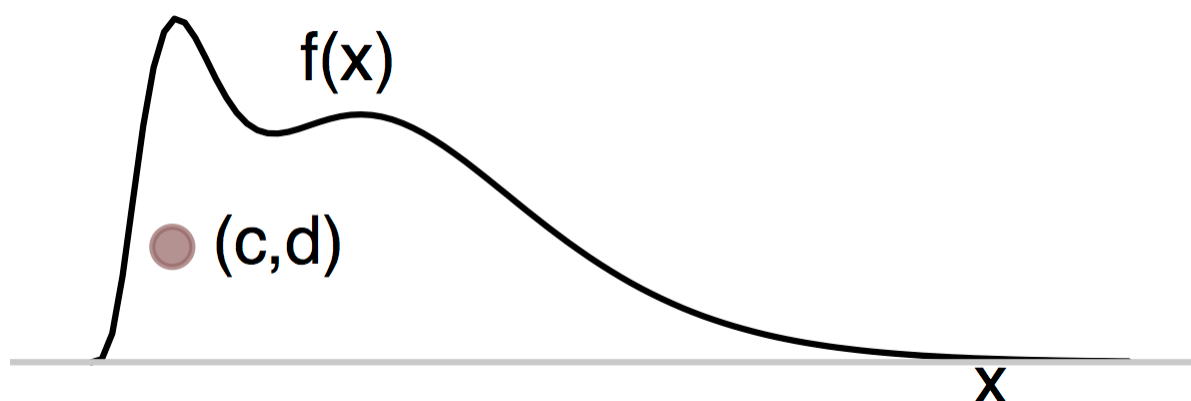
```
In [1]: 1 import math
2 # n = number of values to generate
3 def box_muller(n):
4     u1 = sp.random.uniform(low=0,high=1,size=n//2)
5     u2 = sp.random.uniform(low=0,high=1,size=n//2)
6     r = np.sqrt(-2*np.log(u2))
7     teta = 2*math.pi*u1
8     z1 = r*np.cos(teta)
9     z2 = r*np.sin(teta)
10    return np.concatenate((z1,z2))
11
```

```
In [4]: 1 import scipy as sp
2 import numpy as np
3 import matplotlib.pyplot as plt
4 %matplotlib inline
5
6 n = 1000000
7 n, bins, patches = plt.hist([ np.random.randn(n), box_muller(n)
```





## RVG indirect methods - The 'Accept-Reject' method



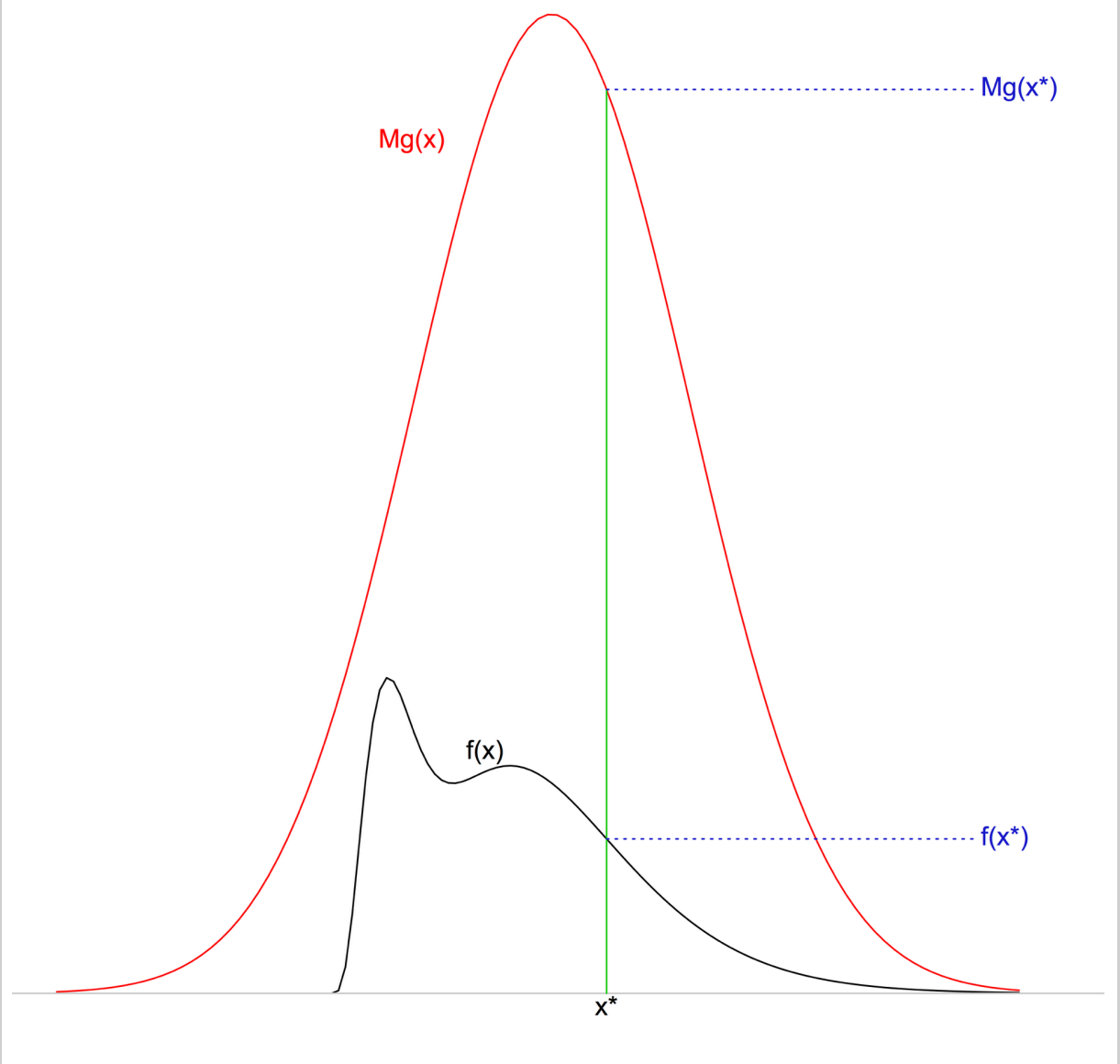
- Let's  $f(x)$  a *target density*, i.e. a density we want to draw from, for which we have problems in getting the values.
- Key point 1: we should know the mathematical functional form of a density  $f$  of interest up to a multiplicative constant.
- Key point 2: we use a simpler (to simulate) density  $g$  to generate the random variable for which the simulation is actually done.

## The 'Accept-Reject' method: ingredients and requirements

- Ingredients:
  1. Two densities,  $f$  and  $g$ .
  2. The functional analytical form of the density  $f$  we wish to simulate from (that's why we call this density the target density) is known up to a multiplicative constant.
  3. The density  $g$  is simple to simulate from and is called \textit{the candidate density} to generate from the random variable for which the simulation is intended.
- Requirements:
  1.  $f$  and  $g$  are on the same support (or, at least, their supports are compatible; for example, if  $g(x) > 0$  then  $f(x) > 0$ ).
  2. There exists a constant  $M$  such that  $Mg(x) \geq f(x)$ .

## The 'Accept-Reject' method: algorithm (version 1)

- Have  $g(x)$  for which  $x$  is easily generated,  $f(x)$ , whose functional form is known, and a constant  $M$  such that  $Mg(x) \geq f(x)$ .
- $\forall x$ , and  $c \in (0, 1)$ ,
  1. generate  $x^*$  from  $g(x)$
  2. generate  $b \sim U(0, Mg(x^*))$ ,  $(x^*, b)$  is now random in the area under  $Mg(x)$
  3. If  $b \leq f(x^*)$ , return  $x = x^*$ . Otherwise, start all over from step 1.



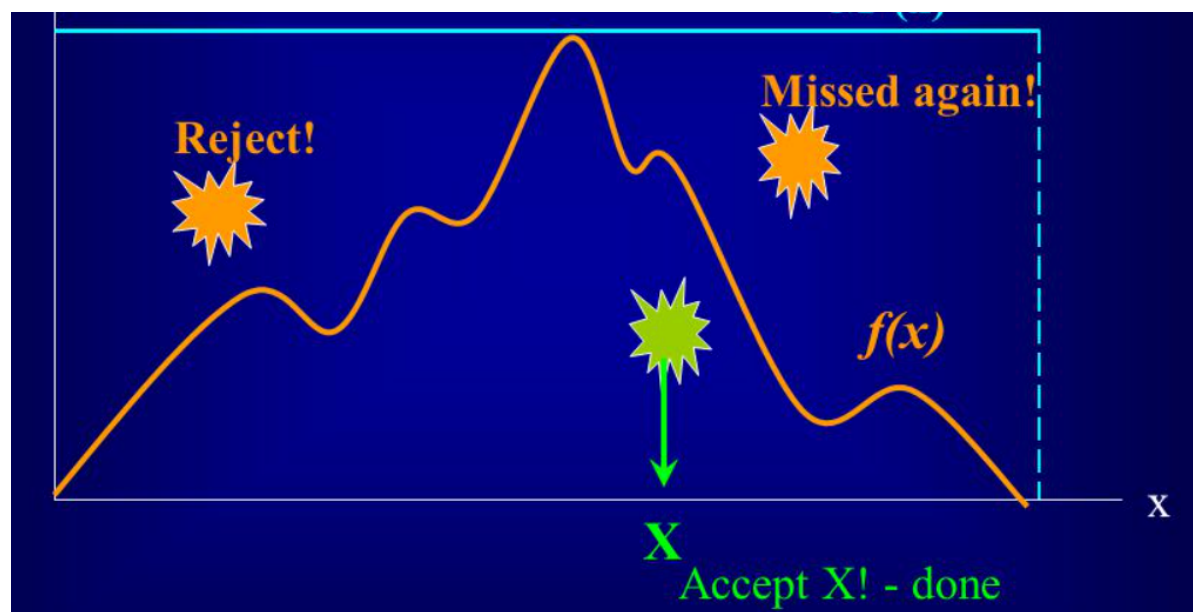
## The 'Accept-Reject' method: algorithm (version 2)

- **Accept-Reject algorithm:**
  1. Generate  $Y \sim g, U \sim U(0, 1)$ .
  2. Accept  $X = Y$  if  $U \leq \frac{f(Y)}{Mg(Y)}$ .
  3. Return to step 1 otherwise.

## The 'Accept-Reject' method: the 'dart similitude'

To better understand the idea underlying accept-reject method the following practical example could be useful.

- Suppose you have a board where you throw lots of darts. Suppose this board is a 1 meter by 1 meter square board.
- We have seen how to approximate the value  $\pi$  using the uniform (and in a certain sense we used the uniform like the darts hitting the unit circle area). Now we want to use the uniform again to approximate the value generation from a density  $f(x)$ .
- The 'accept-reject' method uses the same underlying idea as the darts: throw many darts at a board that is  $c$  high and with base  $[0, 1]$  on which we have drawn our desired density  $f(x)$ .
- Reject the darts that land above the curve, and accept those that land under it.
- There will be more darts near values of  $x$  for which  $f(x)$  is large than where  $f(x)$  is small, so with enough darts you will approximate the density function!



1. Check that the following function:

$$f(x) = \begin{cases} \frac{x^2}{18}, & -3 < x < 3; \\ 0, & \text{otherwise} \end{cases}$$

is a proper density on  $(-3, 3)$ .

- Plot its graph.
- Use accept-reject sampling to generate a set of random values for this density on  $(-3, 3)$ , using an appropriate density for  $g$ .

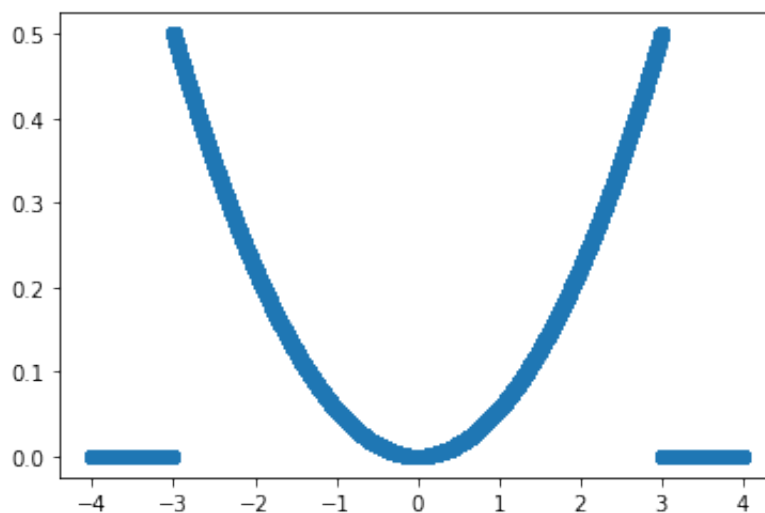
```
In [5]: 1 import scipy.integrate as integrate
2 import scipy.special as special
3 import matplotlib.pyplot as plt
4 result = integrate.quad(lambda x: x**2/18, -3, 3)
5 result
6 #since result is 1 the function is density
```

Out[5]: (1.0, 1.1102230246251565e-14)

```
In [6]: 1 def f(x):
2     if -3. < x < 3.:
3         return x**2/18.
4     else:
5         return 0.
```

```
In [7]: 1 n_examples = 1000000
2 x = np.linspace(-4.,4.,n_examples)
3 vecf = np.vectorize(f)
4 y = vecf(x)
5 plt.scatter(x,y)
```

Out[7]: <matplotlib.collections.PathCollection at 0x7fd2aa94f490>

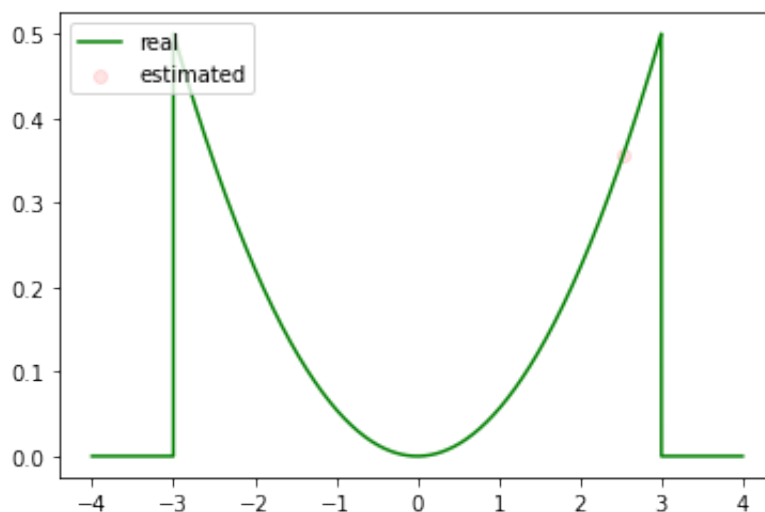


```

In [9]: 1 # exploiting uniform distribution, better choice
        2 M = 5
        3 #generate acceptance threshold
        4 u = np.random.uniform(0,1,n_examples)
        5 # generate points from the uniform
        6 xs = np.random.uniform(-4,4,n_examples)
        7 # compute the accepted points
        8 ratio = vecf(xs) / M*(1/8)
        9 indices = (u <= ratio)
       10 plt.scatter(xs[indices],vecf(xs[indices]), c = 'r',label='estimated')
       11 plt.plot(x, y, c='g', marker='-', linestyle='-', label='real')
       12 plt.legend(loc='upper left')
       13 plt.show
       14 print('acceptance rate = ' + str(np.sum(indices)/len(xs)))

```

acceptance rate = 0.01

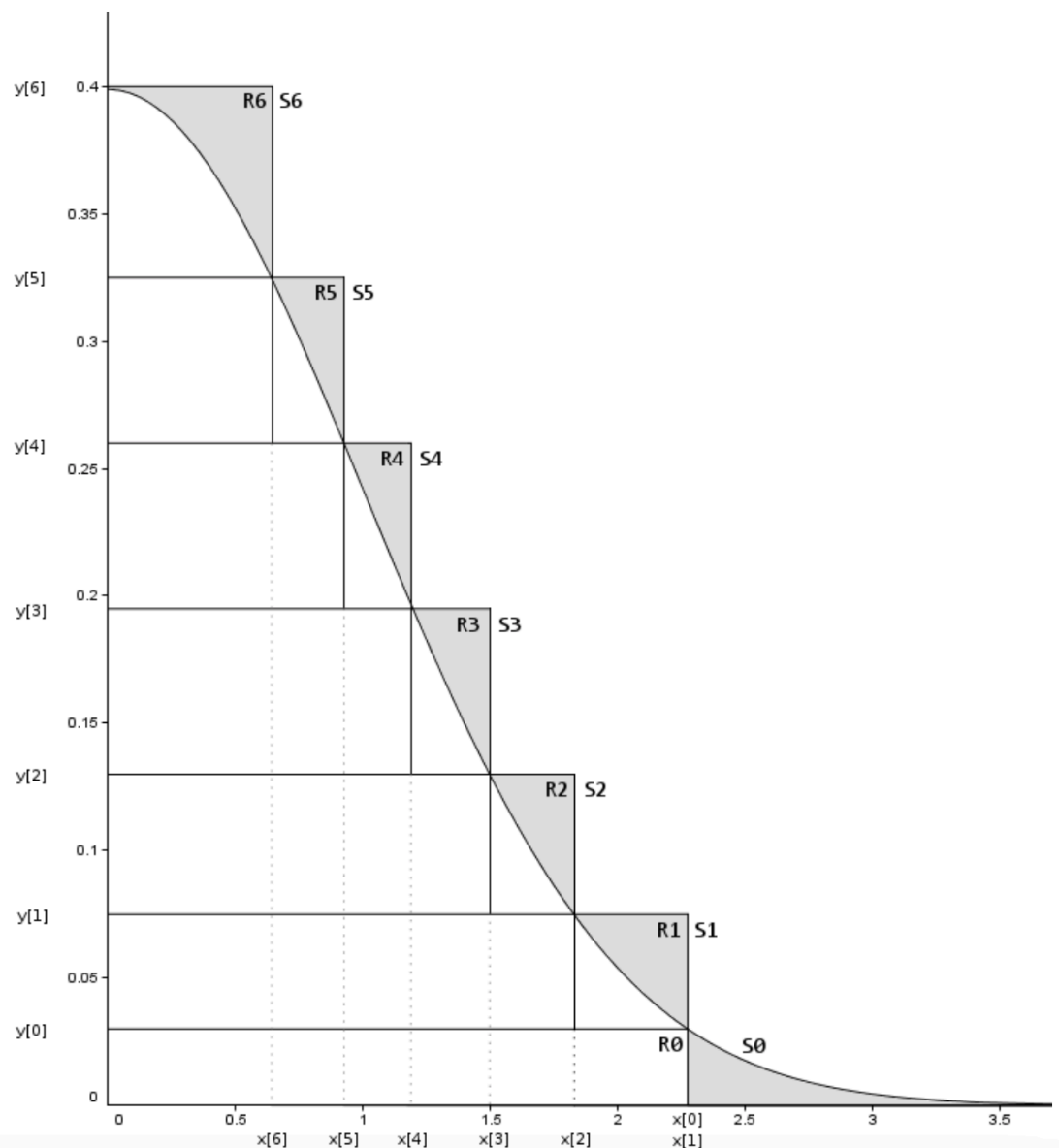


## RVG for normal implemented in Numpy

- Until a few months ago the Box-Muller method was implemented in Numpy with the method `numpy.random.Generator.standard_normal`
- Now the Inverse method and a faster method called "Ziggurat" are used.

## The Ziggurat algorithm (Marsaglia and Tsang, 2000)

- The Ziggurat algorithm covers the pdf with a series of horizontal rectangles rather than a single square like in the accept-reject algorithm, and in an arrangement that attempts to cover the pdf as efficiently as possible, i.e with minimum area outside of the pdf curve. The following diagram demonstrates the approach.
- Note that we operate on one side of the pdf ( $x \geq 0$ ), generating both positive and negative sample values requires that as a final step we randomly flip the sign of the generated non-negative values.



```
In [25]: 1 rng = np.random.default_rng()
          2 rng.standard_normal(10)
```

```
Out[25]: array([-0.23665748, -0.48325906, -0.13102053,  1.45738656,  0.6931
774 ,
          1.05605885, -0.68306166, -0.02044268, -0.02995312,  0.3514
9289])
```

```
In [1]: 1 # To run slideshow type jupyter nbconvert /Users/giancarlomanzi.
          2 # from terminal
          3 #/Users/giancarlomanzi/Documents/Box Sync BackUp PC Lavoro 2406.
          4
          5 #This is to let you have larger fonts...
          6 from IPython.core.display import HTML
          7 HTML("""
          8 <style>
          9
         10 div.cell { /* Tunes the space between cells */
         11 margin-top:1em;
         12 margin-bottom:1em;
         13 }
         14
         15 div.text_cell_render h1 { /* Main titles bigger, centered */
         16 font-size: 2.2em;
         17 line-height:1.4em;
         18 text-align:center;
         19 }
         20
         21 div.text_cell_render h2 { /* Parts names nearer from text */
         22 margin-bottom: -0.4em;
         23 }
         24
         25
         26 div.text_cell_render { /* Customize text cells */
         27 font-family: 'Times New Roman';
         28 font-size:1.5em;
         29 line-height:1.4em;
         30 padding-left:3em;
         31 padding-right:3em;
         32 }
         33 </style>
         34 """)
```

```
Out[1]:
```

```
In [ ]: 1
```



