Random Numbers and Variable Generation

Alberto Garfagnini

Università di Padova

AA 2020/2021

Why Random Numbers ?

- random numbers are commonly used for

▷ simulation of physical systems involving stochastic variables. Several simulations of physical or real systems (e.g. passage of ionizing particles through matter, hospital acceptance system simulation) need random variables

▷ sampling : to study and/or use different probability distributions

▷ numerical analysis : different techniques involving random numbers are employed to solve problems with numerical techniques (from simple to complex ones)

▷ computer programming : random numbers are often used in current computer programs

▷ decision making

▷ games theory

Random Number Generation

Historical excursus

- early times : manual techniques used. Es. coin flipping, dice rolling, card shuffling.

↠ in 1995, RAND Corporation published a list of 1 Million random numbers obtained with mechanical methods

- later on : physical devices: noise diodes, Geiger counters

- computer era : simple algorithms on a computing element.

↠ They are not based on a specific physical device. Run fast, require little storage, and they can reproduce a given sequence of random numbers

The Middle-Square Method

- the first to suggest an algorithm for random number generation was John von Neumann back in 1946

Algorithm

1. take a number with a large number of digits, for instance 10, and square it
2. extract the 10 central digits
3. repeat the sequence from 1

5772156649

⅄

33317 7923805949 09184

⅄

62786 7007174077 89056

Q&A

Q: the generated sequence is not randomly generated, since each number is determined by its predecessor. Why it is called random ?

A: Yes, but it seems random, Therefore it is called *pseudo-random*

The Linear Congruential Generator (LCG)

- it was the most popular. Introduced by D. H. Lehmer in 1949
- it allows to generate a random sequence $\{X_n\}$ using

$$X_{n+1} = (aX_n + C) \mod m$$

- where

$$\begin{array}{ccl} 0 < a < m & : & \text{multiplier} \\ 0 < C < m & : & \text{increment} \\ 0 < X_\circ < m & : & \text{seed. i.e. starting point} \\ m > 0 & : & \text{modulus} \end{array}$$

Example

- let's consider the following generator

$$X_{n+1} = (7 \cdot X_n + 7) \text{ modulus } 10$$

- starting with $X_\circ = 7$, we get

$$\{X_n\} = \{7, 6, 9, 0, 7, 6, 9, 0, \ldots\}$$

- the sequence repeats, with a 4 elements cycle

LCG parameters and examples

- to get an useful sequence we need a large cycle
- several parameters have been studied (many papers in literature)
- from a Theorem (see D. Knuth, *The Art of Computer Programming*, vol 2, semi-numerical algorithms, Addison Wesley 1981, ISBN 0-201-03822-6)
- the LCG period is at most m if and only if

i) c is relatively prime to m

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

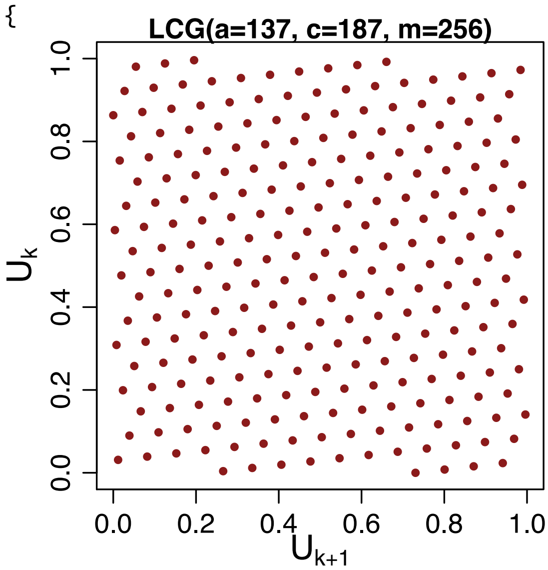
iii) $a - 1$ is a multiple of 4, if m is a multiple of 4

Source	m	a	c
Numerical Recipes	2^{32}	1664525	1013904223
Borland C/C++	2^{32}	22695477	1
glibc	2^{32}	1103515245	12345
ANSI C	2^{32}	1103515245	12345
Borland Delphi, Virtual Pascal	2^{32}	134775813	1
Microsoft Visual/Quick C/C++	2^{32}	214013	2531011
Apple CarbonLib	$2^{31} - 1$	16807	0
MMIX (D. Knuth)	2^{64}	6364136223846793005	1442695040888963407

LCG in R

- let's consider the LCG: $X_{n+1} = (137 \cdot X_n + 187) \bmod 2^8 = 256$
- when we plot the points (X_{j+1}, X_j)
- we find out that the points do not fill up the whole space, but they lay on selected lines
- the distance between the lines is $\sqrt{m}/m = 16/256 = 1/16 = 0.625$

```
lcg.user <- function(nsample=100, seed=1) {
  rand <- vector(length = nsample)
  m <- 256; a <- 137; c <- 187
  d <- seed
  for (i in 1:nsample) {
    d <- (a * d + c) %% m
    rand[i] <- d / m
  }
  return(rand)
}
u <- lcg.user(257)
points( u[-1], u[-257],
        col='firebrick4', pch=20)
```



- this problem was discovered on the `RANDU` generator, available on the IBM in 1950-1960

G. Marsaglia, *Random Numbers Fall Mainly in the Planes*, Proc. Natl. Acad. Sci. USA. 6 (1968)

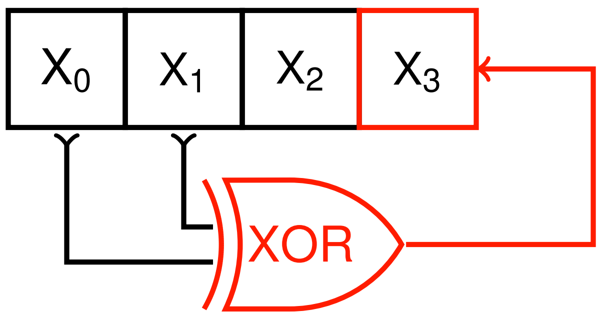
Shift Register generators

- each bit of the number is seen as an element of a binary vector
- logical linear functions are applied on each bit
- one set of generators is based on the `XOR` logical function

Example

- let's assume a 4-bit number: $\{X_0 X_1 X_2 X_3\}$
- `XOR` is applied on bits X_0, X_1 and the result is inserted on the most significant bit (with shift towards less significant bits)

0	1101	8	1000
1	1010	9	0001
2	0101	10	0010
3	1011	11	0100
4	0111	12	1001
5	1111	13	0011
6	1110	14	0110
7	1100	15	1101



- The number `0000` is excluded from the sequence

The 64bit XOR Shift generators

Algorithm

i) init the seed with a number $\neq 0$ on 64-bits

ii) apply the following operations in sequence:

$$
\begin{aligned}
x &= x \oplus (x >> a_1) \\
x &= x \oplus (x << a_2) \\
x &= x \oplus (x >> a_3)
\end{aligned}
$$

a_1	a_2	a_3
21	35	4

iii) release x

- the generator period is $2^{64} - 1$

Lagged Fibonacci generators

- they are to be considered an extension to the LCGs
- they use a recurring formula

$$X_{n+1} = (X_{n-r} \,\square\, X_{n-s}) \mod m$$

- where $\square$ indicates a generic binary operator, $\square = +, -, *, \oplus, \otimes, ldots$
- They are indicated as $F(r,\ s,\ \square)$ generators

Examples

- $F(0, 1, +)$: generates the standard Fibonacci sequence:

$$X_{n+1} = (X_n + X_{n-1}) \mod m$$

- the Knuth-TAOCP-2002 generator:

$$F(37, 100, +) : X_{n+1} = (X_{n-37} + X_{n-100}) \mod m = 2^{30}$$

- the period is around 2^{219}

D. Knuth, *The Art of Computer Programming*, Vol 2, semi-numerical algorithms, Addison Wesley 2002

Random number generation in R

- random numbers, in a specific interval, can be generated using the `runif(n, lower, upper)` function
- the underlying random number generator can be set/retrieved using the `RNGkind(kind = NULL, normal.kind = NULL, sample.kind = NULL)` function
- `set.seed` uses a single integer argument to set as many seeds as are required

```
RNGkind()
# [1] "Mersenne-Twister" "Inversion"

RNGkind("Wich")
RNGkind()
# [1] "Wichmann-Hill" "Inversion"

.Random.seed
# [1]    400 24434 13963 16439

RNGkind("Super") # matches  "Super-Duper"
RNGkind()
# [1] "Super-Duper" "Inversion"

.Random.seed # new, corresponding to  Super-Duper
# [1]          402 -1462836548 -1846862707
```


Random number generators in R

- `Wichmann-Hill` : the Wichmann–Hill generator has a cycle length of 6.9536e12

  B. A. Wichmann and I. D. Hill, *n Efficient and Portable Pseudo-Random NumberGenerator*, Applied Stat. 33 (1984), 123
- `Marsaglia-Multicarry` : a multiply-with-carry RNG. It has a period of more than 2^{60} and passed all Marsaglia Diehard battery tests
- `Super-Duper` : this is Marsaglia's famous Super-Duper from the 70's. It has a period of about $4.6 * 10^{18}$ for most initial seeds. R uses the implementation due to Reeds et al (1982–84)
- `Mersenne-Twister` : it is a twisted GFSR with period $2^{19937} - 1$. In R, the initialization method due to B. D. Ripley is used.
- `Knuth-TAOCP-2002` : a 32-bit integer GFSR using lagged Fibonacci sequences with subtraction. The period is roughly 2^{129}
- `Knuth-TAOCP` : an earlier version of the algorithm due to Knuth (1997). This generator is written in interpreted R code
- `L'Ecuyer-CMRG` : a combined multiple-recursive generator from L'Ecuyer (1999). The period is around 2^{191}
- `user-supplied` : use a user-supplied generator

Diehard Battery of Test of Randomness

- a collection of complete statistical tests for random number generators
- initiated by G. Marsaglia
- original version in https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/
- updated version: https://webhome.phy.duke.edu/~rgb/General/dieharder.php
- an R package exists: RDieHarder: An R interface to the DieHardersuite of RandomNumber Generator Tests

Some of the tests

- Birthday spacing
- Overlapping Permutations
- Ranks of matrices
- Monkey tests
- Count the 1s
- Parking lot test
- Minimum distance test
- Random spheres test
- The squeeze test
- Overlapping sums test
- Runs test
- The craps test

Generating from a probability distribution

- this is a fundamental aspect of all Monte Carlo methods
- given a sequence $\{X_n\}$ of pseudo-random numbers between 0 and X_{max}, it is always possible to re-scale them between 0 and 1 as follows: $u_j = X_j / X_{max}$
- four basic methods are used:

i) inverse transform method

ii) composition method

iii) acceptance/rejection method

iv) ratio-of-uniforms method

The inverse transform sampling method

- it's a direct method and it is based on the following facts:

1) all cumulative distributions are monotone increasing functions in the interval $[0, 1]$
2) if the analytical form of $F(X)$ is known, it is also invertible:

$$F^{-1}(y) = \inf\{x : F(x) \geq y\} \quad u \in [0, 1]$$

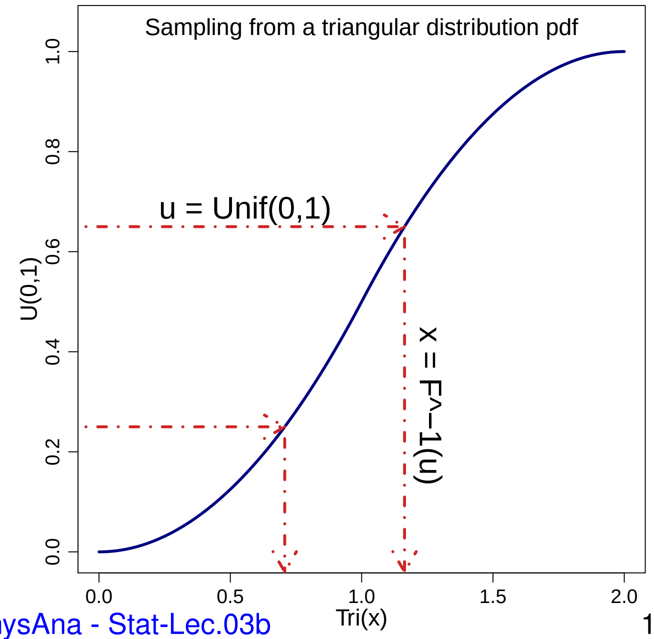
3) there is a 1:1 correspondence between CDFs, since they have the same image

- given X and Y with CDFs $F(X)$ and $G(Y)$
- we ask for the same probability, and search for x_i and y_i such that

$$F(x_i) \equiv P(X \leq x_i) = G(y_i) \equiv P(Y \leq y_i)$$

- assuming

$$\begin{aligned} G(y) &= \mathcal{U}(0,1) = u \\ &\rightarrow F(x_i) = u \\ &\rightarrow x_i = F^{-1}(u) \end{aligned}$$



The inverse transform sampling method - ex 1

Algorithm

1) generate $u \in \mathcal{U}(0, 1)$
2) compute $X = F^{-1}(u)$
3) release X, as it follows $X \sim F(x)$

Exercise 1

- generate random numbers from $\mathcal{U}(a, b)$
- the probability density and cumulative functions are

$$f(x) = \frac{1}{b-a} \qquad a \leq x \leq b \qquad F(x) = \frac{x-a}{b-a}$$

- we generate $u \in \mathcal{U}(0, 1)$

$$u = \frac{x-a}{b-a} \qquad \Rightarrow \qquad x = a + u(b-a)$$

The inverse transform sampling method - ex 2-3

Exercise 2

- generate random numbers from $f(x) = 2\,x$ with domain $[0, 1]$
- we evaluate the cumulative density function as

$$F(x) = \int_0^x 2\,y\,dy = x^2 \text{ for } 0 \le x \le 2$$

  - we generate $u \in \mathcal{U}(0, 1)$

$$u = x^2 \qquad \Rightarrow \qquad x = \sqrt{u}$$

Exercise 3

- generate random numbers from $\mathrm{Exp}(\lambda)$

$$f(x) = \lambda \mathrm{e}^{-\lambda\,x} \qquad F(x) = 1 - \mathrm{e}^{-\lambda\,x}$$

  - we generate $u \in \mathcal{U}(0, 1)$

$$\begin{aligned} u &= 1 - \mathrm{e}^{-\lambda\,x} \\ \mathrm{e}^{-\lambda\,x} &= 1 - u = u \\ -\lambda\,x &= \ln u \\ x &= -\tfrac{1}{\lambda} \ln u \end{aligned}$$

u and $1 - u$ have the same probability distributions

Example: generating from a discrete distribution

- let's assume the probabilities assume discrete values:

$$f(X) = \begin{cases} C_j & x_{j-1} < x < x_j \\ 0 & \text{otherwise} \end{cases}$$

- we set

$$P_j = \int_{x_{j-1}}^{x_j} f(x)\,dx = \int_{x_{j-1}}^{x_j} C_j dx = C_j(x_j - x_{j-1})$$

- and

$$F_j = \sum_{k=1}^{j} P_k$$

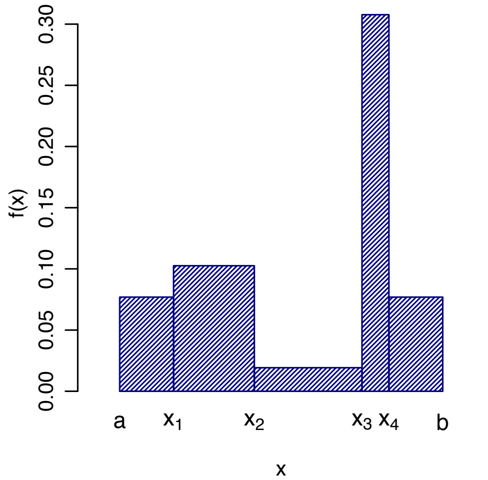
$$F(x) = \sum_{j=1}^{i-1} + \int_{x_{i-1}}^{x} C_j dx = F_{j-1} + C_i(x - x_{i-1})$$

- generating $u \in \mathcal{U}(0, 1)$, by inversion

$$u = F_{i-1} + C_i(x - x_{i-1})$$

$$x = x_{i-1} + \frac{u - F_{i-1}}{C_i}$$

Generating from a discrete distributionin R



Algorithm

- generate random numbers from $\mathcal{U}(0, 1)$
- find i such that

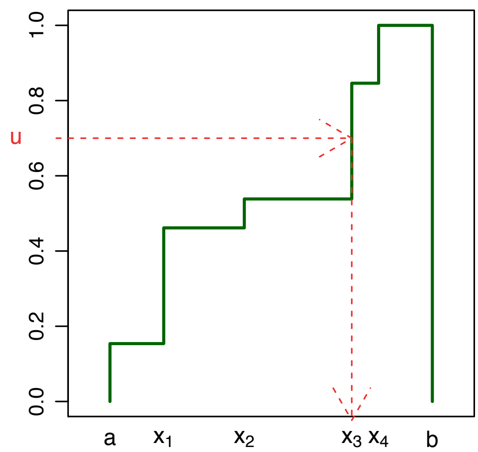
$$\sum_{j=1}^{i-1} P_j \le u < \sum_{j=1}^{i-1} P_j$$

- deliver $x = x_{i-1} + (u - F_{i-1})/C_i$

Example

- from u we find $i = 3$
- $x = x_2 + (u - F_2)/C_3 = x_2 + (u - 6)/1 \;\Rightarrow\; u = 9/13$

$C_1 = 2 \quad C_2 = 4 \quad C_3 = 1 \quad C_4 = 4 \quad C_5 = 2$
$F_1 = 2 \quad F_2 = 6 \quad F_3 = 7 \quad F_4 = 11 \quad F_5 = 13$



The Composition sampling method

- it is based on the fact that our pdf can be written as linear combination of other pdfs

$$F(x) = \sum_{j=1}^{r} \omega_j F_j(x)$$

- with

$$0 < \omega_i < 1 \quad \text{and} \quad \sum \omega_i = 1$$

Algorithm

1) generate $u \in \mathcal{U}(0, 1)$
2) according to the weights, ω_i, extract the correct index j
3) generate x from $F_j(x)$

Example with the Composition sampling method

- we want to sample from the pdf

$$f(x) = \frac{5}{12}\left[1 + (x-1)^4\right] \quad \text{with} \quad 0 \le x \le 2$$

- we can rewrite it as follows

$$f(x) = \frac{5}{6} f_1(x) + \frac{1}{6} f_2(x)$$

- therefore, $\omega_1 = 5/6$ and $\omega_2 = 1/6$ with $\omega_1 + \omega_2 = 1$

$$f_1(x) = \frac{1}{2} \quad \Rightarrow \quad F_1(x) = \int_1^x \frac{dx}{2} = \frac{x}{2}$$

$$f_2(x) = \frac{5}{2}(x-1)^4 \quad \Rightarrow \quad F_2(x) = \int_1^x \frac{5}{2}(x-1)^4\, dx = \frac{(x-1)^5}{2} + \frac{1}{2}$$

Algorithm

- generate $u_1, u_2 \in \mathcal{U}(0,1)$
- if $u_1 < 5/6$, $\Rightarrow x = 2\, u_2$
- else $\Rightarrow 2u_2 - 1 = (x-1)^5 \Rightarrow x = (2\, u_2 - 1)^{1/5} + 1$

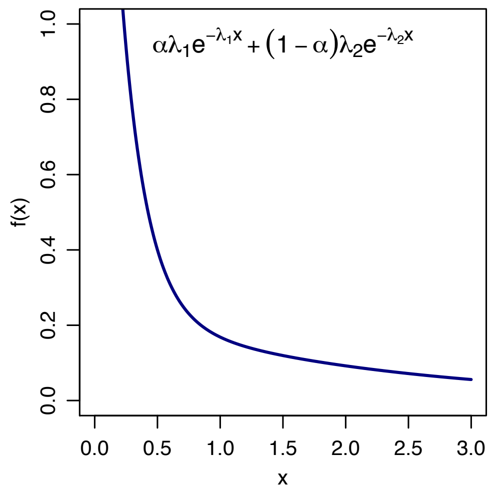
Example with the Composition sampling method

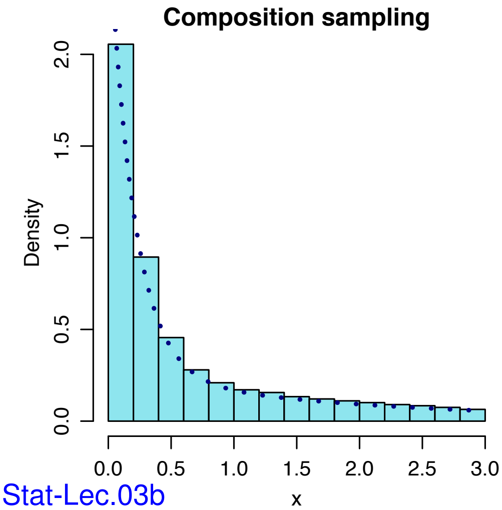
- we want to sample from two exponential distributions

$$f(x) = \alpha \lambda_1 \mathrm{e}^{-\lambda_1 x} + (1-\alpha)\lambda_2 \mathrm{e}^{-\lambda_2 x}$$

- the weights are: $\omega_1 = \alpha$ and $\omega_2 = 1 - \alpha$, with $\omega_1 + \omega_2 = 1$

$$F_1(x) = 1 - \mathrm{e}^{-\lambda_1 x} \quad \text{and} \quad F_2(x) = 1 - \mathrm{e}^{-\lambda_2 x}$$





Algorithm

- generate $u_1, u_2 \in \mathcal{U}(0,1)$
- if $u_1 < \alpha$, $\Rightarrow x = \ln u_2/\lambda_1$
- else $x = \ln u_2/\lambda_2$

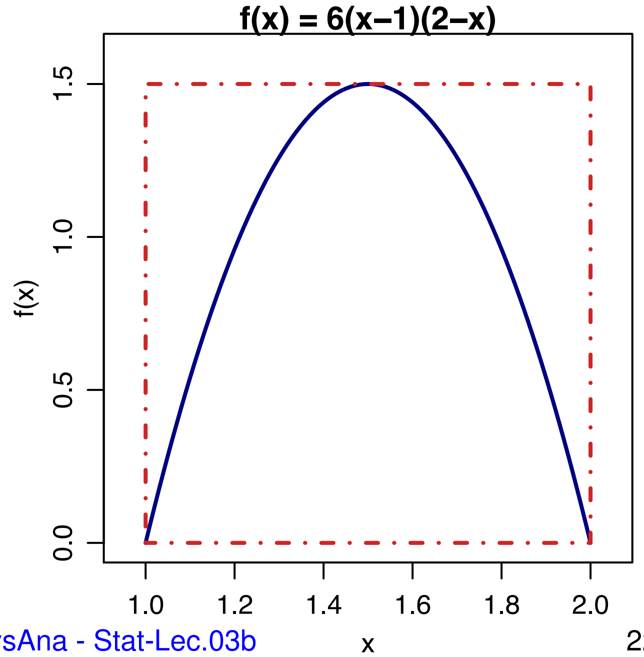
The acceptance/rejection method

- this is very useful when we are not able to compute the analytical form of the CDF
- or when the CDF is not easily invertible

- the method, due to von Neumann in 1951, is based on the hypothesis that our pdf is defined analytically in the interval $[a, b]$ and that $\forall x \in [a, b] \rightarrow f(x) < M$

Algorithm

- generate $u_1 \in \mathcal{U}(0, 1)$
- compute $x_1 = a + (b - a) \cdot u_1$
- generate $u_2 \in \mathcal{U}(0, 1)$
- if $u_2 \cdot M < f(x_1)$ we accept and release x_1
- otherwise we restart the algorithm



The acceptance/rejection method

- the efficiency of the method is given by the ratio of the two areas

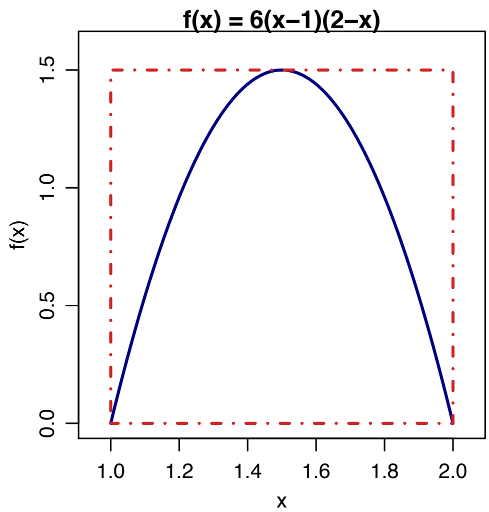
$$\epsilon = \frac{\int_a^b f(x)\, dx}{M\,(b-a)} = \frac{1}{M\,(b-a)}$$

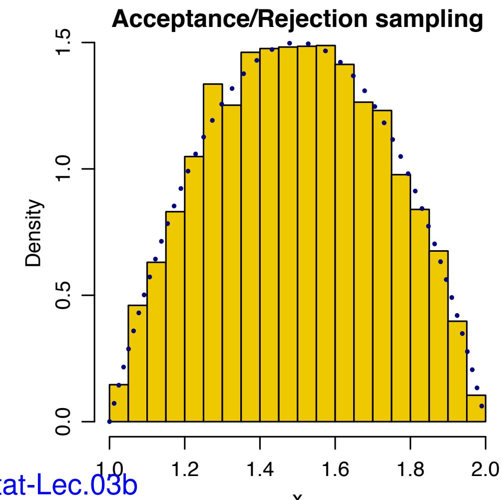
```
a <- 1; b <- 2
f.1 <- function(x) {6*(x-1)*(2-x)}

n <- 10000
u.1 <- runif(n, a, b)
u.2 <- runif(n, 0, 1)
f.max <- 1.5
y <- ifelse(u.2 * f.max < f.1(u.1), u.1, NA)
y.clean <- y[!is.na(y)]

hist(y.clean, breaks=seq(1,2,0.05), freq=FALSE,
        col="gold2", xlim=c(1, 2), xlab="x",
        main='Acceptance/Rejection␣sampling')
curve(f.1, col='navy', lt=3, lw=3, add=TRUE)

efficiency <- length(y.clean)/length(y)
cat(paste("efficiency:␣", efficiency, '\n'))
```





Example: sampling from a disc 1

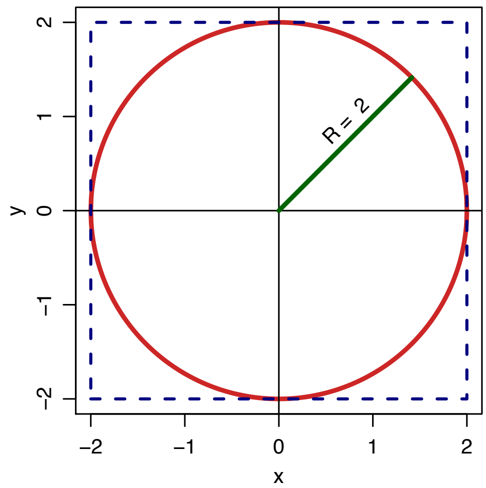
- we want to sample, uniformly, inside a disc of radius R
- i.e. sample points (x_j, y_j) such that $x_j^2 + y_j^2 \leq R$

Acceptance/Rejection sampling algorithm

- generate $u_1, u_2 \in \mathcal{U}(0, 1)$
- compute $x_j = R(1 - 2u_1)$ and $y_j = R(1 - 2u_2)$
- if $x_j^2 + y_j^2 \leq R$, accept and release (x_j, y_j)
- otherwise we restart the algorithm

- the efficiency of the method is given by the ratio

$$\epsilon = \frac{A_{disc}}{A_{square}} = \frac{\pi R^2}{4R^2} = \frac{\pi}{4}$$



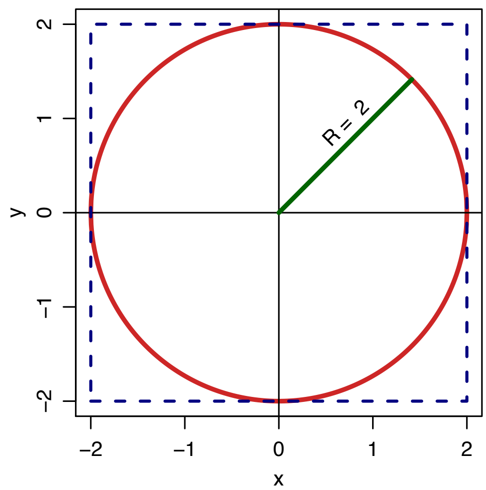
Example: sampling from a disc 2

- the alternative is to change from Cartesian to polar coordinates

$$\begin{cases} x_j &= R\cos\theta_j \\ y_j &= R\sin\theta_j \end{cases}$$

- the probability for a point (x_j, y_j) to be at a distance $r + dr$ from the disc center is

$$F(r) = \int_0^r f(\rho) d\rho = \int_0^r \frac{2\pi\rho d\rho}{\pi R^2} = \frac{r^2}{R^2}$$



Algorithm

- generate $u_1 \in \mathcal{U}(0, 1)$
- using the inverse transform, $u_1 = r^2/R^2 \Rightarrow \hat{r} = R\sqrt{u_1}$
- generate $u_2 \in \mathcal{U}(0, 1)$
- compute $\hat{\theta} = 2\pi u_2$
- evaluate

$$\begin{cases} x_j &= \hat{r}\cos\hat{\theta} \\ y_j &= \hat{r}\sin\hat{\theta} \end{cases}$$

this method has 100% efficiency, but computations are heavier since trigonometric functions are required

Normal distribution - Box-Müller

- if $X \sim \mathrm{Norm}(\mu, \sigma^2)$, the pdf is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$$

- the inverse-transform method is inefficient (we do not have an analytical CDF)
- to simplify we can sample $X \sim \mathrm{Norm}(0,1)$ and then transform $Z = \mu + \sigma X$

The Box-Müller algorithm

- let's consider the pdf of two independent normal distributed random variables $\Rightarrow$ (X, Y) is a random point in the plane
- let's move to polar coordinates (r, θ)
- the joint pdf becomes

$$f(r,\theta) = \frac{1}{2\pi} r \cdot \exp -\frac{r^2}{2}$$

- since $x = r\cos\theta$ and $y = r\sin\theta$

$$f(x,y) = \frac{1}{2\pi} r \cdot \exp \frac{-(x^2+y^2)}{2}$$

1. generate two independend random variables, $U_1, U_2 \in \mathcal{U}(0,1)$
2. release

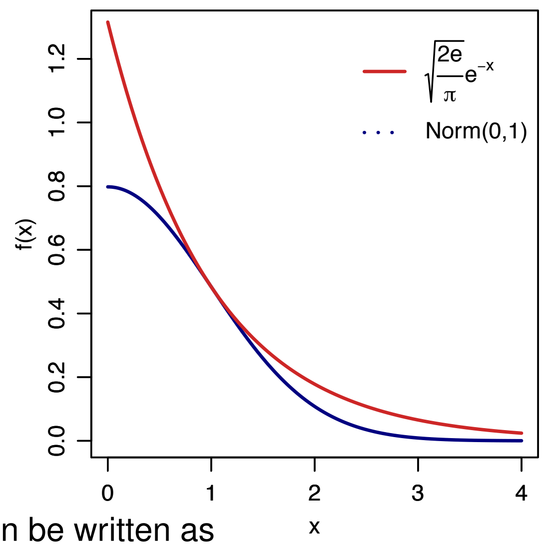
$$X = \sqrt{-2\ln U_1}\cos 2\pi U_2 \quad \text{and} \quad Y = \sqrt{-2\ln U_1}\sin 2\pi U_2$$

Normal distribution - Acceptance/Rejection

- an alternative method to generate from $X \sim \mathrm{Norm}(0,1)$ is based on the acceptance/rejection method
- let's generate from the pdf

$$f(x) = \sqrt{\frac{2}{\pi}} \exp -x^2/2 \quad \text{with} \quad x \geq 0$$

- (the sign can be generated with another $\mathcal{U}(0,1)$)
- we bind $f(x)$ by $C \cdot g(x)$ where $g(x) = \exp(-x)$
- the smallest constant such that $f(x) \leq C \cdot g(x)$ is $C = \sqrt{2e/\pi}$



- the acceptance condition $U \leq f(X)/(C\exp -X)$ can be written as

$$U \leq exp\left[-(X-1)^2/2\right]$$

- which is equivalent to

$$-\ln U \geq \frac{(X-1)^2}{2} \quad \text{with} \quad X \sim \mathrm{Exp}(1)$$

- but $-\ln U$ follows from $\mathrm{Exp}(1)$, therefore the inequality can be rewritten as

$$V_1 \geq \frac{(V_2-1)^2}{2} \quad \text{with} \quad V_1 = -\ln U \text{ and } V_2 = X$$

- both V_1 and V_2 are independent and $\mathrm{Exp}(1)$ distributed