Informatics

Random Numbers Generation

Claudio Sartori
Department of Computer Science and Engineering
claudiosartori@uniboit
https://wwwuniboit/sitoweb/claudiosartori/

Learning Objectives

- Randomness: what's for?
- Randomness and computers?
- The principles
- Random generation in R
- Examples

Random numbers

- used in
  - statistics
  - programming
    - simulation
    - games
    - program testing
- tools
  - *tables of random numbers*
  - hardware generation
  - software generation

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	8	0	9	4	2	5	2	5	8	2	4	7	1	3	4	7
2	3	5	6	3	2	1	9	8	8	2	1	1	9	0	4	5
3	1	3	3	0	6	3	3	1	3	7	5	3	9	6	9	3
4	3	5	6	5	0	0	1	6	2	2	4	3	6	4	3	2
5	7	8	5	0	5	9	2	6	5	5	8	8	7	3	1	1
6	4	4	9	0	5	4	1	7	9	7	2	7	6	1	5	3
7	6	6	4	6	9	1	0	4	9	3	1	8	8	8	1	9
8	3	6	2	6	5	9	9	5	1	2	1	5	9	7	5	3
9	4	6	6	5	4	8	2	0	7	5	5	4	0	6	1	2
10	6	4	9	8	7	5	1	9	0	4	7	4	7	8	1	8
11	6	7	2	2	9	8	6	9	9	3	6	1	7	8	7	5
12	9	7	4	8	5	9	3	2	5	1	1	5	2	7	2	1
13	5	6	4	1	1	4	1	7	1	4	1	9	7	4	3	4
14	7	4	4	4	9	2	0	0	8	8	4	0	5	8	8	2
15	8	2	7	9	3	0	1	9	4	6	7	2	3	7	4	3
16	0	1	6	1	7	6	1	7	1	0	2	4	2	3	8	7
17	7	3	8	8	9	7	5	9	7	5	5	5	6	6	2	4
18	7	8	3	0	4	7	1	4	3	6	9	5	2	9	1	9
19	9	8	8	7	4	2	1	6	6	5	2	6	4	5	3	5
20	1	2	6	1	2	5	1	6	8	5	6	9	2	3	1	0
21	3	9	4	7	4	9	3	7	7	6	3	4	2	5	4	3
22	4	5	5	0	8	1	0	3	1	2	5	0	2	3	0	4
23	1	3	4	4	9	6	9	7	2	3	8	3	6	9	7	6
24	8	9	7	6	5	8	2	3	8	4	8	7	0	4	6	0
25	7	7	1	0	9	9	4	3	6	9	7	8	8	2	7	3
26	6	9	5	9	6	0	0	8	8	4	4	2	2	2	8	2
27	7	9	4	1	2	3	1	2	2	4	3	1	6	7	0	2
28	2	2	8	4	0	8	9	6	9	1	0	7	5	5	4	2
29	9	5	9	4	7	4	1	6	9	3	6	5	6	0	4	5
30	4	6	1	3	8	5	4	9	6	3	6	9	3	2	0	8

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	7	4	3	3	3	6	2	0	1	8	9	7	2	1	3	4
2	2	6	1	8	2	7	5	1	2	6	2	7	1	0	9	5
3	8	7	3	8	6	6	1	5	1	5	3	8	8	5	4	3
4	4	7	9	6	6	0	9	5	5	2	8	3	1	6	2	0
5	2	1	9	2	4	6	4	5	3	5	3	0	5	5	8	9
6	5	9	0	1	4	8	7	8	9	9	8	0	9	8	7	7
7	7	5	3	7	2	7	8	6	9	3	7	3	2	4	4	5
8	9	2	2	3	5	6	5	8	2	9	4	4	2	8	9	9
9	9	6	8	3	4	2	5	1	9	1	3	8	1	7	0	9
10	6	8	3	2	9	6	8	3	9	8	7	2	4	0	9	0
11	4	8	8	8	1	3	1	5	9	6	7	9	8	8	3	4
12	0	0	3	3	9	3	0	3	9	7	1	3	4	0	1	2
13	8	1	6	5	7	3	6	8	1	2	1	8	5	0	3	9
14	4	3	9	8	3	9	0	4	9	1	9	9	9	3	3	6
15	3	9	7	9	4	6	8	9	9	0	2	1	6	9	9	0
16	2	8	9	1	6	6	7	7	1	5	8	5	2	4	8	2
17	9	9	7	7	2	0	0	8	5	5	9	6	9	7	4	0
18	1	8	0	4	4	0	4	4	1	0	3	4	2	5	9	7
19	8	4	3	0	5	2	7	0	9	6	0	5	0	7	6	8
20	3	9	3	9	8	7	0	3	9	8	4	1	0	3	5	3
21	6	2	3	9	7	4	5	5	2	0	5	5	7	7	9	5
22	1	1	3	8	9	7	8	8	9	1	4	4	4	5	2	6
23	6	2	5	1	4	2	0	1	2	0	3	8	6	5	5	2
24	3	1	0	6	9	1	6	6	2	7	1	7	7	6	0	1
25	9	7	1	4	9	7	0	0	1	5	6	6	2	8	8	9
26	1	5	2	4	2	5	1	7	5	8	1	8	0	0	8	1
27	9	9	8	4	3	4	6	9	3	0	8	5	4	7	6	2
28	7	3	1	9	3	7	8	2	1	0	6	8	9	5	7	4
29	1	1	8	3	5	9	1	6	9	5	9	9	1	1	4	3
30	5	1	0	9	9	6	8	0	1	1	6	8	6	1	3	3

Random numbers in statistics

- ***sampling***
  - select a sample of items from a larger population
  - either for impossibility to access the entire population or for faster computation
  - the sample must be representative of the population
- **simulation**
  - of complex systems or processes when formal modelling is not viable
- **Monte Carlo methods**
  - random sampling to solve complex problems in mathematics, physics, chemistry, engineering, finance

Problem description

- a software using random numbers requires a software generator
- a computer running any softwares is a *deterministic* machine
  - the output is *functionally determined* by *the input and the status*
- an algorithm can generate numbers that are *seemingly random*
- *pseudo-random* number generators
- *we deal only with pseudo-random generation, therefore the word "pseudo" will be omitted*

Requirements for a pseudo-random generator

- the perfect generator should be able to generate an infinite sequence of numbers, drawn from a given interval, that are statistically independent
- a generator should be
  - efficient
    - eg a simulation could require the generation of millions of numbers
  - repeatable
    - we want to be always able to repeat a scientific experiment

Lehmer generator (i)

- an example of algorithm for generating pseudo-random numbers
  - It is absolutely not the best one, simply it is one of the simplest, and a good example of implementation of random number generation
- proposed in 1951
- parametric algorithm
- generates a **permutation** of the natural numbers up to a given **m**
  - scanning the sequence of numbers of the permutation we obtain the effect of the single number generation
  - **m** is one of the parameters

Lehmer generator (ii)

- there are several choices of the parameters that guarantee a ***seemingly random sequence***
  - statistical tests give results *compatible with the hypothesis of randomness* of the generation
  - each number of the sequence *seems to be independent* from the preceding portion of the sequence, i.e. observing a sequence of generated numbers it is hard to guess next number

Lehmer generator - description

Given

1. modulus: m integer, prime, *big*
2. multiplier: a integer, $1 < a < m$
3. generator $f(z)$: $z_{n+1} = a * z_n \mod m$
4. seed: z_1 integer, $1 \leq z_1 \leq m-1$

Lehmer generator - discussion

- since *m* is prime, the generator does never generate 0, for any $1 \leq z \leq m-1$, therefore the sequence does never collapse to 0
  - otherwise there would exist a1 * m1 * z1 * m2 mod m1 * m2 = 0
- linear transformations of the sequence do not influence the apparent randomness
- the sequence is fully deterministic, but there are many choices for *a* and *m* giving sequences that seem perfectly random
- the values of *a* and *m*, determine the length of the period *p* ($p<=m$), such that $z_p = z_1$
- a *complete period sequence* is a *permutation* of the numbers 1,…,m-1
- there are several pairs *a* and *m* giving complete period sequences
- each number has probability 1/m
- the seed determines the starting point of the sequence
  - Changing the seed we simulate the effect of a different sequence, due to the apparent independence of the numbers in the sequence

Example: a=6, m=13

f(z) = 6z mod 13 1,6,10,8,9,2,12,7,3,5,4,11,1…

period

Parameter choice

- a long period is obviously preferred
- with $m = 2^{31}-1$ there exist 534 of good values for a
- an efficient implementation of f(z) is needed
- $a=16807$ and $m = 2^{31}-1$ is a good choice
  - it requires to manage integers with 46 bit, to contain the maximum value of $a * z$
- the seed can be chosen freely for each experiment

Some of the methods available

- Lehmer
- Middle Square
- Linear Congruential (LCG)
- Quadratic Congruential
- Inverse Congruential (ICG)
- Inverse Congruential Explicit (EICG)
- ICG and EICG composed
- Fibonacci delayed
- Shift register with linear feedback
- Mersenne Twister (*default generator in R*)
  https://en.wikipedia.org/wiki/Mersenne_Twister
- …

ony for general information

Verification of randomness

- uniformity of the distribution in the interval
  - easy to obtain and verify
- independence
  - difficult to obtain and verify
- verification criteria
  - statistical tests
  - theoretical analysis of the algorithm

Verification of randomness (ii)

- uniformity
  - Chi-Square uniformity test
  - Kolmogoroff-Smirnoff
- independence
  - Chi-Square independence test
  - "gap" test
  - …

Random numbers in R

- integer numbers
- real numbers
- uniform
- standard probability distributions
- sampling among given values
- …

The seed

- setting the seed allow to reproduce exactly the random sequence
- it is a good habit to set the seed, and to keep track of the seed used, at the beginning of an experiment
- in this way the experiment can be repeated with constant results

```
set.seed(integer)
```

Workflow for using random values

Repeatability:

- every time you re-execute *the entire script* you will see the same random values
- if you execute single statements without re-executing the seed setting the repeatability is not guaranteed

```
rnd_seed <- 745
set.seed(rnd.seed)
# generate random values
# . . .
# use random values
# . . .
```

Uniform: continuous

```
runif(n number of generated double values
     , min = 0
     , max = 1 interval of generated values
)
```

Discrete: integers

```
sample.int(n generates values from 1 to n
            , size = n #number of generated values
            , replace = FALSE when generating
              more than one value controls
              repetition of values
            , prob = NULL if not set the
              generation is uniform, otherwise
              values probabilities are given as
              a vector of n weights
            )
```

Discrete: general

```
sample (x vector of generated values
          , size #number of generated values
          , replace = FALSE when generating
            more than one value controls
            repetition of values
          , prob = NULL if not set the
            generation is uniform, otherwise
            values probabilities are given as
            a vector of n weights
          )
```

Some random distributions in R

beta: dbeta

binomial (including Bernoulli): rbinom

Cauchy: dcauchy

chi-squared: dchisq

exponential: rexp

F: df

gamma: dgamma

geometric: dgeom

This is also a special case of the negative binomial

hypergeometric: dhyper

log-normal: dlnorm

multinomial: dmultinom

negative binomial: dnbinom

normal: dnorm

Poisson: dpois

Student's t: dt

uniform: dunif

Weibull: dweibull

Quiz

- random integers in two separate intervals
- the hidden mines of a Minesweeper schema
- random points in a given rectangle
- random points inside a circle, given center and radius
- random elements of a data frame
- random letters with uniform distribution and replacement
- a Ruzzle schema

Montecarlo methods (a few basics)

(Source: wikipedia)

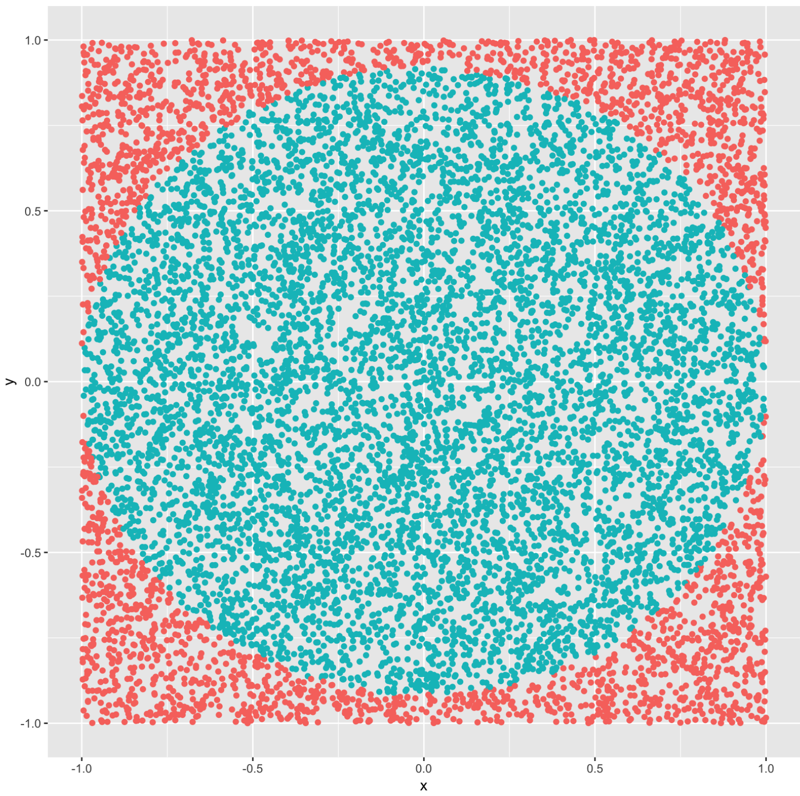
- Computational algorithms that rely on repeated random sampling to obtain numerical results
- Useful for physical and mathematical problems for which an analytic solution is difficult for any reason
- Non-deterministic approach
  - approximation with error
  - several trials
- Examples:
  - optimization
  - numerical integration
  - generating draws from probability distribution

General method (naive explanation)

- Find a method for simulating some situation
- Simulate with a high number of repetitions
  - more repetitions ➔ better precision
- Count the fraction of *success*
- Derive from that fraction the result

A toy example: computation of **PI**

- Generate random points in a square bounding a circle
- Compute the frequency of points that are inside the circle
- PI is four times the ratio between the total number of points and the number of points inside the circle



A toy example: computation of **PI**

- repeat several times with different number of points
  - repeat several times without setting the seed and with the same parameters
  - compute the average of the results and store it
  - this repetition tries to compensate the (pseudo) *non-determinism*

See the example "`montecarlo_pi`"