STAT 513/413: Lecture 8
Computer generated... random numbers?

(the creed of Monte Carlo)

Rizzo, Chapter 3

What is the opposite of random?

The opposite of random - fully within reach/grasp - is called *deterministic*

Note: "fully within reach/grasp" does not mean "we are fully capable to control it", only "we are fully capable of understanding it" - somehow...

For instance, it may be believed that weather is fully deterministic: there is an equation governing all the physics, which does not account for any randomness. The catch, however, is that it sensitive even to small change in initial conditions. That gave rise to a phenomenon called "butterfly effect": a distant butterfly flapping its wing can several weeks later influence the weather: cause a tornado, say

Nonetheless, computers are still believed to be deterministic: just doing what they are being told.

(We are the robots, song of Kraftwerk. In the middle: I am your servant, I am your worker. In Russian, in Cyrillic.)

Really?

Well, that was about the old computers. They were really like that: obedient and stupid. But now?

Well, students of programming are still advised: when your program does not work as anticipated, it still contains an error (it is called bug). If you are unable to find it right away, maybe a short or even longer break will help; but beware: simply by taking break, the bug will not go away. When you start the computer next time, it will be still there...

Yes, on the other hand...

Worst disasters of 2020

Next to coronavirus: Apple operating system Catalina

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

...

[R.app GUI 1.73 (7892) x86_64-apple-darwin17.0]

[Workspace restored from /Users/mizera/Desktop/Lessons/513/.RData]

[Workspace restored from /Users/mizera/Desktop/Lessons/513/.RData]
[History restored from /Users/mizera/Desktop/Lessons/513/.Rhistory]

2021-01-25 17:42:32.639 R[3745:118806] Warning: Expected min height o
view: (<NSPopoverTouchBarItemButton: 0x7fd4cd9db9c0>) to be less than
or equal to 30 but got a height of 32.000000. This error will be
logged once per view in violation.
```


What the heck is this?

But still, computers are at least somewhat "conditionally deterministic"

So, "deterministic" may not be true for the machines as a whole, and in particular for those connected to the internet. (My engineer friend says: every circuit with more than three transistors has a soul.)

But locally - inside R, say, once it is launched, and conditionally on no outside influences, like blackouts and similar - one would say that the above advice about bugs that will not go away by themselves is sound. (Also in this course.) So we can consider what happens in the computer - at least within certain environment, like R - is deterministic.

(It is like cars. We consider them rather deterministic than random.)

Except for the fact that in many programming environments - in R too - there exist functions that return "random numbers"

Let us try it

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
...
> runif(6)
[1] 0.7811872 0.2256249 0.1029852 0.9418297 0.1088203 0.7156732
```

Now, quit R and start it again:

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
...
> runif(6)
[1] 0.7370521 0.4887825 0.2355647 0.1671979 0.9691297 0.2913628
```

Well, they *are* random, are they not?

Well, it depends

Let us try in that this way now:

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
...
> set.seed(007)
> runif(6)
[1] 0.98890930 0.39774545 0.11569778 0.06974868 0.24374939 0.7920104
```


Again, quit and restart R:

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
...
> set.seed(007)
> runif(6)
[1] 0.98890930 0.39774545 0.11569778 0.06974868 0.24374939 0.7920104
```

And there is not one, but several functions returning "random numbers"

Let us do a bit of 0-1 coin tossing again. To obtain something in this vein, we can do

```
> floor(2*runif(32))
 [1] 0 1 1 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0
```


But we can also do


```
> rbinom(32,1,0.5)
 [1] 1 0 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 0
```


And yet another way is

```
> sample(c(0,1),32,replace=TRUE)
 [1] 1 1 1 0 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1 1 1 1 0 0 1 1 0 1
```

All these functions return "random numbers"!

Digression:
recall the sequences from the previous lecture

```
> seq1
[1] 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
> seq2
[1] 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 1 0 1 1
> seq3
[1] 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0
> seq4
[1] 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 1 1 0 1 0 1 0
> seq5
[1] 1 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 1 0 1 0 1 0 1 0 0 0 1 0
```

Back: do all the three ways return the same thing? Apparently not

Well, depends... The first two do, but the third is different

```
> set.seed(007)
> floor(2*runif(32))
 [1] 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0
> set.seed(007)
> rbinom(32,1,0.5)
 [1] 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0
> set.seed(007)
> sample(c(0,1),32,replace=TRUE)
 [1] 1 0 0 1 0 1 0 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 1 1 0 0 1 1 1 0 0 1
```


So, what is going on here?

Digression: funny what I obtained a year ago

```
> set.seed(007); a=floor(2*runif(1000000))
> set.seed(007); b=rbinom(1000000,1,0.5)
> set.seed(007); c=sample(c(0,1),1000000,replace=TRUE)
> sum(abs(a-b))
[1] 0
> sum(abs(a-c))
[1] 0
```


(Explanation: they already said back then that `sample()` has deficiencies and needs to be changed. Apparently, they did it.)

Random numbers in computers: how

There may be several functions of various type in a given programming environment; all, however, derive their random numbers from a certain "proto-function" called

random number generator

In other words: whenever any function has to return a random number (say, with specified distribution or other attributes), it requests one or several random numbers from the random number generator, and then processes it as necessary

Regarding (the proto-function) random number generators: in most systems (including R) they generate a random number between 0 and 1, with uniform distribution: every number in that interval has equal probability. (Only in some systems, random number generators return rather random 0-1 sequences instead (with both outcomes, 0 and 1, having the same probability, 1/2)

And now: the secret revealed

The random number generator - and thus all subsequent random numbers produced by a computer is in fact a *completely deterministic* recursive algorithm: once we know its *state* (a number, or a vector of numbers), we can determine all the states that will follow and all numbers that will be generated. Each time a random number is generated, it is thus like taking it out of a certain predetermined list. This recursive, deterministic nature is why they are not considered "really" random – they only "appear random" (Knuth). Because of this, the adjective "random" is sometimes replaced by "pseudo-random".

Nonetheless, we prefer simple "random", as that makes for less typing - and is also not that contradictory with our "beyond reach" definition of randomness

And, as their design improved very much over the years many of them really appear random quite well

Their determinism is not always a bad thing: if we initialize the generator in a certain way, we obtain each time the same sequence: this is convenient, for instance, when we still work on the algorithm ("debugging")

Hardware random generators

Long time ago, when computers just started, there were constructed also "hardware random number generators" (like, small people inside tossing coins, etc.) They were "truly random", in the sense that nobody could predict the outcome. They were abandoned, however, because they were outpaced by computers: they turned out to be way too slow for compared to processors.

(Also, they were irreproducible - which is not always good; sometimes you would like to debug.)

Thus, deterministic algorithms prevailed as random number generators, algorithms that generate random numbers "in the state of sin" (von Neumann), recursively: each random number depends on the previous one(s). This turned out fast enough.

State of the generator: saving and retrieving

In R, the state of the random number generator is kept in an "invisible" vector `.Random.seed`

```
> ls()
[1] "a" "b" "x"
> ls(all=TRUE)
[1] ".Random.seed" "a"            "b"            "x"
> .Random.seed
  [1]         403           6  1384307324  -971116865  2006898581
...
[622]  1925606336  -709985239 -2145007461 -1987340948  -427022882
```


It is recommended that users do not manipulate `.Random.seed` directly - except for saving it to another variable and then possibly retrieving it back. Once saved (at the beginning of a session, say), and then the saved value assigned back into `.Random.seed`, everything "starts over" - resulting in the same sequence of random numbers. If `.Random.seed` is saved *at the end* (of a session, say) and next time (say, at the beginning of another session) assigned back, then the random generator "takes over where it stopped last time"; this may be desirable when we want next random numbers to be really different from the previous ones

Function `set.seed`, randomization

A convenient way to control `.Random.seed` is function `set.seed()`. This command has one argument. When it is an integer then it sets the state of the random number generator - to be the same each time the same integer is used (in the same or another session); different integer arguments, however, result in different states.

If the argument of `set.seed()` is `NULL`, then it "randomizes" the random number generator: it initializes it using data from the system clock, thus making its subsequent input considerably more random. This is convenient when we want to run the experiment in an "impartial" fashion, initializing the random number generator in an "objective" way. The same effect is achieved by leaving and reentering R, or by deleting `.Random.seed` through executing `rm(.Random.seed)`: once `.Random.seed` is deleted, it is immediately replaced by a new one "created from the current time and process ID".

The Good Manners of Monte Carlo

First, we are debugging, so then we would like to have always the same sequence of random numbers

```
> set.seed(1)
> runif(6)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
> runif(6)
[1] 0.94467527 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675
> set.seed(1)
> runif(6)
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
```


Once we consider the algorithm correct, it is ready for the actual "computer experiment"; to run it in an "impartial fashion", we randomize the random number generator using one of the ways described above

```
> set.seed(NULL); runif(6)
[1] 0.43856362 0.63981600 0.34235782 0.04603239 0.99229755 0.44843181
```

Nonetheless, even in this last run it is advised to save and keep the seed - just in case something goes wrong

And practically in this course

Needless to say, you should always keep - that is, control - the seeds. However, for your final run, the run producing results to be submitted, *do not randomize*, but only

```
> set.seed(your_personal_number)
```

Random numbers in computers: why?

- computer experiments
- graphics
- perhaps also to mitigate "Buridan ass" situations
- and so-called Monte Carlo or simulation algorithms

Jean Buridan - 14th century French philosopher; the paradox of a donkey dying of hunger between two equally distant piles of hay was consider before by Aristotle (4th century BC, a man equally hungry and thirsty) and Al-Ghazali (11th century, two equally good dates)

Random number generators in R

To learn more about random number generators in R:

```
> ?RNG
> RNGkind()
[1] "Mersenne-Twister" "Inversion"
```

These are defaults. The second one specifies the way of generating random numbers with normal distribution - we will discuss that later. The first one gives you the type of the basic - uniform distribution on $(0, 1)$ - random generator: the options are "Wichmann-Hill", "Marsaglia-Multicarry": "Super-Duper", "Mersenne-Twister", "Knuth-TAOCP-2002", "Knuth-TAOCP", "L'Ecuyer-CMRG", and "user-supplied" - but that is something you really need not to know...

So, what "appear completely random" means here?

Well, that is a question... and we are not going to solve it here.

The best way to think about it is this way: you think of random numbers as outcomes of independent and identically distributed random variables with certain distribution. The computer ones are good, if they *practically* exhibit the same properties as the *true ones* would; and those "completely random" will have some kind of uniform distribution.

It is best to think about this in terms of coin tosses, sequences of 0's and 1's. What would be expect of those?

For instance, that both of them come about equally likely...

Or that pairs of them come equally likely...

Or... or... or... or...

The second volume, first part of The Art of Computer Programming by D. E. Knuth has a very nice discussion for those interested

Do they come equally likely?

```
> table(floor(2*runif(100)))

 0  1
57 43
> table(floor(2*runif(1000000)))

     0      1
499767 500233
> table(floor(2*runif(1000000)))

     0      1
500058 499942
> table(floor(2*runif(1000000)))

     0      1
499813 500187
```

How about the pairs?

```
> x=floor(2*runif(1000000))
> table(x[seq(1,length(x),2)]*10+x[seq(2,length(x),2)])

     0      1     10     11
124911 124923 125089 125077

> x=floor(2*runif(1000000))
> table(x[seq(1,length(x),2)]*10+x[seq(2,length(x),2)])

     0      1     10     11
125242 124509 124878 125371

> x=floor(2*runif(1000000))
> table(x[seq(1,length(x),2)]*10+x[seq(2,length(x),2)])

     0      1     10     11
124931 125273 124405 125391
```

How can I say they really are equally likely?

```
In chisq.test(x) : Chi-squared approximation may be incorrect
> x=floor(2*runif(1000000))
> z=table(x[seq(1,length(x),2)]*10+x[seq(2,length(x),2)])
> chisq.test(z)

Chi-squared test for given probabilities

data:  z
X-squared = 1.6493, df = 3, p-value = 0.6483

> x=floor(2*runif(1000000))
> z=table(x[seq(1,length(x),2)]*10+x[seq(2,length(x),2)])
> chisq.test(z)

Chi-squared test for given probabilities

data:  z
X-squared = 7.2551, df = 3, p-value = 0.0642
%%% but I didn't set or keep the seed!!!
```

Stanislaw Ulam (1909-1984)

Going to Monte Carlo: his uncle was always borrowing money

And finally: The Creed of Monte Carlo

For all practical purposes - we believe:

that the computer-generated random numbers obey all probability laws

which derives from the fact that - as we believe - they follow the distributions they are supposed to

and the fact that - as we believe - that in one uninterrupted (by manipulating with seed) sequence they behave like independent random variables

That is, we work with series of computer generated random numbers as with series of independent random variables with given distribution, with all probabilistic consequences - and thus from now on we can truly omit "pseudo-" and call them just "random" (if their character will need to be emphasized, we will call them "computer random numbers")