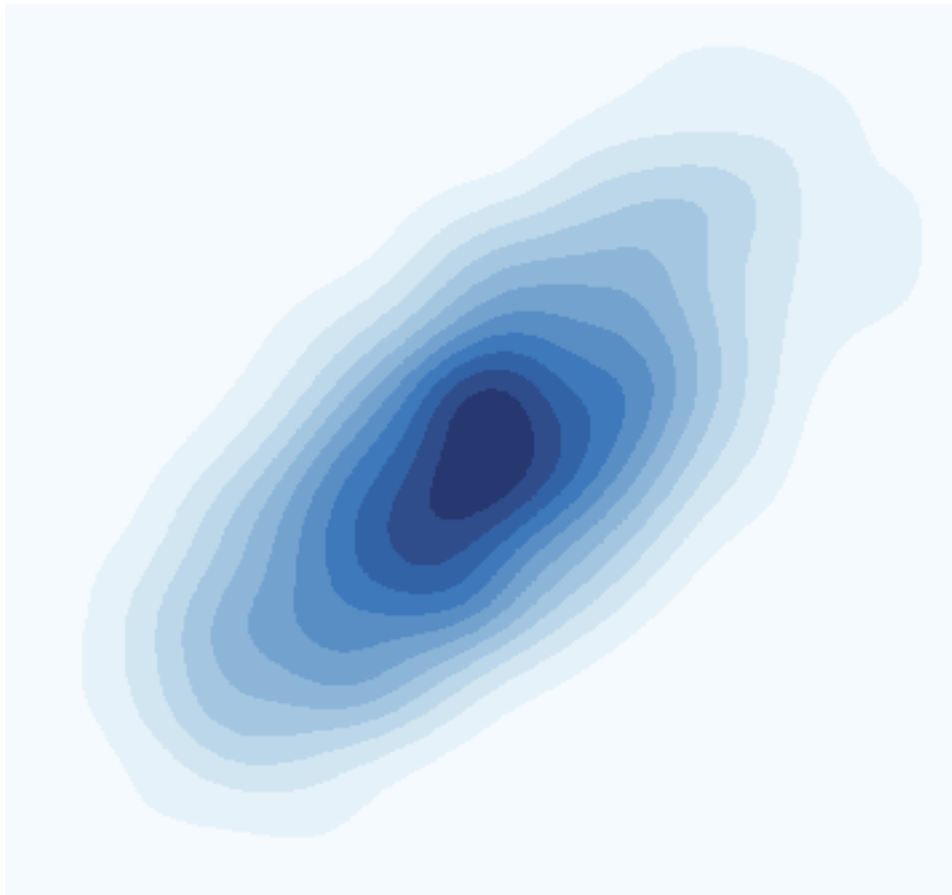


# Cables Sous-Marins

NEEL Pauline, PETROS Samson, RAKOTOVAO Jonathan, VOYLES Evan

15 Mars 2022



Any one who considers  
arithmetical methods of  
producing random digits is, of  
course, in a state of sin

---

*John von Neumann*

## 1 Génération des variables aléatoires

Un ordinateur n'est qu'un gros agencement complexe des circuits. Regnés par les lois physiques, les opérations provenant du mouvement des électrons sont encodés par les fonctions booléennes. Les *fonctions* - dans le sens mathématique - sont des objets purement déterministes. Autrement dit, une fonction associe à une donnée d'entrée une valeur unique appartenant à l'espace d'arrivée. Si on lui donne plusieurs fois la même valeurs, par exemple  $x_1$  et  $x_2$  tels que  $x_1 = x_2$ , la définition d'une fonction oblige que  $f(x_1) = f(x_2)$ . D'où vient l'énigme. Comment générer des variables aléatoires quand il reste à notre disposition que des méthodes déterministes?

On n'ose pas à répondre à cette question plus éloquentement que celle-ci offert par John von Neumann, l'un des meilleurs mathématiciens, pionniers, informaticiens de tous les temps; générer des variables aléatoires sur un ordinateur n'est tout simplement pas possible. Cependant, cela ne nous empêchera pas d'essayer quand même.

## 2 Loi uniforme

On commence notre projet en étudiant la loi la plus simple parmi les lois usuelles - la loi uniforme. Non seulement parce qu'elle est simple, mais en plus à partir de cela, nous allons pour construire des algorithmes plus complexes (notamment la méthode de la transformée inverse; la méthode de rejet) permettant de simuler des lois différentes comme la loi normale, la loi exponentielle, etc. Il s'agira principalement d'échantillonner une variable  $X \sim \mathcal{U}(0, 1)$  puis ensuite effectuer des manipulations mathématiques pour produire une variable suivant une autre loi ciblée.

Alors sans plus tarder, on formalise nos objectives. Le principe est le suivant: nous allons générer une suite  $(x_n)$  à partir d'une graine  $x_0$  et une fonction  $f : \mathbb{R} \rightarrow \mathbb{R}$  telle que

$$\begin{cases} x_0 \in \mathbb{R} \\ x_n = f(x_{n-1}) \quad \forall n \in \mathbb{N} \end{cases}$$

On souhaite trouvé une fonction vérifiant certaines qualités désirées. Par exemple, on veut que notre fonction ait une période suffisamment long et qu'elle produise des valeurs très variés. Etudions la fonction  $x \mapsto (x + 1) \% 2$ , où  $\%$  est l'opération de modulus et on fixe une graine  $x_0 = 1$ .

$$\begin{aligned} f(x_0) &= (1 + 1) \% 2 = 2 \% 2 = 0 \\ f(x_1) &= (0 + 1) \% 2 = 1 \% 2 = 1 \\ f(x_2) &= (1 + 1) \% 2 = 0 \end{aligned}$$

Cette fonction produit alors la suite

$$\{1, 0, 1, 0, 1, 0, 1, 0, \dots\}$$

dont la période est 2 et évidemment dont les valeurs ne sont aussi variés qu'on le souhaite. On va parler plus tard en détail qu'est-ce que cela veut dire être "suffisamment variés"; pour l'instant on se contente de dire que cette suite-là n'atteint pas nos attentes. Heureusement pour nous, il y existe pleins de fonctions qui satisfont nos buts flous.

### 2.1 Générateur congruentiel linéaire

La méthode la plus directe à implémenter pour générer une telle suite est dit *générateur congruentiel linéaire*. Congruentiel parce qu'il s'agit d'une opération modulo et linéaire vu qu'il y est compris une transformation affine. Dans le cas général, on considère les fonctions de la forme:

$$f(x; a, c, m) = (ax + c) \bmod m.$$

D'ailleurs, la fonction étudiée dans la section précédente est un générateur congruentiel linéaire dont les paramètres  $a = 1$ ,  $c = 1$ , et  $m = 2$ . Ne vous inquiétez pas, ne pas tous les générateur congruentiel linéaire (LCG) sont fait de même qualité. Les choix des paramètres qui sont utilisés par des logiciels connus sont détaillé sur un page wikipédia et leurs propriétés déjà bien étudiés. Vu que l'objectif de notre projet c'est d'approfondir la connaissance autour des méthodes pour générer les variables (pseudo) aléatoires, on a décidé d'implémenter notre propre LCG hybride. Pour  $m$ , on choisit  $2^{31} - 1$ , un prime mersenne qui est très connu. Pour  $a$ , on s'amuse en choisissant 12345678. Finalement, on affecte l'incrémenteur  $c$  la valeur 1.

$$f_{\text{sousmarin}}(x) = (12345678x + 1) \bmod (2^{31} - 1)$$

## 2.2 Implémentation en R

Afin d'implémenter une version de notre fonction en le langage de programmation R, on doit s'éloigner un peu du formalisme et s'orienter dans l'applications. C'est-à-dire qu'on va pas garder une valeur  $x_0$  pour toute l'éternité; on aura une variable globale déterminant l'état du générateur qui serait mis à jour quand on veut générer une suite des valeurs.

```

1  # Initialiser la graine (une variable globale) a 0
2  g_SEED_SOUSMARIN <- 0
3
4
5  # similaire a la fonction de R set.seed, mettre a jour
6  # la valeur de g_SEED_SOUSMARIN
7  set_seed <- function(seed) {
8    assign("g_SEED_SOUSMARIN", seed, envir = .GlobalEnv)
9  }
10
11 # La fonction LCG pure qu'on a defini en parti 3
12 f_sousmarin <- function(x) {
13   (12345678 * x + 1) %% (2^31 - 1)
14 }
15
16 # Generer une suite des variables de taille n en mettant a jour l'etat
17 # de la graine a chaque pas.
18 gen_suite <- function(n) {
19
20   suite <- vector("numeric", n) # allouer un vector de taille n
21
22   for (i in seq_len(n)) {
23     x_i <- f_sousmarin(g_SEED_SOUSMARIN)
24     suite[[i]] <- x_i
25     set_seed(x_i)
26   }
27
28   suite
29 }

```

On a donc implémenté notre propre LCG. Pour renvoyer une valeur dans l'intervall  $]0, 1[$  afin de simuler  $X \sim \mathcal{U}(0,1)$ , on remarque que la division modulo  $m$  renvoi une valeur entre  $]0, m[$ . Pour le normaliser, on divise par la facteur  $m$ .

```

1  r_std_unif <- function(n) {
2
3    suite <- vector("numeric", n)
4
5    for (i in seq_len(n)) {
6      x_i <- f_sousmarin(g_SEED_SOUSMARIN)
7      suite[[i]] <- x_i / (2^31 - 1)
8      set_seed(x_i)
9    }
10
11    suite
12  }
13

```

On imite le comportement et la signature de la fonction dans R de base runif avec

```

1  r_unif <- function(n, min = 0, max = 1) {
2
3    spread <- max - min
4    suite <- vector("numeric", n)
5
6    for (i in seq_len(n)) {
7      x_i <- f_sousmarin(g_SEED_SOUSMARIN)
8      suite[[i]] <- (x_i * spread) / (2^31 - 1) + min
9      set_seed(x_i)
10    }
11
12  }

```

```
13     suite  
14 }
```

## 2.3 Tests statistiques d'une loi uniforme

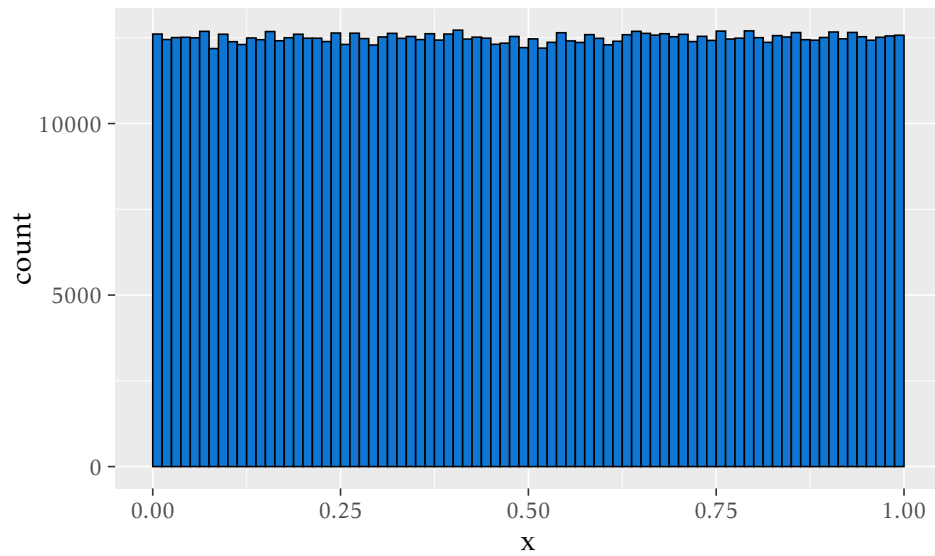


Figure 1: Histogram généré à partir d'un appel à `r_unif` avec  $n = 100000$

## 2.4 Diehard