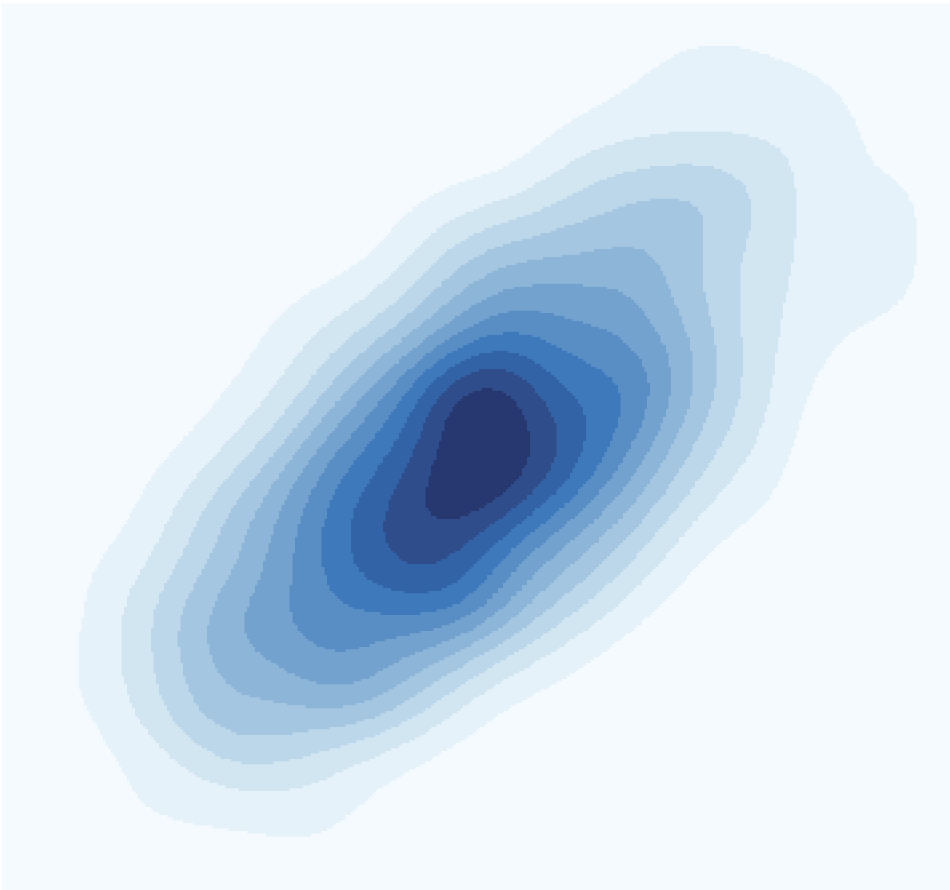
Cables Sous-Marins

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 electrons sont encodés par les fonctions booléennes. Les *fonctions* - dans le sens mathémathique - sont des objets purement déterministes. Autrement dit, une fonction associe à une donné d'entrée une valeur unique apartenant à l'espace d'arrivé. Si on lui donne plusiers 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 éloquemment que celle-ci offert par John von Neumann, l'un des meilleurs mathémathiciens, pionniers, informaticiens de tous les temps; générer des variables aléatoires sur un ordinateur n'est tout simplement pas possible. Cependant, cela ne nous empechera pas d'essayer quand même[1].

2 Loi uniforme

On commence notre projet en étudiant la loi la plus simples 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 transormée inverse; la méthode de rejet) permettant de simuler des lois differentes comme la loi normale, la loi exponentielle, etc. Il s'agira principalement d'échantilloner 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 $f : \mathbb{R} \longrightarrow \mathbb{R}$ 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 trouver une fonction qui vérifie certaines qualités désirées. Par exemple, on veut que notre fonction ait une période suffisamment long et qu'elle produise des valeurs uniformément réparti sur un interval. Etudions la fonction $x \mapsto (x+1)\,\%\,2$, où % est l'opération de modulus et on fixe une graine $x_0 = 1$.

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

Cette fonction produit alors la suite

$$\{1, 0, 1, 0, 1, 0, 1, 0, \ldots\}$$

dont la période est 2 et évidamment dont les valeurs ne sont aussie 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 expectations. Heuresement pour nous, il y existe pleins de fonctions qui satisfaitent nos buts flous.

2.1 Générateur congruentiel linéaire

La méthode la plus directe à implementer 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 précedente est une 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'objective

[1] Il s'agira de produire des variables dites pseudo-aléatoires

de notre projet c'est d'approfondir la connaissance autour des méthodes pour générer les variables (pseudo) aléatoires, on a décidé d'implementer 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.

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

2.2 Implémentation en R

Afin d'implémenter une version de notre fonction en le language de programmation R, on doit s'éloigner un peut du formalisme et s'orienter dans l'applications. C'est-à-dire qu'on va pas garder une valuer 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.

```r

# Initialiser la graine (une variable globale) a 0
g_SEED_SOUSMARIN <- 0

# similaire a la fonction de R set.seed, mettre a jour
# la valeur de g_SEED_SOUSMARIN
set_seed <- function(seed) {
    assign("g_SEED_SOUSMARIN", seed, envir = .GlobalEnv)
}

# La fonction LCG pure qu'on a defini en parti 3
f_sousmarin <- function(x) {
    (12345678 * x + 1) %% (2^31 - 1)
}

# Generer une suite des variables de taille n en mettant a jour l'etat
# de la graine a chaque pas.
gen_suite <- function(n) {

    suite <- vector("numeric", n) # allouer un vector de taille n

    for (i in seq_len(n)) {
        x_i <- f_sousmarin(g_SEED_SOUSMARIN)
        suite[[i]] <- x_i
        set_seed(x_i)
    }

    suite
}
```

On a donc implémenté notre propre LCG, `f_sousmarin`. Pour renvoyer une valeur dans l'interval $]0, 1[$ afin de simuler $X \sim \mathcal{U}(0,1)$, on remarque que la division modulo m renvoi une valeur entre $]0, m-1[$. Pour le normaliser, on divise par la facteur $m-1 \equiv 2^{31}-2$.

```r

r_std_unif <- function(n) {

    suite <- vector("numeric", n)

    for (i in seq_len(n)) {
        x_i <- f_sousmarin(g_SEED_SOUSMARIN)
        suite[[i]] <- x_i / (2^31 - 2)
        set_seed(x_i)
    }

    suite
}
```

Si on considère le cas générale où l'on souhaiterait générer des variables uniformément répartis dans l'interval $]a, b[$, on commence tout d'abord en échantillonant $X \sim \mathcal{U}(0,1)$. Ensuite, on multiplie X par l'ecárt entre a et b, $(b-a)$, pour produire une variable $X_{b-a} \in]0, b-a[$. Finalement, on décale X_{b-a} en additionant a pour finir avec le variable aléatoire uniformément réparti dans l'interval $]0+a, b-a+a[\equiv]a, b[$. Après cette transformation affine appliqué à X, on finit avec $X_{a,b} \sim \mathcal{U}(a,b)$.

On imite le comportement et la signature de la fonction dans R de base `runif` avec l'implémentation suivante

```r

r_unif <- function(n, min = 0, max = 1) {

    spread <- max - min
    suite  <- vector("numeric", n)

    for (i in seq_len(n)) {
        x_i <- f_sousmarin(g_SEED_SOUSMARIN)
        suite[[i]] <- (x_i * spread) / (2^31 - 2) + min
        set_seed(x_i)
    }

    suite
}
```

2.3 Tests statistiques d'une loi uniforme

Alors comment peut-on vérifier que notre LCG produisent des valeurs qui sont vraimenet répartis uniformément? Quand il s'agit de produire des milliards d'observations, on peut pas facilement vérifier à la main si notre fonction n'a pas de structure évidente (sauf la période qui est mathématiquement inévitable). Pourtant, on peut commencer avec une exploration visuelle.

Pour ce faire, on utilise la fonction `r_std_unif` définie au-dessus pour générer 1E6 valeurs aléatoires qui sont supposés d'être uniformément répartis sur l'interval $]0, 1[$.

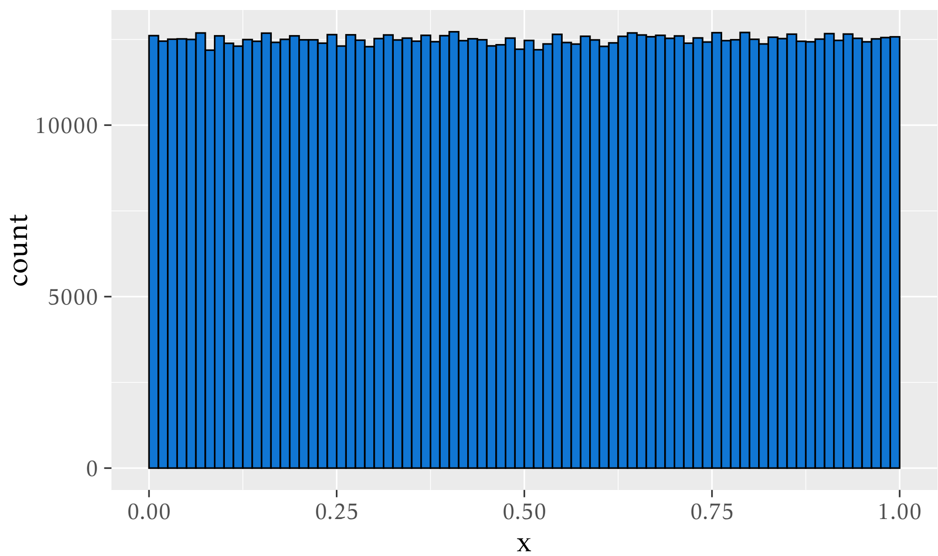


Figure 1: Histogram généré à partir d'un appel à `r_std_unif` avec $n = 1000000$. On initiliase la graine de notre générateur avec un appel à `set_seed(0)`.

On peut aussi facilement constater que les statistiques de notre échantillon correspondent bien a celles-ci que l'on attend.

```r
library(sousmarin)

set_seed(0)
x <- r_std_unif(1E6)

mu  <- mean(x)  # 0.5000658
sig <- std(x)   # 0.2877586
```

On réitère le simple fait qu'il n'y à **rien d'aléatoire** avec la génération de ces valeurs et que vous pouvez vérifier leurs quantité en téléchargant notre package **ICI**[2].

On passe à l'analyse. Avec notre échantillon x de taille 1E6, le moyen de l'échantillon $\bar{x} = 0.50006584$ et l'écart type de l'échantillon est $s = 0.2877586$. Comme la moyenne d'une loi uniforme est $\frac{b-a}{2} = \frac{1-0}{2} = 0.5$, on est très satisfaits que $\bar{x} = 0.50006584 \sim 0.5$. Parallèlement, l'écart type d'une loi uniforme est $\frac{b-a}{\sqrt{12}} = \frac{1-0}{\sqrt{12}} = 0.2886751$, ce qui n'est pas du tout loins de notre $s = 0.2877586$.

2.4 Diehard

My boy George Marsaglia

2.5 Vitesse

Les boucles en R sont **LENTS**. Genre horriblement lents. Pour la suite, on va implementer les fonctions en C en utilisant le package Rcpp. On va peut-etre explorer des autres optimisations au niveau de parallelisation, qui portera avec eux leurs propres problemes telles que les "data races" - quand deux "threads" essaye d'acceder et modifier l'etat du générateur en même temps.

3 Projections

Méthode de transormation inversé, méthode de rejet, Transforme Box-Muller.

[2] On mettra ici un lien du github (voire un lien de CRAN????) et des intstruction pour télécharger notre package