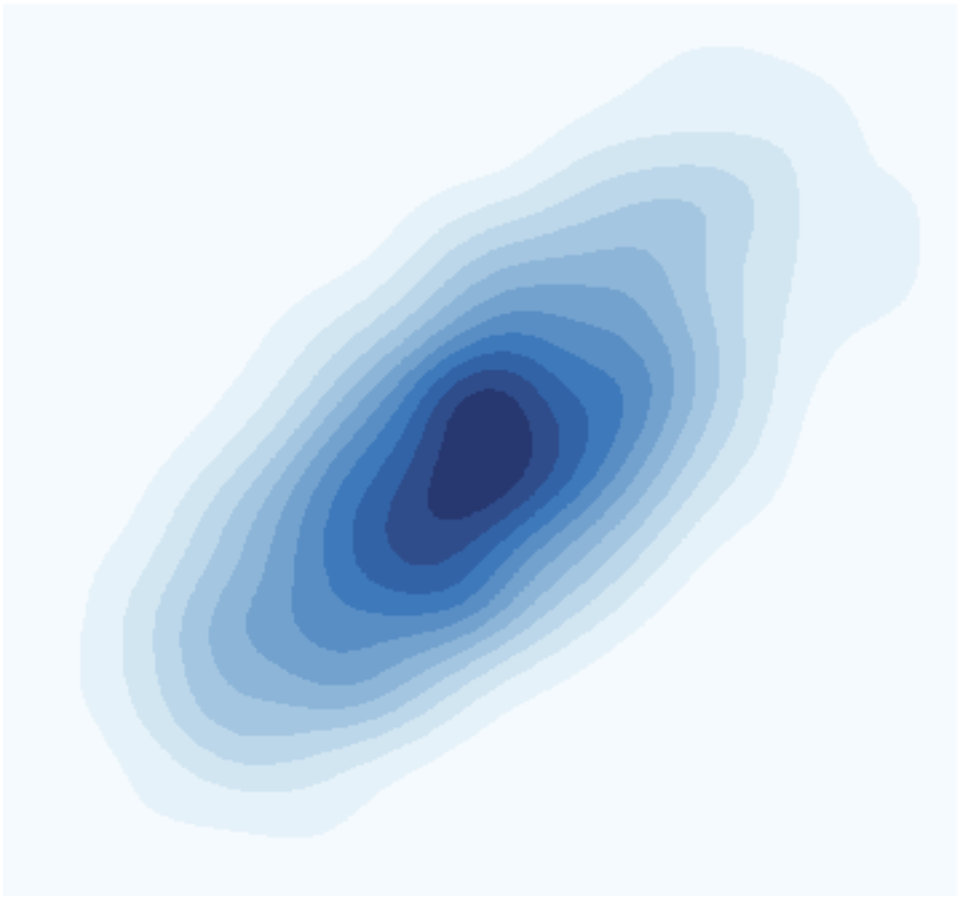
Cables Sous-Marins

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 de variables aléatoires

Un ordinateur n'est qu'un gros agencement complexe des circuits. Regnées par les lois physiques, les opérations provenant du mouvement des électrons sont encodées 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 unique valeur dans l'espace d'arrivée. Si on lui donne plusieurs fois la même valeur, par exemple x_1 et x_2 telles 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 alors que nous disposons seulement de méthodes déterministes?

Nous ne pouvons pas à répondre à cette question plus éloquemment que le fait John von Neumann, l'un des meilleurs mathémathiciens, pionniers, informaticiens de tous les temps; générer des variables aléatoires sur un ordinateur est tout simplement impossible. Cependant, cela ne nous empêchera pas d'essayer quand même[1].

2 Loi uniforme

On commence notre projet en étudiant la loi la plus simple parmi les lois usuelles - la loi uniforme. Tout d'abord, parce qu'elle est simple, mais la loi uniforme va également nous permettre construire des algorithmes plus complexes, notamment la méthode de la transormée inverse ou la méthode de rejet. Ainsi, cela nous permettra de simuler différentes lois, comme la loi normale, la loi exponentielle, etc. Il s'agira principalement d'échantilloner une variable $X \sim \mathcal{U}(0,1)$ puis ensuite d'effectuer des manipulations mathématiques pour produire une variable suivant une autre loi ciblée.

Alors sans plus tarder, on formalise nos objectifs. 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 longue et qu'elle produise des valeurs uniformément réparties sur un intervalle. Etudions la fonction $x \mapsto (x+1) \,\%\, 2$, où % est l'opération de modulos 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

$$\{1, 0, 1, 0, 1, 0, 1, 0, ...\}$$

dont la période est 2 et évidemment dont les valeurs ne sont pas aussi variées qu'on le souhaite. Nous verrons plus précisement ce que "suffisamment variés" veut dire. Pour l'instant, on se contente de dire que cette suite-là n'atteint pas nos attentes. Heureusement pour nous, il existe de nombreuses fonctions qui remplissent nos critères recherchés, nous les détaillerons plus tard.

2.1 Générateur congruentiel linéaire

La méthode la plus directe à implémenter pour générer une telle suite est un *générateur congruentiel linéaire*. Congruentiel parce qu'il s'agit d'une opération modulo et linéaire vu qu'il y a 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 sont $a = 1$, $c = 1$, et $m = 2$. Ne vous inquiétez pas, il existe un large panel de générateurs congruentiels linéaires (LCG). Les choix des paramètres utilisés par les logiciels connus sont détaillés sur une page Wikipédia et leurs propriétés sont déjà bien étudiées. Vu que l'objectif de

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

notre projet est d'approfondir la connaissance autour des méthodes pour générer des 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 choissisant 12345678. Finalement, on affecte à 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 language de programmation R, on doit s'éloigner un peu de la pureté de la théorie et se salir les mains dans le code ! C'est-à-dire que l'on ne 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.

```

    # 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 definie en partie 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 vecteur 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'intervalle $]0, 1[$ afin de simuler $X \sim \mathcal{U}(0,1)$, on remarque que la division modulo m renvoie une valeur entre $]0, m-1[$. Pour le normaliser, on divise par le facteur $m - 1 \equiv 2^{31} - 2$.

```

    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 général où l'on souhaiterait générer des variables uniformément réparties dans l'intervalle $]a, b[$, on commence tout d'abord par échantillonner $X \sim \mathcal{U}(0,1)$. Ensuite, on multiplie X par l'écart entre a et b, $(b-a)$, pour produire une variable $X_{b-a} \in]0, b-a[$. Finalement, on décale X_{b-a} en

additionnant a pour finir avec la variable aléatoire uniformément répartie dans l'intervalle $]0+a, b-a+a[\equiv]a, b[$. Après cette transformation affine appliquée à X, nous avons $X_{a,b} \sim \mathcal{U}(a, b)$.

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

```
r_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

Comment vérifier que notre LCG produit des valeurs qui sont véritablement réparties uniformément ? Quand il s'agit de produire des milliards d'observations, on ne 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 supposées être uniformément réparties sur l'intervalle $]0, 1[$.

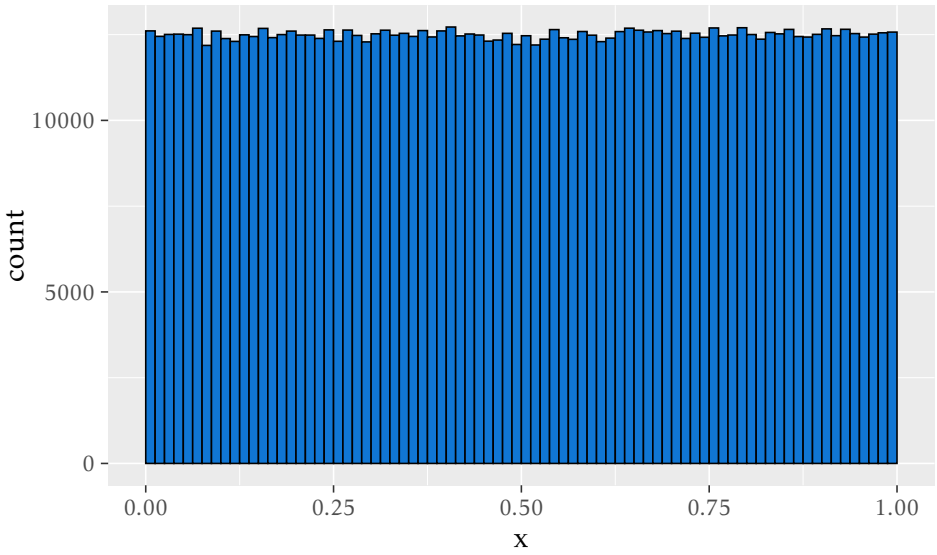


Figure 1: Histogramme généré à partir d'un appel à `r_std_unif` avec $n = 1000000$. On initialise la graine de notre générateur avec un appel à `set_seed(0)`.

On peut aussi facilement vérifier que les statistiques de notre échantillon correspondent bien à celles que l'on attend.

```
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 a **rien d'aléatoire** avec la génération de ces valeurs et que vous pouvez vérifier leur 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 $\frac{b-a}{2} = \frac{1-0}{2} = 0.5$, nous sommes ravis de voir 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 est proche de notre $s = 0.2877586$.

2.4 Diehard

My boy George Marsaglia

2.5 Vitesse

Les boucles en R sont **LENTES**. Genre horriblement lentes. Pour la suite, on va implémenter les fonctions en C en utilisant le package Rcpp. Pour suivre, nous tentons d'explorer d'autres optimisations au niveau de la parallélisation. Ces améliorations nous donneront d'autres problèmes à résoudre tels que les conditions de courses - quand deux "threads" essaient d'accéder et de modifier l'état du générateur en même temps.

3 Projections

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

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