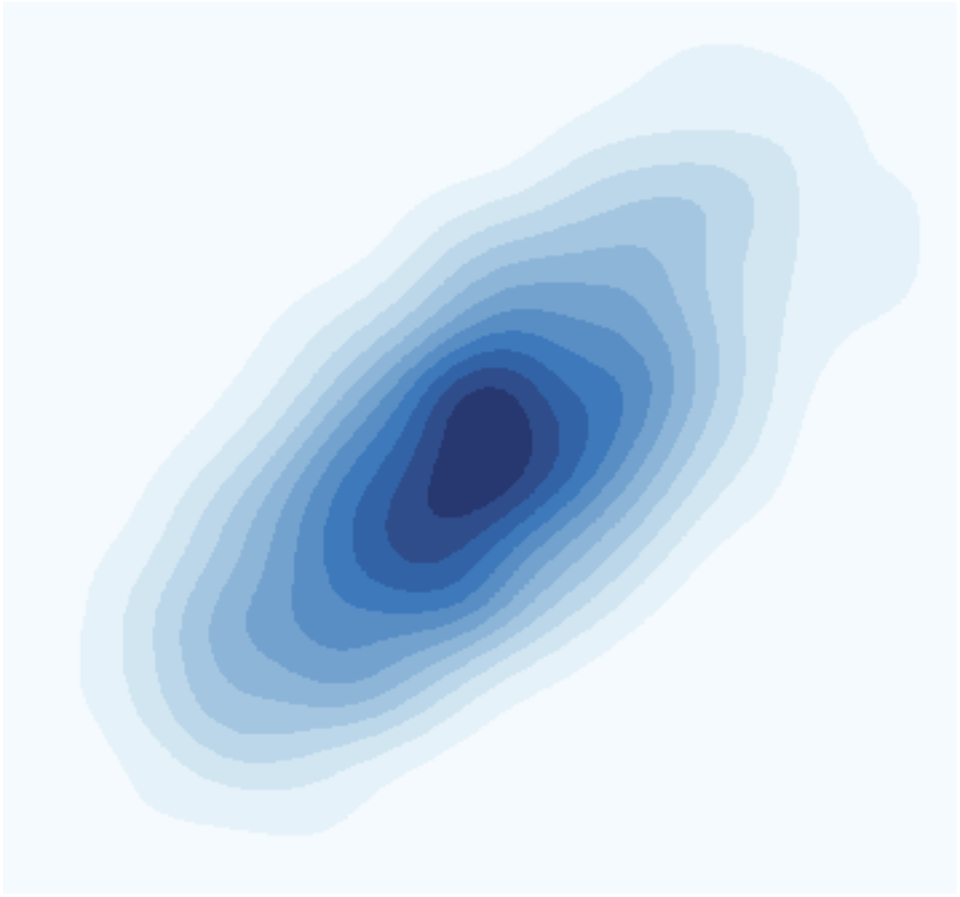
Simulation conditionnelle d'un processus gaussien

NEEL Pauline, PETROS Russom Samson, RAKOTOVAO Jonathan, VOYLES Evan

10 Mai 2022



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

John von Neumann

1 Introduction

L'objectif principal de ce projet est d'étudier et d'implémenter une procédure permettant de générer des simulations conditionnelles de processus gaussiens à l'aide de la méthode spectrale et du krigeage. Nous procéderons pour cela pas à pas.

La première étape sera d'abord d'appréhender ce qu'est un algorithme de simulation et de comprendre comment cela fonctionne. Nous commencerons donc ce projet par une question simple : comment générer des réalisations d'une variable aléatoire normale (cadre univarié). Pour y répondre, nous allons s'intéresser à la méthode d'acceptation-rejet ainsi qu'à la méthode de Box-Muller.

On se placera ensuite dans un cadre multivarié : nous utiliserons la décomposition de Cholesky de la matrice de variance-covariance afin de générer des réalisations d'un vecteur gaussien. Enfin, dans un cadre spatial, nous simulerons des processus gaussiens grâce à la méthode spectrale, puis conditionnerons les simulations obtenues à l'aide du krigeage. L'implémentation des différents algorithmes se fera sous R.

De plus, en utilisant Rshiny nous coderons une application web qui permettra à des utilisateurs de générer puis de visualiser des réalisations de variables aléatoires, vecteurs ou processus gaussiens.

2 Simulation classique

2.1 Génération de variables aléatoires

Un ordinateur n'est qu'un gros agencement complexe de circuits. Régnées 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 implique 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 mathématiciens, 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. Il s'agira de produire des variables dites pseudo-aléatoires.

2.2 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 de construire des algorithmes plus complexes, notamment la méthode de la transformée 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'échantillonner 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 $f : \mathbb{R} \longrightarrow \mathbb{R}$ telles 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. En effet, nous savons qu'elle sera périodique, il est donc importante que sa période soit très grande pour pas qu'un schéma soit visible.

On souhaite également 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, ce sont des fonctions pseudo-aléatoires, elles sont déterminées mais leurs comportements s'approchent de l'aléatoire.

2.2.1 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 notre projet est d'approfondir la connaissance autour des méthodes générant des variables (pseudo) aléatoires, on a décidé d'implémenter notre propre LCG hybride. Pour m, on choisit $2^{31}-1$, un nombre de Mersenne qui est très connu. Pour a, on s'amuse en choisissant 12345678. Finalement, on affecte à l'incrémenteur c la valeur 1.

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

2.2.2 Implémentation en R

Dans ce projet, nous faisons le choix d'implémenter notre propre version de runif afin de comprendre en profondeur la notion d'aléatoire. En effet, toutes les autres lois que nous allons pouvoir simuler, seront constuites à partir de runif. Il était donc important pour nous de faire cette étape. 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 mise à jour quand on veut générer une suite de 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 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)
        }

```

```
28          suite
29      }
```

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 $m - 1 \equiv 2^{31} - 2$.

```
1
2       r_std_unif <- function(n) {
3
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 / (2^31 - 2)
9               set_seed(x_i)
10          }
11
12          suite
13      }
```

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

```
1
2       r_unif <- function(n, min = 0, max = 1) {
3
4           spread <- max - min
5           suite  <- vector("numeric", n)
6
7           for (i in seq_len(n)) {
8               x_i <- f_sousmarin(g_SEED_SOUSMARIN)
9               suite[[i]] <- (x_i * spread) / (2^31 - 2) + min
10              set_seed(x_i)
11          }
12
13          suite
14      }
```

2.2.3 Vérification pour la 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 par 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 $]0,1[$.

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

```
1       library(sousmarin)
2
3       set_seed(0)
4       x <- r_std_unif(1E6)
5
6       mu  <- mean(x) # 0.5000658
7       sig <- std(x)  # 0.2877586
```

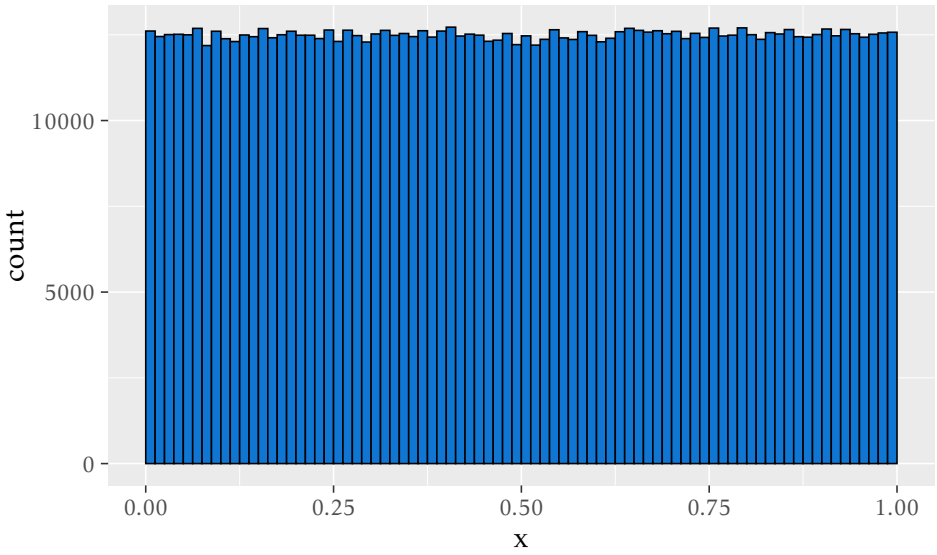


Figure 1: Histogramme généré à partir d'un appel à `r_std_unif` avec $n = 1000000$. On initialise la graine de notre générateur avec un appel à `set_seed(0)`.

On 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 téléchargeant notre package **ICI**[1].

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 $s = 0.2877586$.

2.2.4 Diehard

2.2.5 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[2]. 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.

2.3 Méthode de transformation inversée

La méthode de transformation inversée consiste à échantillonner une variable aléatoire $X \sim \mathcal{U}(0,1)$ et à utiliser l'expression analytique de la fonction de répartition d'une loi cible. Comme la fonction de répartition $F(x)$ est une fonction croissante définie sur $\mathbb{R}^n$ à valeurs dans $[0,1]$, si on peut trouver une expression fermée de son inverse $F^{-1} : [0,1] \longrightarrow \mathbb{R}$, on applique la méthode de transformation inversée afin de réaliser des simulations.

Pour éclairer la méthode, on va étudier la fonction de répartition de la loi exponentielle. Pour rappel, une variable aléatoire suivant une loi exponentielle de paramètre λ a pour fonction de densité $f(x) = \lambda e^{-\lambda x}$ pour $x \geq 0$, 0 sinon. On a choisi cette loi parce qu'elle est munie d'une fonction de répartition facilement calculable et surtout dont l'inverse a une expression analytique. Calculons sa fonction de

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

[2] Il se peut que l'on aura pas assez de temps pour realiser ca

répartition:

$$\begin{aligned} F(x) &= \int_{-\infty}^{x} f(t)dt \\ &= \int_{-\infty}^{0} 0dt + \int_{0}^{x} \lambda e^{-\lambda t} dt \\ &= 0 + \left[-e^{-\lambda t}\right]_0^x \\ &= 1 - e^{-\lambda x} \end{aligned}$$

Calculons maintenant son inverse, F^{-1} :

$$\begin{aligned} y &= 1 - e^{-\lambda(F^{-1}(y))} \\ e^{\lambda(F^{-1}(y))} &= 1 - y \\ -\lambda F^{-1}(y) &= \ln(1-y) \\ F^{-1}(y) &= \frac{-\ln(1-y)}{\lambda} \end{aligned}$$

La loi exponentielle est l'une des quelques lois où l'on peut facilement trouver l'inverse de la fonction de répartition. Cela nous permet d'échantilloner une variable aléatoire $X \sim \exp(\lambda)$ efficacement à partir d'une seule réalisation d'une loi uniforme. Il s'agit tout simplement de tirer $U \sim \mathcal{U}(0,1)$ et ensuite évaluer $X = F^{-1}(U)$.

L'implémentation en R ne prend qu'une seule ligne:

```r
rexp_inv <- function(n, lambda = 1) {
    # Generate n realizations of a uniform random variable n times
    (-1 / lambda) * log(runif(n, 0, 1))
}
```

2.4 Methode d'acceptation-rejet

Comment faire lorsque l'on veut simuler une loi dont on ne peut pas calculer l'inverse de la fonction de répartition ? Une solution est d'utiliser la méthode d'acceptation-rejet. Prenons un exemple simple pour mieux comprendre. Notons f la fonction densité de loi que l'on souhaite simuler : $f(x) = 6x(1-x)$ sur $[0, 1]$.

Le principe est simple : on va borner f par une fonction g que l'on sait simuler. Ici, on prendra la fonction constante $g(x) = 1.5$. On simule des points uniformément répartis dans le rectangle noir. Supposons que l'on souhaite simuler 10 simulations, on tire alors 10 réalisations de $X \sim \mathcal{U}(0,1)$: on aura les abscisses des points. Ensuite, pour chaque abscisse, on tire $Y \sim \mathcal{U}(0,1.5)$ ce qui représentera l'ordonnée de notre point. On aura alors tiré des points uniformément répartis dans le rectangle. Finalement, on garde seulement ceux se trouvant en dessous de la fonction $f(x)$. Ainsi, l'ensemble de ces points seront bien distribués selon la loi $f(x)$

3 Simulation géostatistique

3.1 Motivation

Le monde de simulation classique opère sous l'hypothèse d'indépendance des réalisations successives.

3.2 R shiny

Pour faire la simulation sans passer par du code R, nous avons créé une interface web afin de récupérer les variables simulées. cette interface est implémentée avec la bibliothèque Rshiny, cette bibliothèque est simple à utiliser et permet de créer des sitewebs avec des applications(fonctions) qui s'exécutent en arrière-plan pour faire la simulation.

Nous n'avons pas encore terminé de coder l'application mais dans la version finale l'utilisateur aura le choix de télécharger les variables Aléatoires simulés.

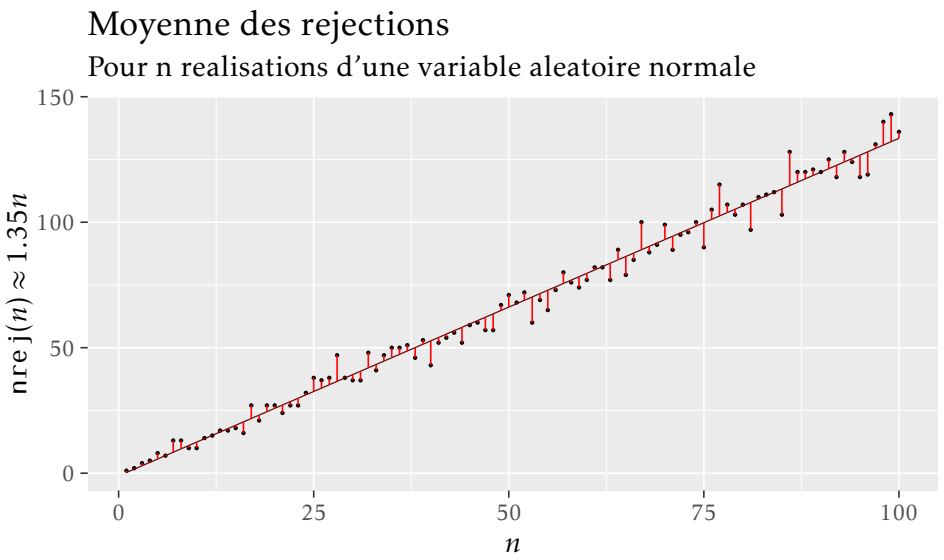


Figure 2

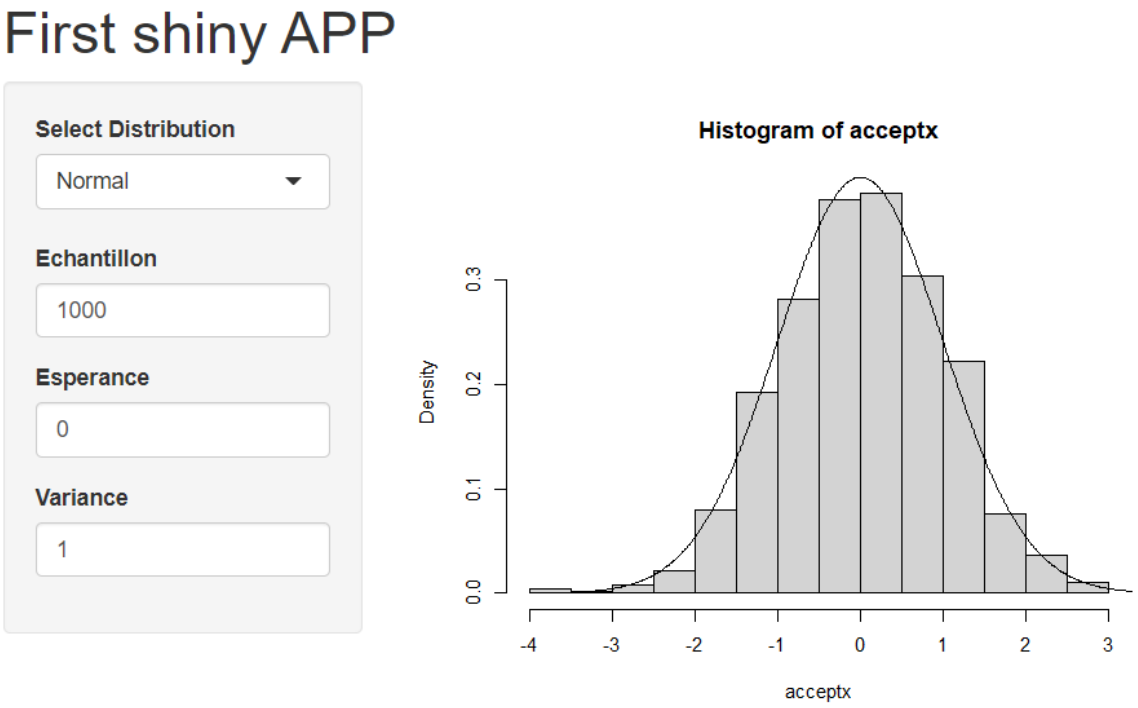


Figure 3: app shiny pour faire la simulation

3.3 Réflexions

Ce projet d'initiation a été une véritable découverte pour nous, puisque on a appris à utiliser de nouveaux langages de programmation, R et Rshiny. De plus, on a découvert la simulation de variables aléatoires multivariées qui sont utiles lorsque l'on veut simuler certains phénomènes physiques. met plustot ce que je viens d'envoyer, merci encore

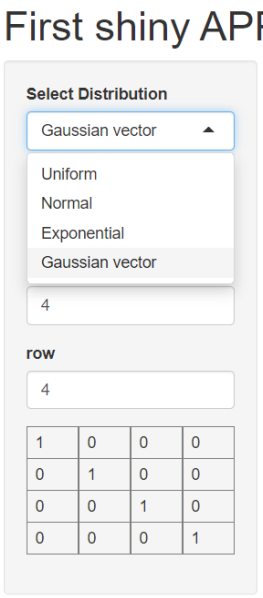


Figure 4: choix de distributions