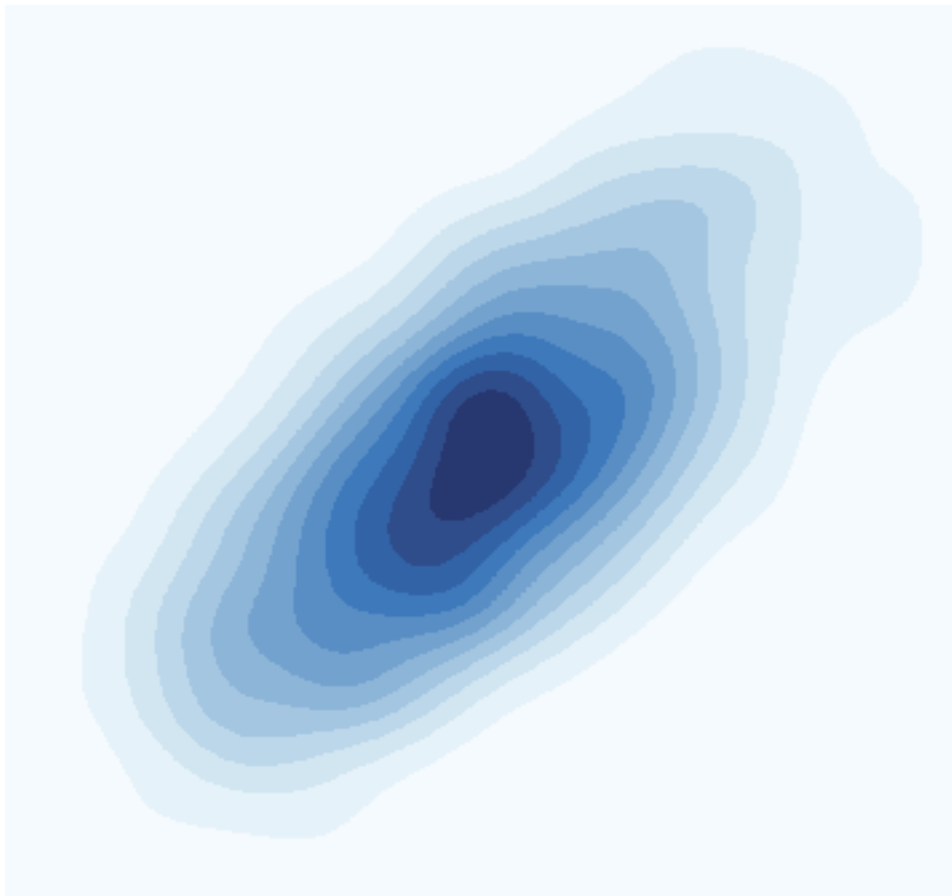


# Simulation conditionnelle d'un processus gaussien

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

19 Mai 2022



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

---

*John von Neumann*

## Table des matières

1	Introduction	2
2	Simulation classique	2
2.1	Génération de variables aléatoires	2
2.2	Loi uniforme	2
2.2.1	Générateur congruentiel linéaire	3
2.2.2	Implémentation en R	3
2.2.3	Vérification pour la loi uniforme	4
2.2.4	Simulation de différentes lois	5
2.3	Méthode de transformation inversée	5
2.3.1	Démonstration	5
2.4	Méthode d'acceptation-rejet	7
2.4.1	Démonstration	7
2.4.2	Simulation	8
2.5	Loi normale	10
2.6	Box-Muller	11
2.7	Vecteur Gaussien	12
2.7.1	Simulation à l'aide de la méthode de factorisation de cholesky	13
3	Simulation géostatistique	13
3.1	Motivation	13
3.2	Cadre d'étude	13
3.2.1	Stationnarité	14
3.3	Simulation spatiale	14
3.3.1	Méthodologie	15
3.4	Krigeage	18
4	Programmation	18
4.1	R shiny	18
4.2	sousmarin	18
5	Conclusion	18
6	Conclusion	19
7	Ce que nous pourrions améliorer	19
7.1	Vitesse	19
8	Réflexions	20
	Références	21

# 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 nous 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 éloquentement 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  $f : \mathbb{R} \rightarrow \mathbb{R}$  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. Étudions la fonction  $x \mapsto (x + 1) \% 2$ , où  $\%$  est l'opération de modulus et on fixe une graine  $x_0 = 1$ .

$$f(x_0) = (1 + 1) \% 2 = 2 \% 2 = 0$$

$$f(x_1) = (0 + 1) \% 2 = 1 \% 2 = 1$$

$$f(x_2) = (1 + 1) \% 2 = 0$$

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 pas aussi variées qu'on le souhaite. Nous verrons plus précisément 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.

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

### 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 constituées à partir de `runif`. Il était donc important pour nous de faire cette étape. Afin d'implémenter une version de notre fonction en langage 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.

---

```
1
2  # Initialiser la graine (une variable globale) a 0
3  g_SEED_SOUSMARIN <- 0
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 definie en partie 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 vecteur 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, `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  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 - 2)
8          set_seed(x_i)
9      }
10
11      suite
12
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  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 - 2) + min
9          set_seed(x_i)
10     }
11
12     suite
13
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 inévitée). 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

```

---

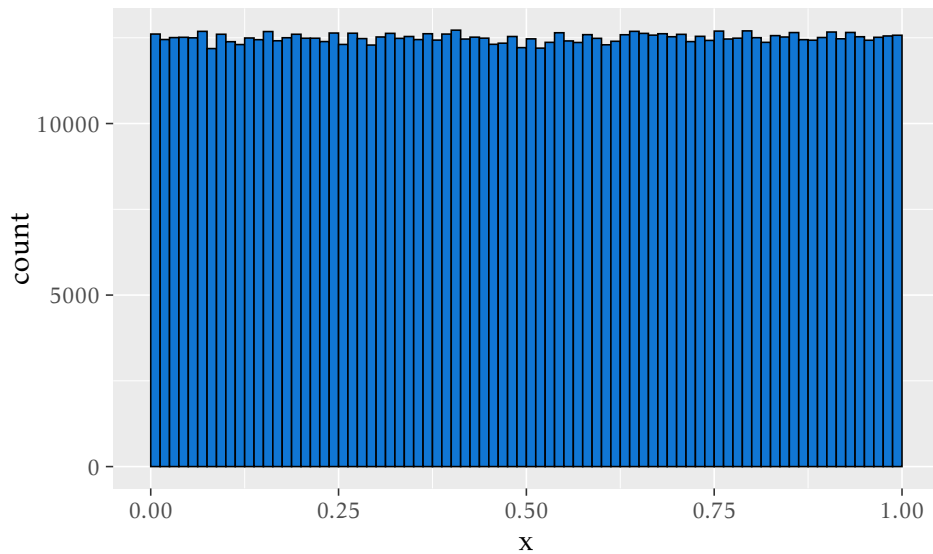


FIGURE 1 – Histogramme généré à partir d'un appel à `r_std_unif` avec  $n = 1000000$ . On initialise la graine de notre générateur avec un appel à `set_seed(0)`.

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**<sup>1</sup>.

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 Simulation de différentes lois

Maintenant que nous avons compris la notion de "aléatoire" et de "pseudo-aléatoire", et que nous avons saisi comment fonctionne la fonction `runif` de R, nous allons pouvoir générer des variables aléatoires plus complexes. Dans la prochaine partie, nous allons implémenter la méthode de transformation inversée, pour simuler une loi exponentielle. Ensuite nous verrons la méthode d'acceptation-rejet pour simuler une loi normale, et finalement la méthode de Box-Muller. Tous ces algorithmes sont contruits à partir de `runif`, fonction de R qui distribue une variable aléatoire uniforme.

### 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  $F^{-1} : [0, 1] \rightarrow \mathbb{R}$ , on applique la méthode de transformation inversée afin de réaliser des simulations.

#### 2.3.1 Démonstration

On pose :

$$u = F(x) \Leftrightarrow x = F^{-1}(u) \quad (1)$$

Par définition, on a  $F(x) = \mathbb{P}(X \leq F^{-1}(u))$ . Il vient alors :  $F(F^{-1}(u)) = \mathbb{P}(X \leq F^{-1}(u))$

Or, par définition de la réciproque :  $F(F^{-1}(u)) = u$  et comme  $F$  est strictement croissante :  $\mathbb{P}(X \leq F^{-1}(u))$ . On en déduit que :  $u = \mathbb{P}(F(X) \leq u)$  et on reconnaît la fonction de répartition de la loi uniforme.

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

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  $\lambda$  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 répartition :

$$\begin{aligned} F(x) &= \int_{-\infty}^x f(t) dt \\ &= \int_{-\infty}^0 0 dt + \int_0^x \lambda e^{-\lambda t} dt \\ &= 0 + \left[ -e^{-\lambda t} \right]_0^x \\ &= 1 - e^{-\lambda x} \end{aligned}$$

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'échantillonner 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 :

---

```
1  rexp_inv <- function(n, lambda = 1) {  
2      # Generate n realizations of a uniform random variable n times  
3      (-1 / lambda) * log(runif(n, 0, 1))  
4  }
```

---

On peut facilement comparer la moyenne empirique avec la moyenne théorique. En effet, on sait que théoriquement, l'espérance de la loi exponentielle de paramètre  $\lambda$  est  $\frac{1}{\lambda}$ . Ainsi, on effectue la moyenne de  $n = 1e4$  réalisations pour plusieurs  $\lambda$ , et on compare ces résultats avec la théorie.

On obtient alors le graphique suivant qui montre que les moyennes empiriques calculées pour chaque lambda correspondent aux valeurs théoriques de l'espérance.

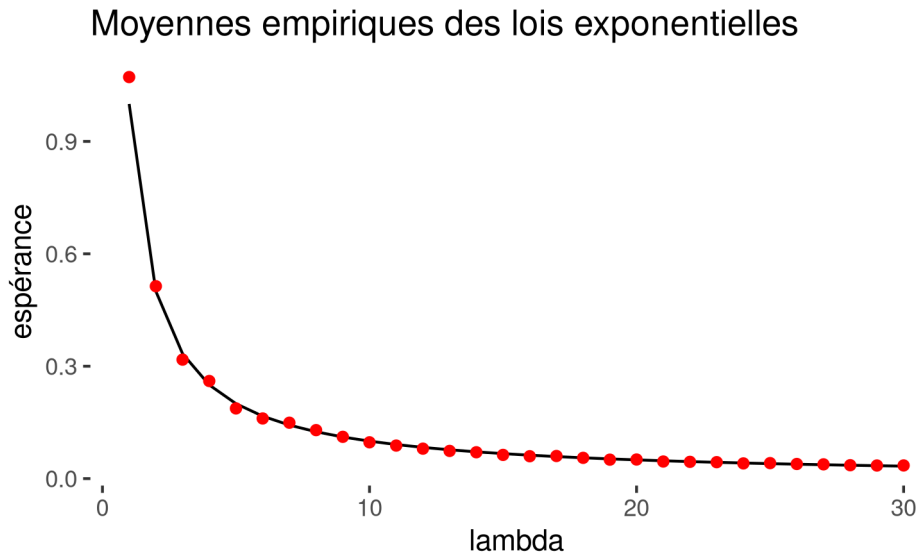


FIGURE 2 – Comparaison moyennes empiriques avec espérance de plusieurs lois exponentielles de paramètre  $\lambda$ . En rouge les moyennes empiriques. En noir les valeurs théoriques de l'espérance,  $1/\lambda$ .

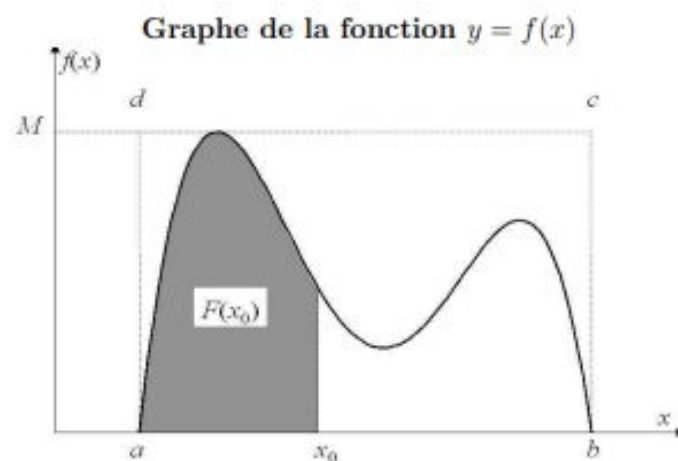
## 2.4 Méthode d'acceptation-rejet

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.

### 2.4.1 Démonstration

Soit  $X$  une variable aléatoire de densité  $f$  et de fonction de répartition  $F$ . Supposons  $f$  nulle en dehors d'un intervalle  $[a, b]$ . Soit  $M$  un majorant de  $f(x) : \forall x \in [a, b] : f(x) \leq M$

On considère le graphe de la fonction  $y = f(x)$  :



On a :

$$\mathbb{P}(X \leq X_0) = \mathbb{P}(X \leq X_0 | A = (X, Y)) = \mathbb{P}(X \leq X_0 | Y \leq f(x)) \quad (2)$$

ce qui donne, d'après la formule des probabilités conditionnelles :

$$= \frac{\mathbb{P}(X \leq X_0, Y \leq f(x))}{\mathbb{P}(Y \leq f(x))} \quad (3)$$



Or,

$$\mathbb{P}(X \leq X_0, Y \leq f(x)) = \frac{\text{surface de la zone hachure}}{\text{surface du rectangle (abcd)}} = \frac{F(X_0)}{M(b-a)} \quad (4)$$

et

$$\mathbb{P}(X \leq X_0, Y \leq f(x)) = \frac{\text{surface sous la courbe}}{\text{surface du rectangle (abcd)}} = \frac{1}{M(b-a)} \quad (5)$$

On en déduit que :

$$\mathbb{P}(X \leq X_0) = \frac{F(X_0)}{M(b-a)} \times \frac{M(b-a)}{1} = F(X_0) \quad (6)$$

La variable  $X$  ainsi obtenue a bien une densité  $f$  et une fonction de répartition  $F$ .

#### 2.4.2 Simulation

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 le compact  $[0, 1]$ .

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 sous la courbe de  $g(x)$ . Supposons que l'on souhaite générer 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 l'abscisse de ceux se trouvant en dessous de la fonction  $f(x)$ . Ainsi, l'ensemble de ces valeurs, les abscisses des points acceptés donc, seront bien distribuées selon la loi  $f(x)$ .

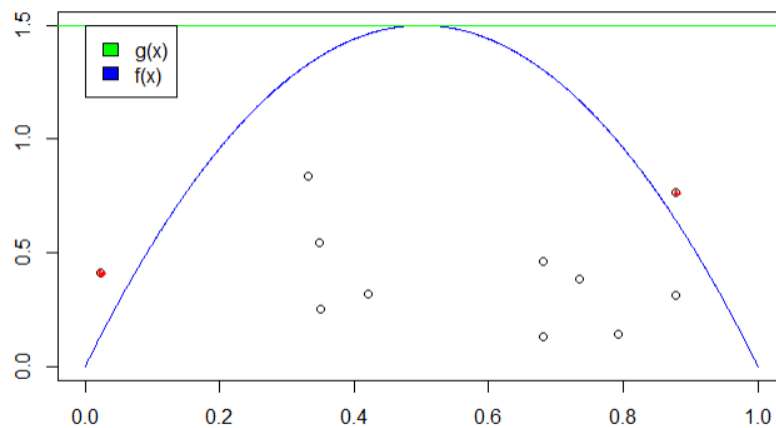


FIGURE 3 – app shiny pour faire la simulation

Sur ce graphique, on voit que les points en rouge se situant au-dessus de  $f(x)$  seront rejetés tandis que les points blancs se situant en-dessous de la courbe seront acceptés.

On peut facilement implémenter l'algorithme d'acceptation-rejet pour une loi quadratique suivant la densité  $f(x) = 6x(1-x)$  sur  $[0, 1]$  sous R :

```
1 rquad <- function(n) {
2   ind <- function(x) { 0 <= x & x <= 1 } # x \in [0, 1]
3   # Uniform density between (0, 1)
4   f <- function(x) {
5     6 * x * (1 - x) * ind(x)
6   }
7 }
```

```

8
9   g <- function(x) {
10     1 * ind(x)
11   }
12
13   c <- 1.5
14   out <- rep(0, n)
15
16   for (i in seq_len(n)) {
17     reject <- TRUE
18     z <- 0
19
20     while (reject) {
21       y <- runif(1)
22       u <- runif(1)
23
24       reject <- u > (f(y) / (c * g(y)))
25       z <- y
26     }
27
28     out[[i]] <- z
29   }
30
31   out
32 }
33
34

```

On simule alors  $n = 10,000$  simulations avec `rquad(1e5)` pour vérifier si notre algorithme produit bien une variable aléatoire dont la densité  $f(x) = 6x(1 - x)$  :

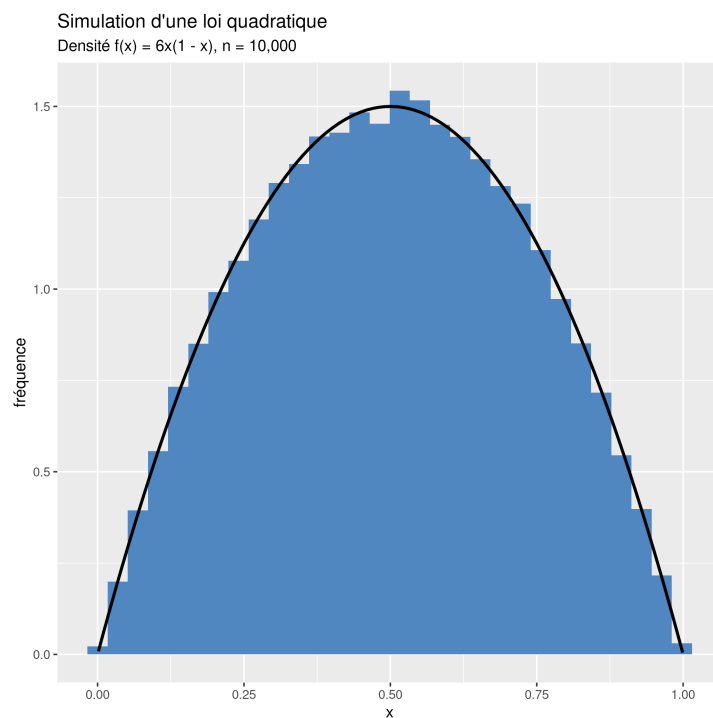


FIGURE 4 – 10,000 réalisations d’une variable aléatoire dont la densité est une fonction quadratique. On a vérifié graphiquement que la méthode d’acceptation-rejet implémenté en `rquad` permet de simuler une variable aléatoire avec  $f(x) = 6x(1 - x)$ .

## 2.5 Loi normale

On peut aussi appliquer la méthode d'acceptation-rejet pour simuler la loi normale. Pour rappel, la loi normale a la fonction de densité suivante :

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{1}{2} \left( \frac{x-\mu}{\sigma} \right)^2$$

Pour ce faire, on reproduit la même méthode expliquée ci-dessus. La différence est qu'avec la loi normale, nous avons un support non compact,  $\mathbb{R}$ .

Alors, nous majorons la fonction densité  $f_X(x)$  en rouge de la loi normale par

$$g(x) = \begin{cases} -\lambda \exp -\lambda x & \text{si } x \leq 0 \\ \lambda \exp -\lambda x & \text{sinon.} \end{cases}$$

la densité de la loi "double exponentielle" en vert qui correspond à une variable exponentielle  $\exp(1)$  affectée d'un signe positif ou négatif, tiré avec probabilité égale : 0.5.

Grâce à la méthode d'inversion, nous savons simuler des réalisations selon la loi de  $g(x)$ . Ces réalisations sont bien distribuées sur  $\mathbb{R}$  entier. Pour avoir une réalisation qui suit une loi normale, on commence donc par tirer  $X$  qui suit la loi de  $g(x)$ . Ensuite, on tire son ordonnée  $Y$  qui suit la loi  $\mathcal{U}(0, g(X))$ . Puis on pose une condition, si  $Y > f(X)$  alors le point est rejeté, ie on ne le prend pas en compte. En revanche, si  $Y \leq f(X)$  alors le point est accepté. On garde uniquement son abscisse  $X$ , qui suit alors la loi de  $f_X(x)$ .

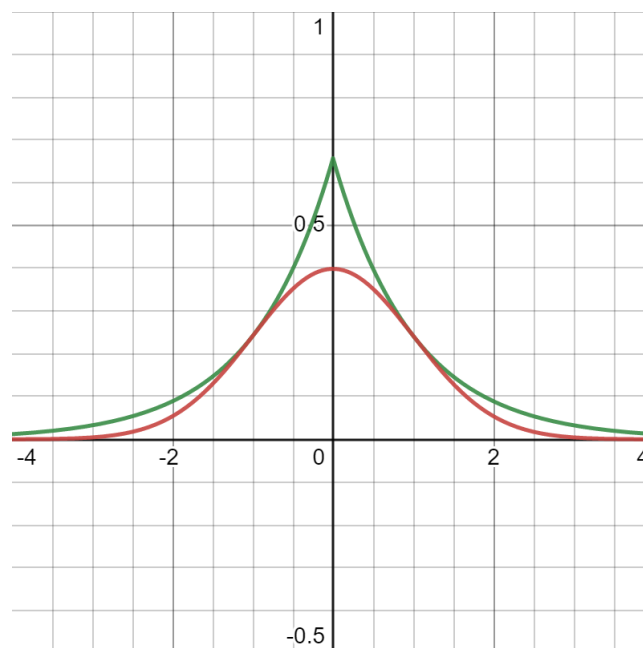


FIGURE 5 – Fonctions densités

On remarque que la fonction majorante en verte est extrêmement serrée autour de la fonction gaussienne que l'on souhaite à simuler. Plus la différence entre  $f_X(x)$  et  $g(x)$  est importante, plus le nombre d'itérations qu'il faut pour accepter une valeur est grand. Cela est évident quand on considère que la probabilité d'accepter une valeur proposée est  $\frac{f_X(x)}{cg(x)}$ . Quand  $f_X(x)$  est près de  $cg(x)$ , la probabilité d'être accepté est presque 1.

On peut implémenter l'algorithme d'acceptation-rejet pour une loi normale centrée réduite sous R :

```

1  rnorm_acc_std <- function(n) {
2
3      x <- vector("numeric", n)
4
5      for (i in 1:n) {
6
7          reject <- TRUE
8      }
9  }
```

```

9      z <- 0
10
11     while (reject) {
12
13       y1 <- rexp(1)
14       y2 <- rexp(1)
15
16       reject <- y2 < ((y1 - 1) * (y1 - 1) / 2)
17
18       z <- abs(y1)
19     }
20
21     if (runif(1) < 0.5) { z <- -z }
22     x[[i]] <- z
23   }
24
25   x
26
27 }

```

Par ailleurs, on peut quantifier le nombre d'itérations qu'il faut pour simuler  $n$  variables aléatoirement suivant une loi normale centrée réduite.

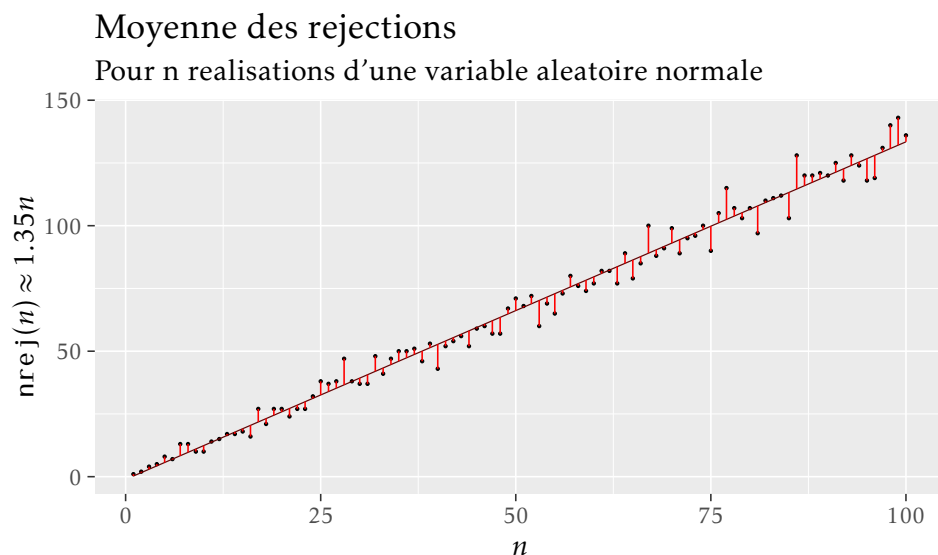


FIGURE 6 – Moyenne des rejets

La pente de cette courbe représente le pourcentage de variables simulées totales sur le nombre de variable acceptées. Ce pourcentage vaut environ 1,35 c'est-à-dire que pour simuler 100 variables, il faudra en moyenne en tirer 135 en tout. Ce chiffre est spécifique au coefficient que nous avons choisi pour la fonction majorante  $g(x)$

## 2.6 Box-Muller

La loi normale n'a pas une densité à support compact et on ne connaît pas d'expression simple de l'inverse de sa fonction de répartition. On ne peut donc, théoriquement, employer la méthode de transformation inversée. On présente ici une méthode qui permet de simuler un couple des variables aléatoires normales, centrées, réduites et indépendantes. On veut simuler  $X \sim \mathcal{N}(0,1)$  et  $Y \sim \mathcal{N}(0,1)$  indépendantes. On connaît la densité jointe de  $X$  et  $Y$  :

$$f_{X,Y}(x,y) = \frac{1}{2\pi} \exp\left(-\frac{x^2 + y^2}{2}\right)$$

On effectue le passage en coordonnées polaires

$$x = \rho \cos(\theta), y = \rho \sin(\theta)$$

et on obtient

$$f_{X,Y}(x,y)dxdy = \frac{1}{2\pi} \exp\left(-\frac{\rho^2}{2}\right) \rho d\rho d\theta = f_{R,\theta}(\rho, \theta) d\rho d\theta$$

Dans la densité jointe des variable  $R$  et  $\theta$ , on reconnaît  $\frac{1}{2\pi}$  = densité de  $\theta$  qui suit une loi Uniforme sur 0 et  $2\pi$ .

$\rho \exp(-\frac{\rho^2}{2})$  = densité de  $R$ .

On en déduit la fonction de répartition de  $R$  :

$$F_R(\rho) = 1 - \exp\left(-\frac{\rho^2}{2}\right)$$

On reconnaît une loi exponentielle de paramètres  $1/2$  pour  $R^2$ .

On a donc les lois de  $R$  et  $\theta$  :

$$R^2 \sim \exp\left(-\frac{1}{2}\right), \theta \sim U([0, 2\pi])$$

La méthode consiste donc à tirer deux variables uniforme  $U_1$  et  $U_2$ , puis on insère ces deux variables dans les formule suivante :

$$R = \sqrt{-2 \ln U_1}$$

$$\theta = 2\pi U_2$$

et on pose :

$$X = R \cos \theta, Y = R \sin \theta$$

Ces deux variables aléatoires sont indépendantes par construction leur densité jointe est définie comme le produit de leurs densités respectives .

Remarque : Pour simuler  $Z \sim N(\mu, \sigma^2)$ , on simule  $X \sim N(0, 1)$  et on effectue la transformation :

$$Z = \mu + \sigma X$$

## 2.7 Vecteur Gaussien

Maintenant que nous savons simuler des variables aléatoires gaussiennes, nous allons nous intéresser à la simulation d'un vecteur aléatoire gaussien  $X = (X_1, \dots, X_d) \sim \mathcal{N}(\mu, \sigma)$ ,  $d \in \mathbb{N}^*$ , avec

$$\mu = E[X] = (E[X_1], \dots, E[X_d])^T \in \mathbb{R}^d$$

et

$$\Sigma = E[XX^T] - E[X](E[X])^T$$

$\Sigma$  est appelée la matrice de variance (ou matrice de variance-covariance) de  $X$ . C'est une matrice de taille  $d \times d$  qui est semi-définie positive.

### 2.7.1 Simulation à l'aide de la méthode de factorisation de cholesky

De manière générale, un vecteur gaussien  $X$  se simule par transformation affine de variables aléatoires gaussiennes centrées réduites indépendantes  $Z$ .

Pour générer  $X$  nous allons suivre cet algorithme :

1. Générer  $Z_1, \dots, Z_n \sim \mathcal{N}(0, 1)$  et  $Z = (Z_1, \dots, Z_n)^T$ .
2. Dériver la décomposition Cholesky de  $\Sigma = \mathbf{L}\mathbf{L}^T$ .
3. Retourner  $X = \mu + \mathbf{L}Z$

l'algorithme est simple à implémenter, la première étape consiste juste à générer des variables aléatoires qui suivent une loi normale centrée réduite et qui ne sont pas corrélées. Ensuite nous faisons une décomposition de Cholesky de la matrice de covariance selon laquelle nous voulons faire la simulation de vecteurs gaussiens, la fonction de R `chol()` est utilisé pour la décomposition. Après la décomposition nous obtenons une matrice triangulaire inférieure  $\mathbf{L}$ . Finalement la transformation affine de l'étape 3 nous permet d'avoir un vecteur gaussien avec la bonne espérance et matrice de covariance.

Exemple : En deux dimension :

Preuve :

La matrice de covariance de tout vecteur aléatoire  $\mathbf{Y}$  est donnée par  $\mathbb{E}(\mathbf{Y}\mathbf{Y}^T)$ , où  $\mathbf{Y}$  est un vecteur colonne aléatoire de taille  $n \times 1$ . Prenez maintenant un vecteur aléatoire,  $\mathbf{X}$ , constitué de variables aléatoires non corrélées, chaque variable aléatoire,  $\mathbf{X}_i$ , ayant une moyenne nulle et une variance unitaire de 1. Puisque les  $\mathbf{X}_i$  sont des variables aléatoires non corrélées de moyenne nulle et de variance unitaire, nous avons  $\mathbb{E}(\mathbf{X}_i\mathbf{X}_j) = \delta_{ij}$ . Par conséquent,

$$\mathbb{E}(\mathbf{X}\mathbf{X}^T) = \text{Id}$$

Pour générer un vecteur aléatoire avec une matrice de covariance  $\Sigma$  donnée, on regarde la décomposition de Cholesky de  $\Sigma$ , c'est-à-dire  $\Sigma = \mathbf{L}\mathbf{L}^T$ . Notez qu'il est possible d'obtenir une décomposition de Cholesky de  $\Sigma$  puisque, par définition, la matrice de covariance  $\Sigma$  est symétrique et définie positive.

Considérons maintenant le vecteur aléatoire  $\mathbf{Z} = \mathbf{L}\mathbf{X}$ . Nous avons

$$\mathbb{E}(\mathbf{Z}\mathbf{Z}^T) = \mathbb{E}((\mathbf{L}\mathbf{X})(\mathbf{L}\mathbf{X})^T) = \mathbb{E}(\mathbf{L}\mathbf{X}\mathbf{X}^T\mathbf{L}^T) = \mathbf{L}\mathbb{E}(\mathbf{X}\mathbf{X}^T)\mathbf{L}^T = \mathbf{L}\text{Id}\mathbf{L}^T = \Sigma$$

Par conséquent, le vecteur aléatoire  $\mathbf{Z}$  possède la matrice de covariance souhaitée,  $\Sigma$ .

## 3 Simulation géostatistique

### 3.1 Motivation

Le monde de simulation classique opère sous l'hypothèse d'indépendance des réalisations successives. C'est-à-dire que le prochain valeur échantillonné n'a aucun rapport avec la valeur précédente. Si on considère qu'une région par exemple d'une montagne où le terrain dans un rectangle 2 dimensionnel a une certaine altitude, on peut imaginer que les *valeurs* de l'altitude sont aléatoires, mais dans un certain sens leurs quantités sont liées géographiquement. Il existe une structure spatiale sur la région.

La simulation géostatistique présente plusieurs nouveaux concepts inédits dans la simulation classique. Une idée intégrale de l'étude des processus stochastique spatiale c'est que la valeur observée est considérée une variable régionalisée, ce qui veut dire qu'une certaine terrain est une réalisations d'une fonction aléatoire. Comme l'angle d'étude est fondamentalement différent que la simulation classique, on doit approcher la simulation différemment.

### 3.2 Cadre d'étude

Dans cette section nous allons vous introduire quelques nouveaux objets mathématiques qui sont souvent utilisés dans le domaine de géostatistique. Il s'agira principalement d'un **processus stochastique**  $Z$ , une **fonction de covariance**  $C(h)$ , et un **domaine**  $\mathcal{D}$  sur laquelle on peut avoir des observations.

**Définition 3.2.1 (Processus stochastique)** Soit  $(\Omega, \mathcal{A}, \mathbb{P})$  un espace probabilisé, et  $\mathcal{D}$  un espace quelconque. On appelle *processus stochastique* toute fonction  $Z$  :

$$Z : \mathcal{D} \times \Omega \rightarrow \mathbb{R}$$

On aimerait utiliser cette définition pour modéliser la configuration spatiale d'un câble sous-marin en tant qu'une réalisation d'un processus stochastique. Dans ce cas, on peut imaginer qu'à chaque endroit du câble il y a une profondeur associée. Par exemple, à 100m le câble pourrait avoir une profondeur de 3.7km. On peut donc considérer que la configuration spatiale d'un câble sous-marin est une fonction aléatoire qui, pour une éventualité  $\omega \in \Omega$  donnée, associe à chaque position  $s \in \mathcal{D} = \mathbb{R}$  une profondeur  $Z(s) \in \mathbb{R}$ . Un autre exemple d'un processus stochastique pourrait être la quantité de précipitation sur l'année dans une région fixe. Dans ce cas-là, on prendra  $\mathcal{D} = \mathbb{R}^2$ .

### 3.2.1 Stationnarité

Une idée qui est super important dans l'étude des fonctions aléatoires spatiales est l'idée de la stationnarité. Grosso modo, la stationnarité est le concept que pour un processus stochastique, il existe des propriétés qui sont invariants aux déplacements dans l'espace. Souvent classifiés sur le principe des *moments statistiques*, la stationnarité nous permet de simplifier nos modèles mathématiques des phénomènes réelles.

**Définition 3.2.2 (Stationnarité à l'ordre 1)** *Un processus stochastique est qualifié de **stationnaire à l'ordre 1** si et seulement si le premier moment existe et est invariant par translation  $h \in D$  :*

$$\mathbb{E}[Z(x)] = \mathbb{E}[Z(x+h)]$$

.

En effet, cela nous dit qu'un processus stochastique  $Z$  stationnaire à l'ordre 1 a un premier moment qui existe (traduction : si  $\mathbb{E}[Z(x)] < +\infty$ ) et que cette espérance est la même partout dans l'espace  $\mathcal{D}$ . Cette condition impose une certaine régularité sous le processus que l'on aimerait modéliser. Au-delà, on pourrait parler de la stationnarité à l'ordre 2 qui impose une condition sur le deuxième moment de  $Z$ , la covariance :

**Définition 3.2.3 (Stationnarité à l'ordre 2)** *Un processus stochastique est qualifié de **stationnaire à l'ordre 2** si et seulement si la deuxième moment existe et est invariant par translation  $h \in D$  :*

$$\text{Cov}[Z(x_1), Z(x_2)] = \text{Cov}[Z(x_1+h), Z(x_2+h)]$$

.

Une conséquence importante de cette hypothèse c'est que la covariance d'un processus stationnaire à l'ordre 2 ne dépend que du vecteur séparant les sites :

$$\text{Cov}[Z(x_1), Z(x_2)] = f(x_1 - x_2)$$

Cette fonction  $f(x_1 - x_2)$  est souvent écrite sous la forme  $C(h)$ , où  $h \in \mathcal{D}$  est le vecteur séparant les sites. Un modèle très souvent appliqué dans la géostatistique est la modèle exponentielle isotropique

$$C(h) = e^{-\frac{|h|}{a}}$$

où isotropique veut dire que  $C(h)$  dépend seulement sur la distance  $|h|$  et pas la direction, et  $a$  est un paramètre qui détermine comment la covariance diminue pour des sites lointaines.

Pour éviter de trop se perdre dans la théorie, nous nous lançons dans la simulation pour relier le monde des processus stochastique classique et spatiale.

## 3.3 Simulation spatiale

Dans ce premier exemple, on souhaite illustrer la simulation spatiale en unifiant le monde spatiale avec ce que nous avons déjà vu précédemment pour la simulation, spécifiquement avec le vecteur gaussien. Pour rappel, pour simuler une loi gaussienne multivariée, il faut l'espérance  $\vec{\mu}$  et la matrice de covariance  $\Sigma$ , ce qui est le vecteur analogue de  $\mu$  et  $\sigma^2$  qui décrit une loi normale en une dimension.

On voudrait simuler une réalisation de la variable régionalisée  $Z(s), s_i \in \mathcal{D}, \mathcal{D} \in \mathbb{R}^2$ . On va donc prendre une discrétisation de l'espace en une grille de  $n \times n$  où les coordonnées  $(i, j) \in \mathbb{R}^2$  de  $s_i \in \mathcal{D}, i \in$

$[[1, n^2]]$  est son indice dans une matrice carrée de  $n$  lignes et  $n$  colonnes remplie par les valeurs  $i \in [[1, n^2]]$  par **ligne dominante**. Pour  $n = 3$ , il s'agit de la matrice suivant :

$$\begin{pmatrix} s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 \\ s_6 & s_8 & s_9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 6 & 8 & 9 \end{pmatrix}$$

Les coordonnées  $\text{coord}(s)$  sont alors

$$\text{coord} \begin{pmatrix} s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 \\ s_6 & s_8 & s_9 \end{pmatrix} = \begin{pmatrix} (1, 1) & (1, 2) & (1, 3) \\ (2, 1) & (2, 2) & (2, 3) \\ (3, 1) & (3, 2) & (3, 3) \end{pmatrix}$$

et on prend une fonction de distance induite par la norme euclidienne :

$$d(s_i, s_j) = \|\text{coord}(s_i) - \text{coord}(s_j)\|_2$$

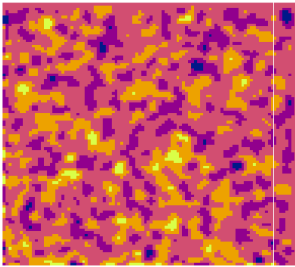
Par exemple,  $d(s_1, s_5) = \|(1, 1) - (2, 2)\|_2 = \sqrt{2}$ .

Maintenants, on est muni avec une notion de distance entre  $s_i, s_j \in \mathcal{D}$ . Il ne reste qu'à faire des hypothèse sur une fonction aléatoire sur  $\mathcal{D}$  si on voudrait simuler une réalisation  $\omega \in \Omega$  de  $Z$ . Pour commencer, on suppose que  $Z$  est un processus stationnaire à l'ordre 2. De plus, on suppose que la covariance de  $Z$  suit un modèle exponentielle

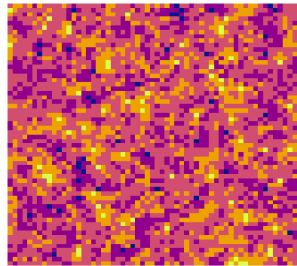
$$C(h) = e^{-\frac{|h|}{a}}$$

Comme on a une fonction de distance, on prend  $h = d(s_i, s_j)$ . C'est maintenant où on relie les des côtés de simulation : On suppose que la valeur de la variable régionalisé est un vecteur gaussien avec espérance  $\vec{\mu}$  et matrice de covariance  $\Sigma$ . Si on prend  $\vec{\mu} = \mathbf{0}$  il ne reste qu'à déterminer une matrice de covariance. C'est ici où on utilise la structure spatiale de notre grille et les distances entre eux pour computer une matrice de covariance et simuler un vecteur gaussien en utilisant la méthode de cholesky.

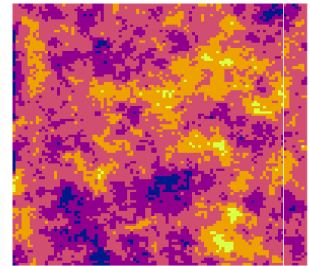
Le fait qu'on suppose que  $Z$  est stationnaire à l'ordre 2 (au premier moment aussi) nous permet d'avoir une covariance à partir d'une distance. On prend un moment pour vous illustrer quelques simulations de ce sort avant de détailler la méthodologie au niveau algorithmique.



(a)  $C(|h|) = m \exp(-\frac{|h|^2}{a^2})$



(b) Tent



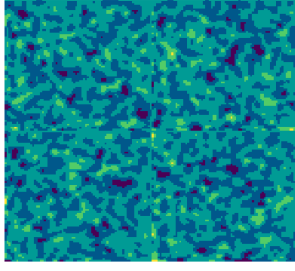
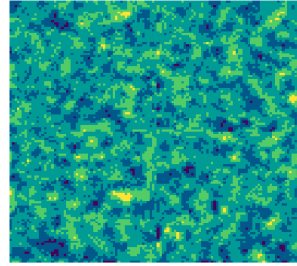
(c)

FIGURE 7 – Three simple graphs

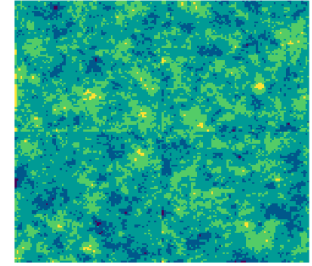
### 3.3.1 Méthodologie

Reprenons l'exemple d'une grille de 3 lignes et 3 colonnes. Pour simuler une réalisation de la fonction aléatoire  $Z$ , on procède en simulant un vecteur gaussien de 9 éléments, dont la matrice de covariance  $\Sigma$  est déterminé par l'hypothèse de stationnarité forte (à l'ordre 1 et 2) et calculé selon un modèle choisis. Nous utiliserons le modèle exponentielle pour commencer. Finalement, pour simplifier l'exemple on prend une espérance  $\mu = \mathbf{0} \in \mathbb{R}^9$ .




 (a)  $C(|h|) = m \exp(-\frac{|h|^2}{a^2})$ 


(b) Sphere



(c)

FIGURE 8 – Three simple graphs

Pour simuler un vecteur gaussien dans  $\mathbb{R}^9$ , on a besoin d'une matrice de covariance de 9 lignes et 9 colonnes :

$$\Sigma = \begin{pmatrix} \text{Cov}[s_1, s_1] & \dots & \text{Cov}[s_1, s_9] \\ \vdots & \ddots & \vdots \\ \text{Cov}[s_9, s_1] & \dots & \text{Cov}[s_9, s_9] \end{pmatrix}$$

Sous l'hypothèse de stationnarité, pour  $C(h)$  le modèle exponentiel et  $d(s_i, s_j)$  la norme euclidienne des coordonnées de  $s_i$  et  $s_j$ , on réécrit :

$$\Sigma = \begin{pmatrix} C(d(s_1, s_1)) & \dots & C(d(s_1, s_9)) \\ \vdots & \ddots & \vdots \\ C(d(s_9, s_1)) & \dots & C(d(s_9, s_9)) \end{pmatrix}$$

En calculant les distances, l'on a :

$$\Sigma = C(H), \quad H = \begin{pmatrix} 0 & 1 & 2 & 1 & \sqrt{2} & \sqrt{5} & 2 & \sqrt{5} & 2\sqrt{2} \\ & 0 & 1 & \sqrt{2} & 1 & \sqrt{2} & \sqrt{5} & 2 & \sqrt{5} \\ & & 0 & \sqrt{5} & \sqrt{2} & 1 & 2\sqrt{2} & \sqrt{5} & 2 \\ & & & 0 & 1 & 2 & 1 & \sqrt{2} & \sqrt{5} \\ & & & & 0 & 1 & \sqrt{2} & 1 & \sqrt{2} \\ & & & & & 0 & \sqrt{5} & \sqrt{2} & 1 \\ & & & & & & 0 & 1 & 2 \\ & & & & & & & 0 & 1 \\ & & & & & & & & 0 \end{pmatrix}$$

Comme la matrice  $\Sigma$  est symétrique, on remplit que la partie triangulaire à droite de  $H$  pour brevété. Illustrons l'un des calculs pour qu'on comprends bien ce que l'on vient de calculer. Pour chaque ligne, on souhaiterait calculer la covariance entre une site fixe et **toutes** les autres sites. Pour la première ligne, on a computed les distances entre  $s_1$  et  $s_i \in \{s_1, \dots, s_9\}$ . Par exemple,

$$H_{1,1} = d(s_1, s_1) = \|(1, 1) - (1, 1)\|_2 = 0$$

$$H_{1,2} = d(s_1, s_2) = \|(1, 1) - (1, 2)\|_2 = 1$$

$$H_{1,6} = d(s_1, s_6) = \|(1, 1) - (2, 3)\|_2 = \sqrt{5}.$$

Une dernière étape c'est d'appliquer le modèle de covariance exponentielle  $C(h; a) = \exp(-\frac{|h|}{a})$  pour tout élément  $h$  de  $H$  afin de trouver  $\Sigma$ , un comportement qui est implémenté par la fonction `get_cov_matrix_exp(n, a)` pour une grille de  $n$  lignes et  $n$  colonnes. Pour notre exemple, on appelle `get_cov_matrix_exp(3, 1)`.

$$\Sigma = C(H; 1) = \begin{pmatrix} 1.00 & 0.37 & 0.14 & 0.37 & 0.24 & 0.11 & 0.14 & 0.11 & 0.06 \\ 0.37 & 1.00 & 0.37 & 0.24 & 0.37 & 0.24 & 0.11 & 0.14 & 0.11 \\ 14 & 0.37 & 1.00 & 0.11 & 0.24 & 0.37 & 0.06 & 0.11 & 0.14 \\ 0.37 & 0.24 & 0.11 & 1.00 & 0.37 & 0.14 & 0.37 & 0.24 & 0.11 \\ 0.24 & 0.37 & 0.24 & 0.37 & 1.00 & 0.37 & 0.24 & 0.37 & 0.24 \\ 0.11 & 0.24 & 0.37 & 0.14 & 0.37 & 1.00 & 0.11 & 0.24 & 0.37 \\ 0.14 & 0.11 & 0.06 & 0.37 & 0.24 & 0.11 & 1.00 & 0.37 & 0.14 \\ 0.11 & 0.14 & 0.11 & 0.24 & 0.37 & 0.24 & 0.37 & 1.00 & 0.37 \\ 0.06 & 0.11 & 0.14 & 0.11 & 0.24 & 0.37 & 0.14 & 0.37 & 1.00 \end{pmatrix}$$

On pourrait appliquer des autres modèles de covariance pour changer la dépendance spatiale de notre fonction aléatoire :

Modèle de tente :

$$C(h; a, \sigma) = \begin{cases} \sigma^2(1 - \frac{|h|}{a}), & \text{si } 0 < |h| \leq a \\ 0 & \text{si } |h| > a \end{cases}$$

`get_cov_matrix_tent(3, 2, 1)`

$$\Sigma = C(H; 2, 1) = \begin{pmatrix} 1 & 0.50 & 0.00 & 0.50 & 0.29 & 0.00 & 0.00 & 0.00 & 0.00 \\ 0.50 & 1.00 & 0.50 & 0.29 & 0.50 & 0.29 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.50 & 1.00 & 0.00 & 0.29 & 0.50 & 0.00 & 0.00 & 0.00 \\ 0.50 & 0.29 & 0.00 & 1.00 & 0.50 & 0.00 & 0.50 & 0.29 & 0.00 \\ 0.29 & 0.50 & 0.29 & 0.50 & 1.00 & 0.50 & 0.29 & 0.50 & 0.29 \\ 0.00 & 0.29 & 0.50 & 0.00 & 0.50 & 1.00 & 0.00 & 0.29 & 0.50 \\ 0.00 & 0.00 & 0.00 & 0.50 & 0.29 & 0.00 & 1.00 & 0.50 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.29 & 0.50 & 0.29 & 0.50 & 1.00 & 0.50 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.29 & 0.50 & 0.00 & 0.50 & 1.00 \end{pmatrix}$$

Ainsi que le modèle sphérique

$$C(|h|; a, m) = \begin{cases} m(1 - (\frac{3}{2} \frac{|h|}{a} - \frac{1}{2} \frac{|h|^3}{a^3})), & \text{si } 0 < |h| \leq a \\ 0 & \text{si } |h| > a \end{cases}$$

et gaussien

$$C(|h|; a, m) = m \exp(-\frac{|h|^2}{a^2}), \quad a > 0$$

restent à notre disposition avec nos implémentations `get_cov_matrix_sphere` et `get_cov_matrix_gauss`.

Maintenant on appelle notre fonction `rmvnorm` pour simuler un vecteur gaussien dans  $\mathbb{R}^9$  avec espérance 0 et matrice de covariance  $\Sigma = C(H; a)$  suivant un modèle exponentiel avec  $a = 3$ .

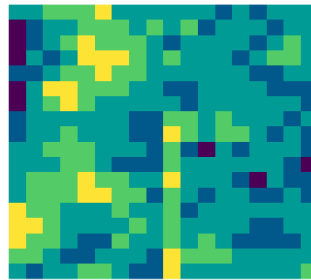


FIGURE 9 – Réalisation  $\omega \in \Omega$  d'un champ gaussien sur  $\mathcal{D} \in \mathbb{R}^2$  dans  $\mathbb{R}$  avec fonction de covariance exponentielle. Les valeurs  $Z(s)$  sont représenté en utilisant l'échelle de couleur viridis.

Pour élucider plus de structure, on fait augmenter la taille de notre grille. Cependant, comme la matrice de covariance est de taille  $n^2$  lignes et  $n^2$  colonnes pour une grille  $n$  fois  $n$ , le nombre d'opérations juste pour calculer  $\Sigma$  est de complexité  $\mathcal{O}(n^4)$ ! Et finalement la décomposition de Cholesky est de complexité  $\mathcal{O}(n^3)$  ainsi que la multiplication matricielle.

En résumé, cette méthode de simulation spatiale souffre de la malédiction des dimensions donc elle n'est pas traitable pour des grilles dont  $n$  dépasse 100. C'est ici où on pourrait parler des autres méthodes telle que la méthode spectrale pour simuler plus efficacement un champ gaussien de covariance exponentielle.

### 3.4 Krigeage

La dernière partie de notre projet s'oriente autour d'une méthode dite le krigeage, qui est effectivement une union de la simulation spatiale avec le problème numérique des moindres carrés.

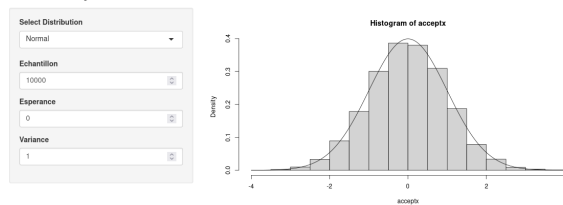
## 4 Programmation

### 4.1 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 sites web avec des applications (fonctions) qui s'exécutent en arrière-plan pour faire la simulation.

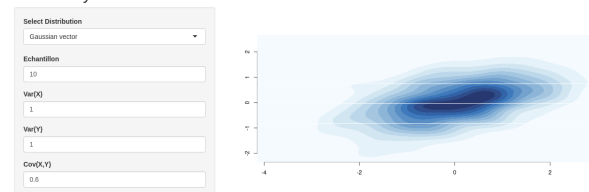
Nous n'avons pas encore terminé de coder l'application mais dans la version finale l'utilisateur aura le choix de télécharger les variables Aléatoires simulés.

First shiny APP



$$(a) C(|h|) = m \exp\left(-\frac{|h|^2}{a^2}\right)$$

First shiny APP



(b) Sphere

### 4.2 sousmarin

Le package R `sousmarin` est disponible sur notre répertoire github `ejovo13/sousmarin` avec les instructions pour comment télécharger la bibliothèque sous R avec `install_github` du package `devtools` écrit par le charmant Hadley Wickham.

Le package `sousmarin` contient les implémentations de toutes les méthodes que nous avons présentées dans ce rapport. La fonction d'acceptation-rejet pour une loi normale a été implémentée en C++ en cohésion avec le package `Rcpp` qui permet la compilation de code source en C++ avec une interface en R. Aussi les fonctions pour générer la matrice de covariance pour une simulation spatiale ont été réécrites en C++ comme les boucles sont très lentes en R. Avant la réécriture, la fonction `get_cov_mat_exp` mettait environ une minute pour simuler une grille de dimension 60 x 60. Après l'implémentation en C++ et nous avons réussi à effectuer une réalisation de 100 x 100 en moins de 10 secondes.

## 5 Conclusion

Ainsi, nous avons commencé par simuler des variables aléatoires simples en se basant uniquement sur `runif` (fonction uniforme de R). Nous savons maintenant utiliser les méthodes d'inversion et d'acceptation-rejet. Nous avons également poussé notre étude en simulation multivariée et en spatial.

Il y a plusieurs choses que nous aurions pu améliorer au cours du projet. Tout d'abord, les boucles en R sont plutôt lentes. Pour la suite, nous aurions pu implémenter les fonctions en C en utilisant le package `Rcpp`. Pour suivre, nous aurions tenté 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.

## 6 Conclusion

Ainsi, nous avons commencé par simuler des variables aléatoires simples en se basant uniquement sur `runif` (fonction uniforme de R). Nous savons maintenant utilisé les méthodes d'inversion et d'acceptation-rejet. Nous avons également poussé notre étude en simulation multivariée et en spatial. Pour cela nous avons utilisé les méthode de de krigeage et éventuellement nous allons voir si nous pouvons implémenter la méthode spectrale pour simuler des processus gaussiens

## 7 Ce que nous pourrions améliorer

### 7.1 Vitesse

Les boucles en R sont péniblement lentes comparé aux autre langages compilés comme le C. Donc, la plupart de nos algorithmes implémentés sont certaines des fois plus lentes que les fonctions de base en R telles que `runif`, `rexp`, etc. Si on avait plus de temps, on aurait aimé implémenter les fonctions en C en utilisant le package `Rcpp` qui permet d'écrire notre propre code en C/C++ à executer sous R.

## 8 Réflexions

Ce projet d'initiation a été une véritable découverte pour moi, puisque j'ai appris à utiliser de nouveaux langages de programmation, R et Rshiny. De plus, j'ai découvert la simulation de variables aléatoires multivariées qui sont utiles lorsque l'on veut simuler certains phénomènes physiques.

---

*Samson*

J'ai acquis de nombreuses compétences en informatique grâce à ce projet. J'ai appris à utiliser Github en projet, le logiciel R et Rshiny. J'ai compris les notions d'aléatoire et pseudo-aléatoire.

---

*Pauline*

J'ai approfondi mes connaissances autour la génération des variables aléatoires qui est indispensable dans le domaine de processus stochastique ou des nombreuses phénomènes physiques sont modélisé par des fonctions aléatoires. J'ai eu ma première rencontre avec la simulation conditionnelle et j'ai pu découvrir des nouveaux outils mathématiques utilisés en géostatistique tel que le variogram, la méthode spectrale, et le krigeage.

---

*Evan*

## Références

- [1] Yann Méneroux (2018), *Introduction à la Géostatistique*, Institut National de l'Information Géographique et Forestière
- [2] Donald E. Knuth (1986) *The T<sub>E</sub>X Book*, Addison-Wesley Professional.
- [3] Karl Sigman (2007), *Acceptance-Rejection Method*
- [4] Karl Sigman (2010), *Inverse Transform Method*
- [5] Eric Marcon (2021), *Krigeage avec R*
- [6] Dikr P. Kroese (2011), Thomas Taimre, and Zdravko I. Botev, *Handbook of Monte Carlo Methods*, Wiley series in Probability and Statistics
- [7] Luc Devroye (1986), *Non-Uniform Random Variate Generation*, Springer Science
- [8] Christian Lantuéjoul (2002), *Geostatistical Simulation*, Springer
- [9] Christian P. Robert, George Casella (1999), *Monte Carlo Statistical Methods Second Edition*, Springer Texts in Statistics
- [10] Jean-Paul Chilès, Pierre Delfiner (2012), *Geostatistics Modeling Spatial Uncertainty*, Wiley series in Probability and Statistics