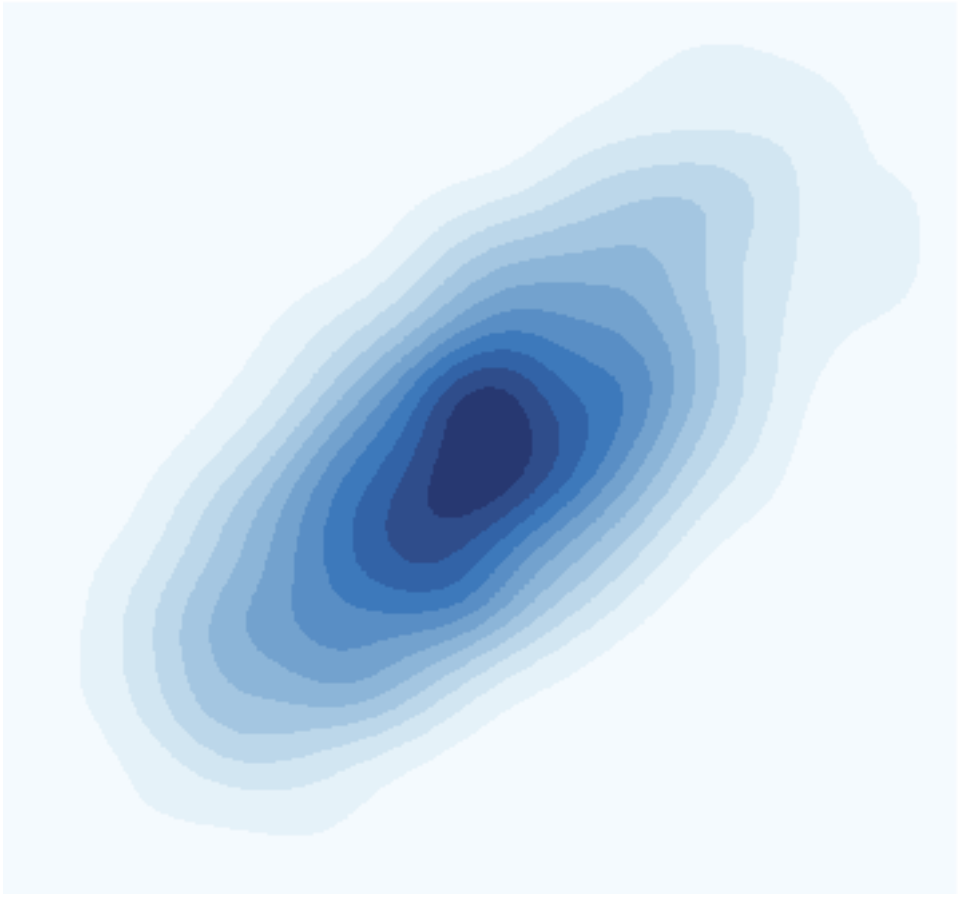
Simulation conditionnelle d'un processus gaussien

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

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)
        }

```

```
        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
    }
```

2.2.3 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.

```
    library(sousmarin)

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

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

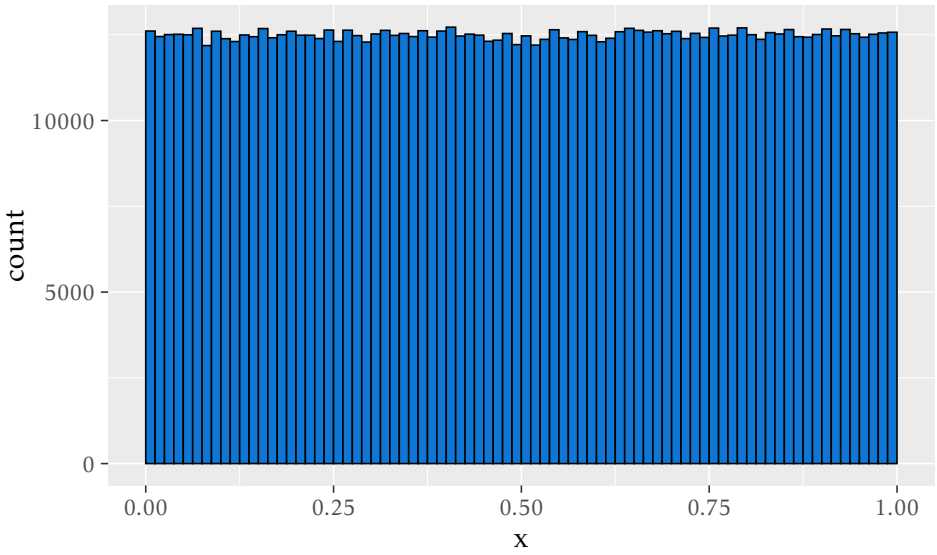


Figure 1 – Histogramme généré à partir d'un appel à `r_std_unif` avec $n = 1000000$. On initialise la graine de notre générateur avec un appel à `set_seed(0)`.

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 $s = 0.2877586$.

2.3 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.

2.4 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.

2.4.1 Démonstration

On pose :

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

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

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

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 foncion de répartition de la loi uniforme.

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 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))
}
```

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 λ est

$$g(x) = \lambda exp(-\lambda x)$$

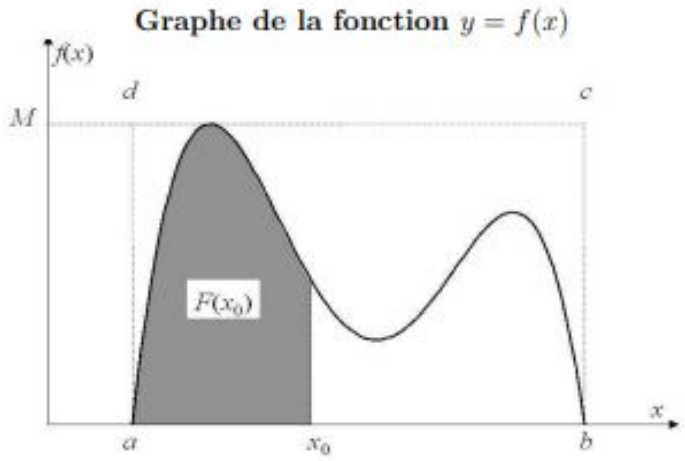
Ainsi, on effectue la moyenne d'un grand nombre de réalisations pour plusieurs λ, et on compare ces résultats avec la théorie.

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 – 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)`.

3 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.



3.1 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 :

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

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

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

Or,

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

et

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

On en déduit que :

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

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

3.1.1 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 $f(x)$.

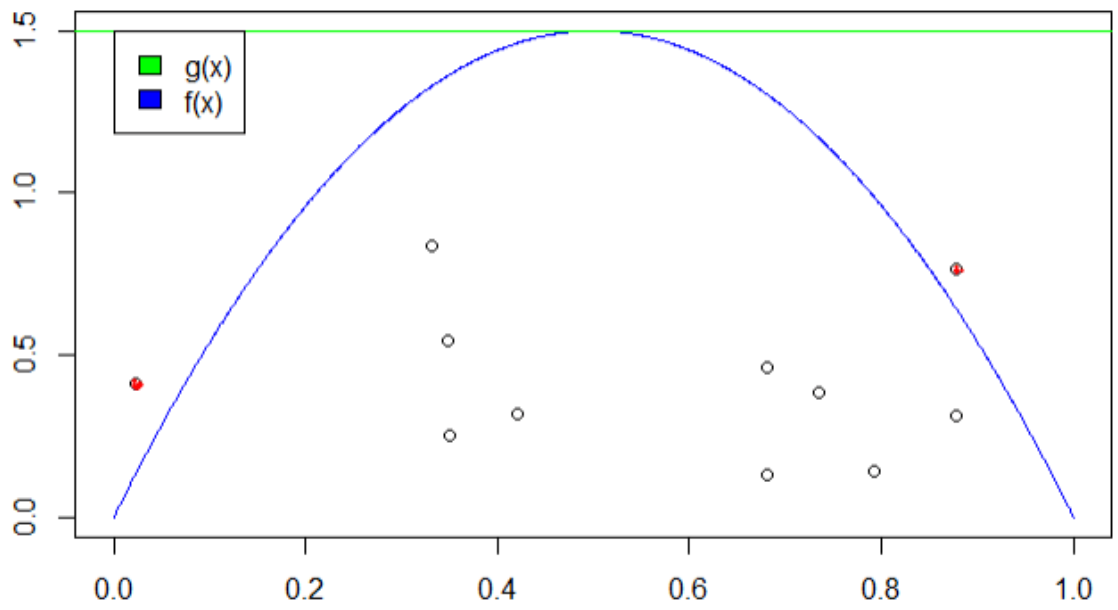


FIGURE 3 – app shiny pour faire la simulation

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 :

```
rquad <- function(n) {

    ind <- function(x) { 0 <= x & x <= 1 } # x \in [0, 1]
    # Uniform density between (0, 1)
    f <- function(x) {
        6 * x * (1 - x) * ind(x)
    }

    g <- function(x) {
        1 * ind(x)
    }

    c <- 1.5
    out <- rep(0, n)

    for (i in seq_len(n)) {

        reject <- TRUE
        z <- 0

        while (reject) {

            y <- runif(1)
            u <- runif(1)

            reject <- u > (f(y) / (c * g(y)))
            z <- y
        }

        out[[i]] <- z
    }

    out
}
```

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

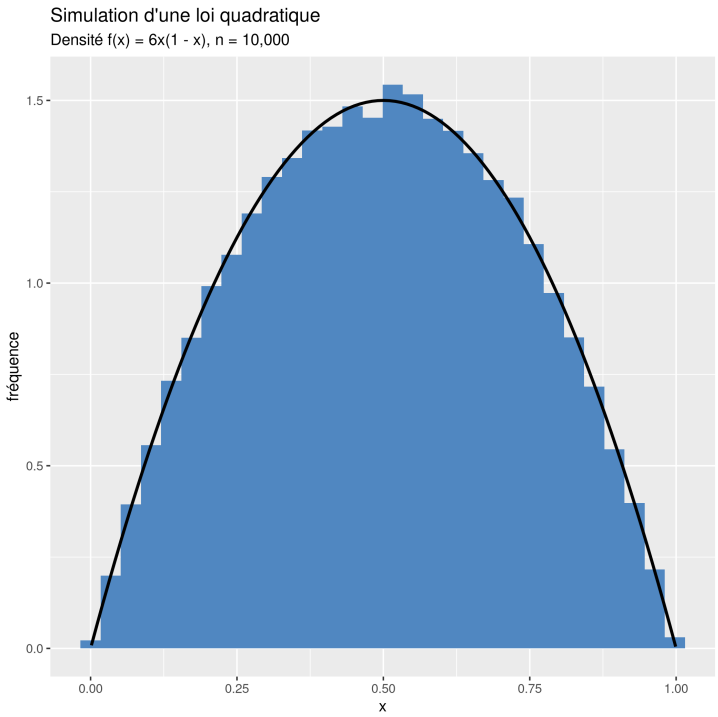


FIGURE 4 – 10,000 réalisations d'une variable aléatoire dont la densité est une fonction quadratique. On a vérifié graphiquement que la méthode d'acceptation-rejet implémenté en `rquad` permet de simuler une variable aléotoire avec $f(x) = 6x(1-x)$.

3.2 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 :

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

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 sont abscisse X, qui suit alors la loi de $f_X(x)$.

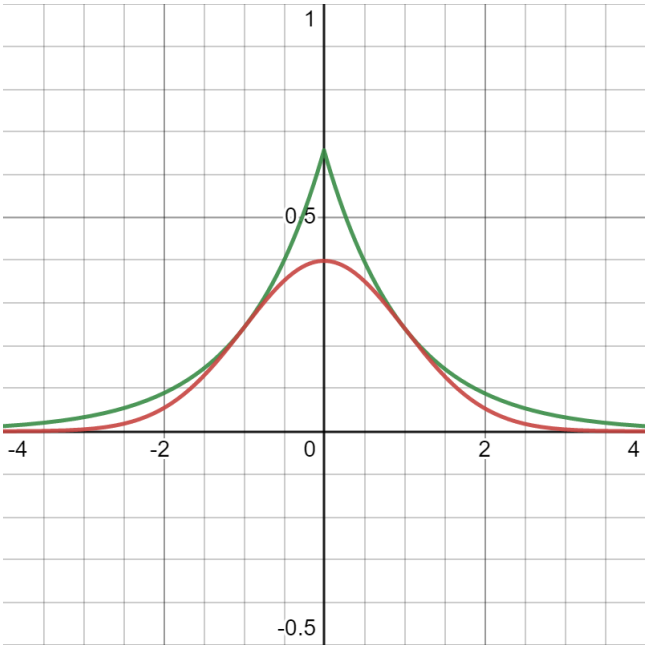


Figure 5 – Fonctions densités

On remarque que la fonction majorante en verte est extrêment serré 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
2  rnorm_acc_std <- function(n) {
3
4      x <- vector("numeric", n)
5
6      for (i in 1:n) {
7
8          reject <- TRUE
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.

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 coeficient que nous avons choisi

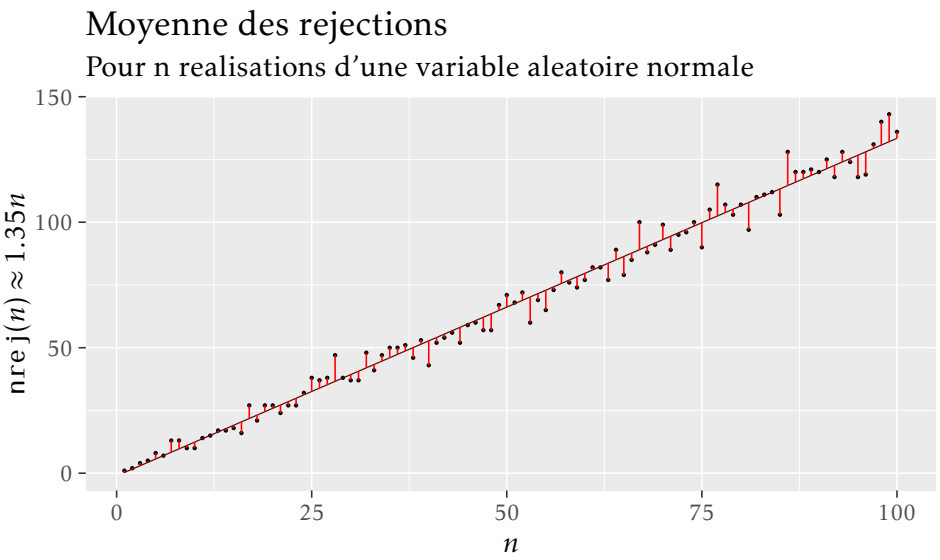


Figure 6 – Moyenne des rejets

pour la fonction majorante $g(x)$

3.3 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 :

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

On effectue le passage en coordonnées polaires

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

et on obtient

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

Dans la densité jointe des variable R et θ, on reconnaît
$\frac{1}{2\pi}$ = densité de θ qui suit une loi Uniforme sur 0 et 2 π.
$\rho\exp(-\frac{\rho^2}{2})$ =densité de R.
On en déduit la fonction de répartition de R :

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

On reconnaît une loi exponentielle de paramètres 1/2 pour R^2.
On a donc les lois de R et θ :

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

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

$$R = \sqrt[1/2]{-2\ln U_1}$$

$$\theta = 2\pi U_2$$

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 :

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

3.4 Vecteur Gaussien

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

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

et

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

Σ est appelée la matrice de variance (ou matrice de variance-covariance) de X. C'est une matrice de taille d × d qui est semi-définie positive.

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

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 **L** . Finalement la transformation affine de l'étape 3 nous permet d'avoir une vecteur gaussien avec la bonne ésperance et matrice de covariance.

Exemple :En deux dimension :

Preuve :

4 Simulation géostatistique

4.1 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 considere qu'une région par exemple d'une montaigne où le terrain dans un rectangle 2 dimensionnel a une certaine altitude, on peut imaginer que les *valeurs* de l'altitude son aléotoire, mais dans un certain sens leurs quantités sont liées géographiquement. Il existe une structure spatiale sur la région.

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

4.2 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 une **domaine** $\mathcal{D}$ sur laquelle on peut avoir des observations.

Définition 4.2.1 (Processus stochastique) *Soit* $(\Omega, \mathcal{A}, \mathbb{P})$ *un espace probabilisé, et* $\mathcal{D}$ *un espace quelconque. On appelle* ***processus stochastique*** *toute fonction* Z *:*

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

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 associé. 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 $\mathcal{D} = \mathbb{R}^2$.

4.2.1 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 concepte que pour un processus stochastique, il existe des propriétés qui sont invariants au déplacements dans l'espace. Souvent classifiés sur le principe des *moments statistiques*, la stationnarité nous permet de simplifier nos modèles mathématiqes des phenomènes réelles.

Définition 4.2.2 (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 le même partout dans l'espace $\mathcal{D}$. Cette condition impose une certaine régularité sous le processus que l'on aimerait modéliser. Au-dela, on pourrait parler de la stationnarité à l'ordre 2 qui impose une condition sur le deuxième moment de Z, la covariance :

Définition 4.2.3 (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* $h \in D$ *:*

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

.

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

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

Cette fonction $f(x_1 - h_2)$ 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 suelement 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.

4.3 Simulation spatiale

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

On voudrait simuler une réalisation de la variable régionalisé $Z(s), s_i \in \mathcal{D}$, $\mathcal{D} \in \mathbb{R}^2$. On va donc prendre une discretization de l'espace en un grille de $n \times n$ où les coordonnées $(i,j) \in \mathbb{R}^2$ de $s_i \in D, i \in [|1, n^2|]$ 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 $\texttt{coord}(s)$ sont alors

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

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

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

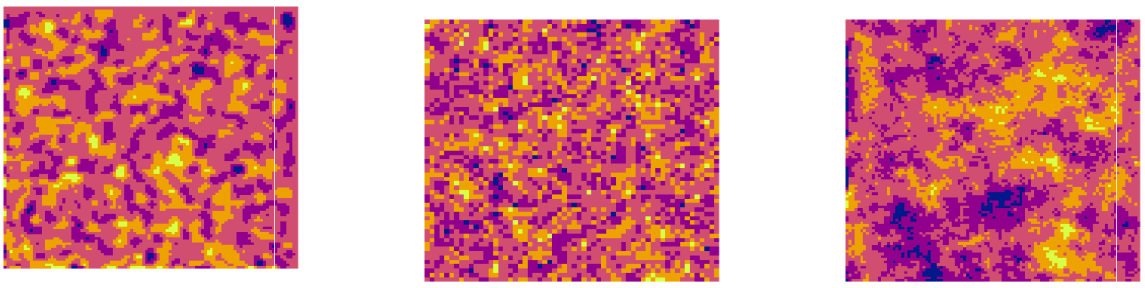
Par exemple, $d(s1, s5) = \|(1,1) - (2,2)\|_2 = \sqrt{2}$.

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 une modèle exponentielle

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

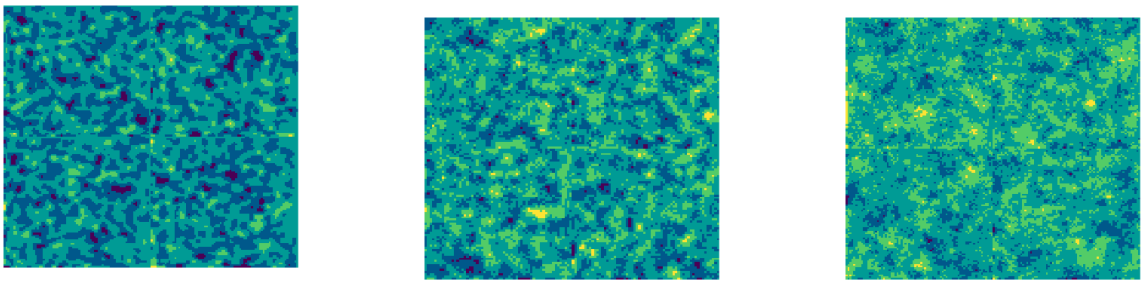
Comme on a une fonction de distance, on prend $h = d(s_i, s_j)$. C'est maintenant où on relie les des côtés de simulation : On suppose que la valeur de la variable régionalisé est un vecteur gaussien avec espérance $\vec{\mu}$ et matrice de covariance Σ. Si on prend $\vec{\mu} = \mathbf{0}$ il ne resete 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



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

Figure 8 – Three simple graphs

4.3.1 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 Σ est déterminé par l'hypothèse de stationnarité forte (à l'ordre 1 et 2) et calculé selon un modèle choisit. Nous utiliserons le modèle exponentielle pour commencer. Finalement, pour simplifier l'exemple on prend une espérance $\mu = \mathbf{0} \in \mathbb{R}^9$.

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

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

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 réecrit :

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

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 Σ est symmétrique, on rempli que le partie triangulaire à droite de H pour breveté. 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 computé 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 Σ, 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 :

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

`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

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

et gaussien

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

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 $a = 3$.

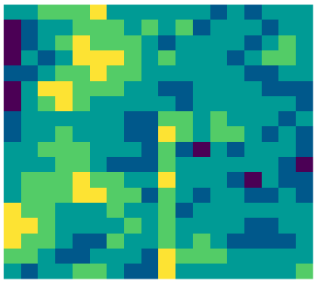


Figure 9 – Réalisation $\omega \in \Omega$ d'un champ gaussien sur $\mathcal{D} \in \mathbb{R}^2$ dans $\mathbb{R}$ avec fonction de covariance exponentielle. Les valuers $Z(s)$ sont représenté en utilisant l'échelle de couleur `viridis`.

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 Σ est de complexité $\mathcal{O}(n^4)$! Et finalement la décomposition de Cholesky est de compléxité $\mathcal{O}(n^3)$ 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 autre méthodes telle que la méthode spectrale pour simuler plus efficacement un champ gaussien de covariance exponentielle.

4.4 Krigeage

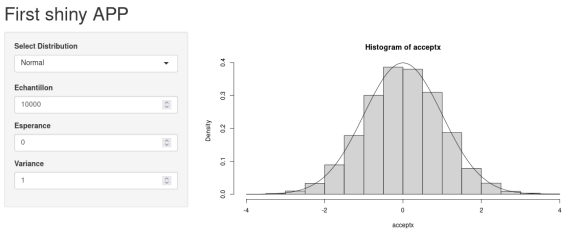
La dernière partie de notre projet s'oriente autour d'une méthode dite le krigéage, qui est effectivement une union de la simulation spatiale avec le problème numérique des moindres carrés.

5 Programmation

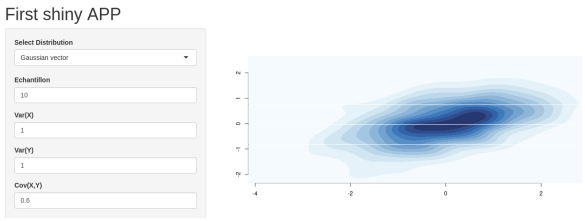
5.1 R shiny

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 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.



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



(b) Sphere

5.2 `sousmarin`

Le package R `sousmarin` est disponible sur notre reperpertoire 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 réecrites 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.

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 mutlivariée et en spatial.

Il y a plusieurs choses que nous aurions pu amélorier 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 tenter 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.

7 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 mutlivariée 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

8 Ce que nous pourrions amélirorer

8.1 Vitesse

Les boucles en R sont péniblement lentes comparé aux autre languages compilés comme le C. Donc, la plupart de nos algorithmes implémentés sont centaines 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.

9 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 nombreuse compétence 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 approfindi mes connaissances autour la génération des variables aléatoires qui est indispensable dans le domaine de processus stochastique ou des nombreuses phenomènes phsyiques 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 Forestièr

[2] Donald E. Knuth (1986) *The TEX 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