Simulation de points aléatoires indépendants et non-indépendants sur surfaces non planes

Sandro Petrillo
Université de Neuchâtel - Diplôme postgrade en statistique
Travail de diplôme

Sous la direction de: Dr. Giuseppe Melfi
Groupe de Statistique - Université de Neuchâtel

Septembre 2005

Table des matières

Table des figures

Chapitre 1

Introduction

Dans ce travail seront passés en revue deux algorithmes qui permettent de simuler sur des surfaces non planes (définies par une fonction de deux variables) des points aléatoires. Le premier algorithme simule des points pseudo-aléatoires indépendants et uniformes, tandis que le deuxième des points indépendants non-uniformes. L'objectif de ce travail est le développement et l'implémentation d'un troisième algorithme, simulant des points pseudo-aléatoires non-indépendants.

L'outil principal nécessaire à toutes ces simulations est un générateur de nombres pseudo-aléatoires uniformes standard. Les applications seront développées avec le logiciel `R` [5].

Par surface non plane on entend une surface définie par une fonction de deux variables dont le gradient n'est pas constant. On va approfondir formellement plus loin la définition des surfaces qui seront utilisées. On présente dans cette introduction quelques exemples graphiques.

FIG. 1.1 – Surface définie par la fonction $f(x, y) = 6e^{-(x^2+y^2)}$

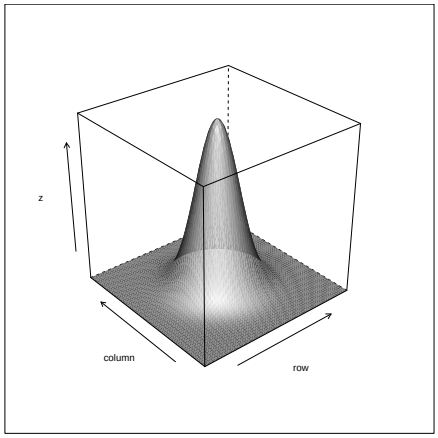
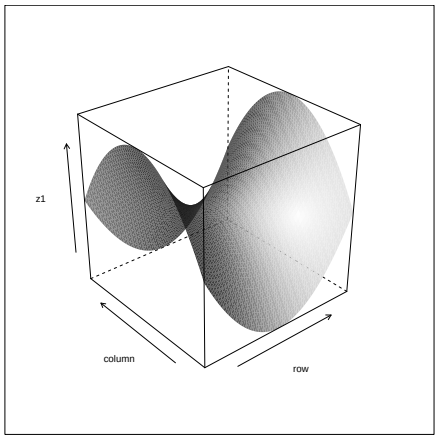


FIG. 1.2 – Surface définie par la fonction $f(x, y) = x^2 - y^2$



Les distributions aléatoires sur des surfaces non planes sont assez usuelles dans la nature. Par exemple, dans les sciences de l'environnement, la géologie, la botanique, la météorologie, on trouve des distributions aléatoires de ce type.

Les types de surfaces utilisées dans ce travail ne vont pas être trop différentes d'un relief géographique réel. Un exemple du type de problème qu'on veut résoudre ici est le suivant : on peut considérer la distribution d'arbres dans une forêt ayant un relief non plan ou la vitesse des vents sur la surface dans des régions de montagne. On pourrait s'intéresser à la quantité d'oxygène produite par des arbres d'une certaine espèce dans une forêt. Celle-ci peut dépendre de l'âge des arbres, mais aussi de la distance entre les arbres de la même espèce, ou encore d'autres facteurs comme la pente du relief et son orientation vers le nord. En considérant que les arbres sont distribués sur la surface (non plane) selon une certaine distribution de probabilité (par exemple uniformément distribués par unité de surface), les distances réciproques sont variables et pour résoudre le problème de l'estimation de la production d'oxygène de toute la forêt, il est nécessaire d'évaluer comment la production d'oxygène est distribuée parmi les arbres. Une approche naturelle est d'appliquer des techniques de Monte Carlo pour le modèle de distribution aléatoire approprié.

Le chapitre suivant est dédié aux méthodes de Monte Carlo, avant d'aborder le thème principal de ce travail, c'est-à-dire les algorithmes de simulation.

FIG. 1.3 – Surface définie par la fonction $f(x, y) = x^2 + y^2$

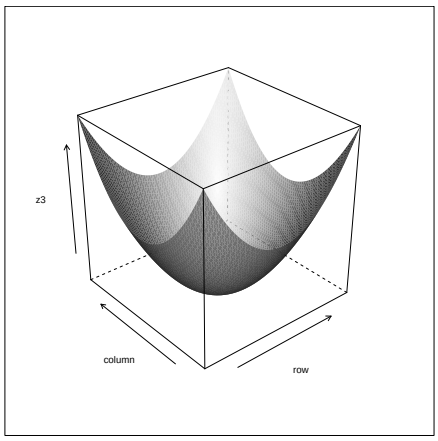
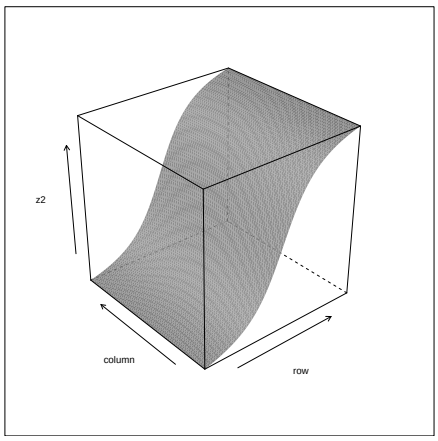


FIG. 1.4 – Surface définie par la fonction $f(x, y) = 6 \arctan(x)$



Chapitre 2

Les méthodes de Montecarlo

La méthode de Monte Carlo est définie comme toute technique numérique de résolution de problèmes mathématiques qui utilise des nombres aléatoires ou pseudo-aléatoires. On attribue la méthode de Monte Carlo développée vers 1949 aux mathématiciens américains Johannes Von Neumann et Stanislav Ulam. Ce n'est toutefois qu'avec l'avènement des ordinateurs que l'on a pu réellement utiliser cette méthode. Quant au nom de Monte Carlo, on le doit bien sûr à la capitale de la principauté de Monaco, célèbre pour son casino. En effet, la roulette est l'un des mécanismes les plus simples pour générer des nombres aléatoires.

On présente dans les sections suivantes deux méthodes très utilisées en simulation : les méthodes d'acceptation/rejet en général et l'échantilloneur de Gibbs, qui serviront à la compréhension des algorithmes des prochains chapitres.

2.1 Les méthodes d'acceptation/rejet

2.1.1 Principe général

Il y a beaucoup de distributions pour lequel il est difficile, voir impossible, de simuler directement. Dans certains cas il est d'ailleurs impossible de représenter la distribution dans une forme utilisable, comme une transformation par exemple. Dans des situations pareilles, il est impossible d'exploiter les propriétés probabilistes directes afin de dériver une méthode de simulation. La classe des méthodes d'acceptation/rejet exige uniquement la connaissance de la forme fonctionnelle de la densité f exprimée avec une constante : l'étude analytique approfondie de f n'est pas nécessaire.
La clé de cette méthode est l'utilisation d'une densité g plus simple pour la simulation. Pour une densité donnée g (appelée *densité instrumentale*) il y a plusieurs densités f (appelées les *densités cible*) qui peuvent être simulées de cette façon. On présente ci-dessous la *méthode d'acceptation/rejet* sous forme d'algorithme [6].
En considérant une densité à laquelle on s'interesse f, la première exigence est

la détermination d'une densité g et d'une constante M telles que

$$f(x) \leq M \cdot g(x)$$

sur le support de f. L'algorithme peut être décrit comme suit :

Algorithme 1 : Méthode d'acceptation/rejet

1. Générer $X \sim g$, $U \sim U(0,1)$;
2. Accepter $Y = X$ si $U \leq \dfrac{f(X)}{M \cdot g(x)}$;
3. Retourner à l'étape 1 sinon.

La méthode d'acceptation/rejet se base sur le résultat suivant :

Proposition 1 : l'algorithme 1 produit une variable Y distribuée selon f.

Démonstration : la fonction de répartition de Y est donnée par

$$P(Y \leq y) = P\left(X \leq y \mid U \leq \frac{f(X)}{M \cdot g(X)}\right) = \frac{P\left(X \leq y, U \leq \frac{f(X)}{Mg(X)}\right)}{P\left(U \leq \frac{f(X)}{Mg(X)}\right)}$$

Les variables X et U étant indépendantes, leur densité conjointe est simplement le produit de leurs densités marginales. La probabilité conjointe est le double intégrale de la densité conjointe :

$$P\left(X \leq y \mid U \leq \frac{f(X)}{M \cdot g(X)}\right) = \int_{-\infty}^{y} \int_{0}^{f(x)/Mg(x)} f_{X,U}(x,u)\, du\, dx$$

La densité conjointe $f_{X,U}(x,u)$ est :

$$f_{X,U}(x,u) = g(x) \cdot 1 = g(x)$$

En écrivant la probabilité avec les intégrales, on obtient donc

$$P(Y \leq y) = \frac{\int_{-\infty}^{y} \overbrace{\int_{0}^{f(x)/Mg(x)} du}^{=f(x)/(Mg(x))}\, g(x)\, dx}{\int_{-\infty}^{\infty} \underbrace{\int_{0}^{f(x)/Mg(x)} du}_{=f(x)/(Mg(x))}\, g(x)\, dx} = \frac{\frac{1}{M}\int_{-\infty}^{y} f(x)\, dx}{\frac{1}{M}\int_{-\infty}^{\infty} f(x)\, dx}$$

Le $1/M$ au numérateur et au dénominateur se simplifie et, vu que $\int_{-\infty}^{\infty} f(x)\, dx = 1$ (pour toutes distributions de probabilité), on obtient

$$P(Y \leq y) = \int_{-\infty}^{y} f(x)\, dx$$

ce qui démontre bien que Y suit la loi de probabilité f.

2.2 L'échantilloneur de Gibbs

L'échantilloneur de Gibbs, connu dans la littérature anglophone avec le nom de "Gibbs sampler", est une manière de générer des distributions de deux (ou plusieurs) variables à partir d'un modèle qui définit les distributions de probabilité conditionnées. Dans le cas d'un modèle à deux variables, la méthode consiste à prendre un élément de départ (X_0, Y_0) et à générer à l'aide de nombres aléatoires (X_n, Y_n) par itération suivant les règles de probabilité suivantes :

$$\begin{aligned} X_n &\sim f(X|Y_{n-1}) \\ Y_n &\sim g(Y|X_n), \end{aligned}$$

où f et g sont les densités de probabilité conditionnelles modélisées (ou connues).

L'échantilloneur de Gibbs peut se généraliser par la définition suivante :

Définition : on suppose que, pour un certain $p > 1$, la variable aléatoire $\mathbf{X} \in \chi$ peut être exprimée comme $\mathbf{X} = (X_1, \ldots, X_p)$, où les X_i sont soit uni-, soit multidimensionnels. De plus, on suppose qu'on peut simuler à partir des densités conditionnelles univariées $f_1, \ldots, f_p$, c'est-à-dire, on peut simuler :

$$X_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p \sim f_i(x_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p)$$

pour $i = 1, 2, \ldots, p$. L'échantilloneur de Gibbs associé est donné par la transition suivante de $X^{(t)}$ à $X^{(t+1)}$:

Algorithme 2 : Echantilloneur de Gibbs

Etant donné $\mathbf{x}^{(t)} = (x_1^{(t)}, \ldots, x_p^{(t)})$, générer

1. $X_1^{(t+1)} \sim f_1(x_1|x_2^{(t)}, \ldots, x_p^{(t)})$,
2. $X_2^{(t+1)} \sim f_2(x_2|x_1^{(t+1)}, x_3^{(t)}, \ldots, x_p^{(t)})$,

$\vdots$

p. $X_p^{(t+1)} \sim f_p(x_p|x_1^{(t+1)}, \ldots, x_{p-1}^{(t+1)})$

Les densités $f_1, f_2, \ldots, f_p$ sont appelées "entièrement conditionnelles"[1] et une caractéristique de l'échantilloneur de Gibbs est que ces dernières sont les seules densités utilisées pour la simulation. Donc, même dans un problème multidimensionnel, toutes les simulations peuvent être univariées, ce qui est habituellement un avantage [6].

[1] Traduit de l'anglais : *full conditionals*.

Chapitre 3

Simulation de points indépendants sur surfaces

3.1 Introduction

Dans ce chapitre seront présentés en détail deux algorithmes pour la simulation de points aléatoires indépendants sur surfaces non planes.

Le premier algorithme (appelé *URDA*[1]) simule des points aléatoires indépendants et uniformes sur une surface non plane. Le deuxième algorithme (*NURDA*[2]) permet de simuler des points aléatoires indépendants sur une surface non plane, mais pas uniformes.

3.2 Distribution uniforme : Algorithme *URDA*

Dans cette section on va présenter le premier des algorithmes pour la simulation de points aléatoires indépendants sur surfaces non planes. Dans un premier temps, on va montrer l'algorithme tel qu'il a été présenté par ces auteurs [4]. Ensuite, suivra un approfondissement avant d'arriver à des applications.

3.2.1 Description de l'algorithme

Supposons d'avoir une surface S définie comme une fonction différentiable f sur un ensemble compact $D \subset \mathbb{R}^2$, c'est-à-dire :

$$S = \left\{(x, y, f(x, y)) \in \mathbb{R}^3 \,|\, (x, y) \in D\right\}.$$

Supposons d'avoir à disposition un générateur de nombres aléatoires, c'est-à-dire une séquence $\{u_h\}_{h\in\mathbb{N}}$ avec $u_h \in (0, 1)$ qui satisfait aux conditions de [2]. Les étapes de l'algorithme sont :

Etape 1 : générer une distribution uniforme de N points dans D. Vu que D est un ensemble compact dans $\mathbb{R}^2$, il est borné et fermé et peut être contenu dans un rectangle $(a, b) \times (c, d)$. Par une transformation affine appropriée,

[1] En anglais : *Uniform Random Distribution Algorithm*
[2] En anglais : *Non Uniform Random Distribution Algorithm*

des points aléatoires distribués uniformément (u_{2k-1}, u_{2k}) dans $(0,1) \times (0,1)$ peuvent être tranformés en des points distribués uniformément dans D, sans considérer les points qui tombent éventuellement en dehors de D. Cette procédure permet de simuler un nombre arbitraire de points aléatoires distribués uniformément en D :

$$(x_i, y_i) \qquad \text{pour } i = 1, \ldots, N.$$

Etape 2 : assigner à chaque point généré en D, un nombre aléatoire, en utilisant encore le générateur de nombres pseudo-aléatoires distribués uniformément dans $(0,1)$. Cette opération peut être considérée comme une fonction :

$$\omega : \{1, \ldots, N\} \longrightarrow (0,1)$$

Etape 3 : en considérant la fonction :

$$m_1(x,y) = \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

définie sur D, calculer :

$$M_1 = \max_D \{m_1(x,y)\}$$

Vu que D est compact et f est différentiable, le maximum de $m_1(x,y)$ existe. La raison pour considérer cette fonction est qu'un élément de surface $\Delta x \Delta y$ correspondant à un point (x,y) en D est projeté à travers la fonction f dans un élément dont l'aire peut être approximée par[3] (voir [1]) :

$$\left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}} \Delta x \Delta y$$

Evidemment, $1 \leq M_1 < \infty$.

Etape 4 : sélectionner le point $(x_i, y_i, f(x_i, y_i))$ dans l'échantillon final de points aléatoires sur S si :

$$\omega(i) < \frac{m_1(x_i, y_i)}{M_1}$$

Cette procédure permet de sélectionner en moyenne le même nombre de points par unité d'aire de surface sur S.
La fonction $m_1(x,y)$ est utilisée pour corriger l'effet de la projection du plan D sur la surface S. Les nombres aléatoires $\omega \in (0,1)$ associés à chaque point simulé dans D sont comparés à la probabilité de sélection des points, qui est $\frac{m_1(x,y)}{M_1}$. Les points simulés dans D subissent une "distorsion" quand projetés sur la surface S : plus la surface est "différente" (en termes de pente) de la surface plane D et moins il y aura de points sur cette projection. La fonction $m_1(x,y)$ équilibre en quelque sorte cette distorsion. En effet, la fonction varie positivement avec la pente de la surface S, dans les directions de x et de y.

[3] Pour une explication détaillée de ce résultat, voir l'annexe A

3.2.2 Remarques sur l'algorithme *URDA*

Le nombre de points sélectionnés dans l'échantillon final, n, n'est pas fixe. C'est une fraction de N, le nombre de points simulés dans l'espace D. On a :

$$n = q \cdot N, \qquad 0 \leq q \leq 1$$

La valeur de q est proche de 1 quand la fonction $m_1(x, y)$ varie dans un interval étroit.

L'algorithme qui vient d'être présenté peut être généralisé à des distributions aléatoires plus complexes (non uniformes). Cela pourrait être utile dans des cas où la densité ne dépend pas uniquement de la pente mais aussi d'autres facteurs dont l'influence pourrait être représentée par une fonction positive $t(x, y)$ appropriée. C'est le cas de l'algorithme *NURDA* qui va être présenté dans la prochaine section.

3.3 Distribution non-uniforme : Algorithme *NURDA*

Le deuxième algorithme qu'on va présenter (toujours tiré de [4]) a pour but de simuler des points aléatoires indépendants non uniformes. La procédure est semblable à celle pour le premier algorithme, sauf que dans ce cas il y a l'introduction d'une fonction positive $t(x, y)$ qui va permettre de rendre non uniformes les points simulés sur la surface S, selon le modèle de densité que la fonction t représente.

3.3.1 Description de l'algorithme

Les hypothèses de départ sont les mêmes que celles de l'algorithme *URDA*, à savoir l'existence d'une surface S définie comme une fonction différentiable f sur un ensemble compact $D \subset \mathbb{R}^2$, c'est-à-dire :

$$S = \left\{(x, y, f(x, y)) \in \mathbb{R}^3 | (x, y) \in D\right\}$$

L'outil à disposition est un générateur de nombres (pseudo)-aléatoires. Voici les étapes de l'algorithme 2 (*NURDA* : *Non Uniform Random Distribution Algorithm*) :

Etape 1 : générer N points (x_i, y_i) pour $i = 1, \ldots, N$ dans D comme dans l'algorithme 1.

Etape 2 : affecter un nombre aléatoire uniforme $(0, 1)$ à chaque point généré comme dans l'algorithme 1.

Etape 3 : en considérant la fonction :

$$m_2(x, y) = t(x, y) \cdot \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

définie sur D, calculer :

$$M_2 = \max_D \{m_2(x, y)\}$$

Etape 4 : sélectionner le point $(x_i, y_i, f(x_i, y_i))$ dans l'échantillon final de points aléatoires dans S si :

$$\omega(i) < \frac{m_2(x_i, y_i)}{M_2}$$

Cette procédure permet de sélectionner un nombre approprié de points par unité d'aire de surface sur S, selon la densité sélectionnée $t(x, y)$.

3.3.2 Remarques sur l'algorithme *NURDA*

Comme pour l'algorithme 1 (*URDA*) le nombre de points sélectionnés dans l'échantillon final sera une fraction q, $0 \leq q \leq 1$, des N points simulés dans D. La présence de la fonction $t(x, y)$, cependant, limite ultérieurement la possibilité que la valeur de q soit proche de 1. En effet, pour que cela se passe, il ne suffit plus que la fonction $\left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$ varie dans un interval étroit : il faut aussi que la fonction $t(x, y)$ (qui est positive par définition) ne présente pas de "grandes" variations.

Le rôle de la fonction positive $t(x, y)$ est celui de rendre non uniformes les points simulés sur la surface S, selon le modèle de densité que la fonction t représente. Plus les valeurs de t sont élevées, plus la densité de points par unité de surface sera élevée et cela selon une fonction linéaire des valeurs de t.

Pour une discussion et des applications de ces deux algorithmes, voir aussi [8].

3.4 Implémentation des deux algorithmes avec le logiciel `R`

Dans cette section seront présentés et expliqués des fonctions qu'on a créé pour implémenter les algorithmes 1 et 2 (*URDA* et *NURDA*). Les étapes seront développées à travers le code `R` et expliquées au fur et à mesure. Le code complet des fonctions créées ainsi que des explications seront données dans l'annexe B. A titre d'exemple, nous considérons un domaine D et une fonction f définis à l'avance, le code étant applicable à n'importe quel domaine et/ou fonction.

3.4.1 Code `R` : Algorithme *URDA*

Etape 1 : afin de générer N points pseudo-aléatoires uniformes dans $D = (a, b) \times (c, d)$, on définit d'abord les quatre coordonnées (`aa, bb, cc, dd`). Ensuite, on génère $2N$ nombres pseudo-aléatoires u_i, $i = 1, \ldots, 2N$ distribués uniformément dans $(0, 1)$ pour leur appliquer la transformation affine suivante :

$$\begin{aligned} x_i &= u_i \cdot (b - a) + a, \qquad i = 1, 3, 5, \ldots, 2N - 1 \\ y_i &= u_i \cdot (d - c) + c, \qquad i = 2, 4, 6, \ldots, 2N \end{aligned}$$

La fonction ci-dessous `step1`, qui prend comme arguments `N` (le nombre de points qu'on veut simuler), `aa, bb, cc, dd` (correspondants aux limites de la surface $D = (a, b) \times (c, d)$), simule N points (x_i, y_i) distribués uniformément dans D et les stocke dans un objet `DD`[4] :

```
step1<-function(N=1000, aa=0, bb=1, cc=0, dd=1){
u12<-runif(2*N);
i<-1:N; i1<-2*i; i2<-2*i - 1;
x<-(u12[i1]*(bb-aa))+aa;
y<-(u12[i2]*(dd-cc))+cc;
DD<-data.frame(x,y)
print(DD)
}

DD<-step1(N=1000, aa=-3, bb=3, cc=-3, dd=3)
```

La dernière commande crée un objet `DD` avec la fonction `step1` qui vient d'être créée, contenant deux colonnes : les valeurs (x_i, y_i) correspondantes aux points aléatoires uniformes dans la surface de "départ" D définie dans $(-3, 3) \times (-3, 3)$.

Etape 2 : l'assignation d'un nombre aléatoire uniforme dans $(0, 1)$ à chaque point généré précédemment est précédée par la génération d'une suite de N (dans ce cas 1000) valeurs $\omega(i) : i = 1, \ldots, N$:

```
N<-1000
w<-runif(N)
```

Etape 3 : on considère dans cet exemple la surface S définie par la fonction :

$$f(x, y) = 6 \cdot e^{-(x^2+y^2)}$$

dans le domaine $D = (-3, 3) \times (-3, 3)$. Avec `R` on a deux possibilités pour définir cette fonction : comme objet `function` ou comme `expression`. En définissant $f(x, y)$ comme objet de type `expression` il est possible de calculer les dérivées partielles dont on a besoin pour le calcul de la fonction :

$$m_1(x, y) = \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

Voici la commande `R` :

```
fxy<-expression(6*exp(-(x^2+y^2)))
```

Les deux dérivées partielles $\dfrac{\partial f(x, y)}{\partial x}$ et $\dfrac{\partial f(x, y)}{\partial y}$ sont dans ce cas :

$$\frac{\partial f(x, y)}{\partial x} = 6e^{-(x^2+y^2)} \cdot (-2x) = -12xe^{-(x^2+y^2)}$$
$$\frac{\partial f(x, y)}{\partial y} = 6e^{-(x^2+y^2)} \cdot (-2y) = -12ye^{-(x^2+y^2)}$$

[4] Les noms `aa, bb, cc, dd` et `DD` des variables utilisées dans `R` ont été choisis de cette façon pour ne pas créer de conflits avec certains mots-clé du logiciel. Par exemple, `D()` est une fonction "interne" de `R` et donc `DD` a été choisi comme nom du domaine D pour ne pas créer de conflits.

En utilisant l'objet `fxy` il est possible de calculer ces dérivées partielles symboliquement avec la commande `D()`, qui crée un objet de type `expression`:

```
dfdx<-D(fxy, "x")
dfdy<-D(fxy, "y")
```

L'évaluation numérique des deux dérivées partielles est faite avec la commande `eval()`. On peut ainsi calculer la fonction $m_1(x, y)$:

```
m1<-sqrt(1+eval(dfdx,envir=DD)^2+eval(dfdy,envir=DD)^2)
```

Le calcul de $M_1 = \max_D m_1(x, y)$ est simple :

```
M1<-max(m1)
```

Etape 4 : il reste à sélectionner le point $(x_i, y_i, f(x_i, y_i))$ dans l'échantillon final de points aléatoires si :

$$\omega(i) < \frac{m_1(x_i, y_i)}{M_1}$$

On crée un vecteur "vide" de longeur N :

```
n<-numeric(N)
```

On crée une boucle `for` pour "remplir" le vecteur avec la valeur 1 si le point doit être sélectionné et 0 sinon :

```
for(i in 1:N){
if(w[i]<m1[i]/M1) n[i]<-1 else n[i]<-0
}
```

À ce point, il faut sélectionner les points (x_i, y_i) où le vecteur `n` vaut 1 et calculer $f(x, y)$ pour ces coordonnées :

```
S<-DD[n==1, ]
f<-eval(fxy, envir=S)
S<-data.frame(S,f)
```

Le `data.frame S` contient les coordonnées $(x_i, y_i, f(x_i, y_i))$ des points sélectionnés. Avec l'objet `S` on peut faire le graphique en trois dimensions. Avec `R` il y a deux librairies qui permettent de faire des graphiques en trois dimensions : *scatterplot3d* [3] et *lattice* [7]. Pour pouvoir utiliser ces librairies, il est nécessaire de les avoir installées[5]. Voici un exemple de la fonction `scatterplot3d` et le graphique résultant (Figure 3.1).

```
scatterplot3d(S,highlight.3d=TRUE,pch=20)
```

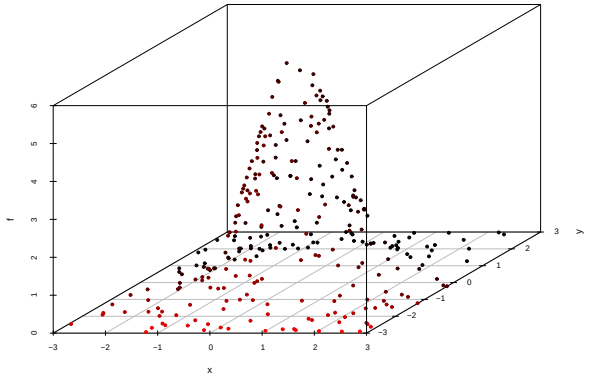
3.4.2 Code `R` : Algorithme *NURDA*

Pour l'implémentation de cet algorithme, les étapes 1 et 2 restent les mêmes que pour l'algorithme URDA. On reprend uniquement les commandes `R`, sans commentaires. La surface S prise comme exemple est la même qu'auparavant :

$$f(x, y) = 6e^{-(x^2+y^2)}$$

[5]Les librairies `R` sont librement disponibles sur le site officiel du logiciel www.r-project.org. Elles peuvent aussi être installées directement à partir de `R` avec la commande `install.packages("scatterplot3d")` et `install.packages("lattice")`, si on est connectés à internet. Pour les charger, après l'installation, il faut utiliser la commande `library(scatterplot3d)` et `library(lattice)`.

FIG. 3.1 – Résultat de l'algorithme *URDA* avec $f(x,y) = 6e^{-(x^2+y^2)}$



Etape 1 : `DD<-step1(N=1000, aa=-3, bb=3, cc=-3, dd=3)`

Etape 2 : `N<-1000;w<-runif(N)`

Etape 3 : ce qui différencie cet algorithme du précédent est la présence d'une nouvelle fonction $t(x,y)$ qui permet de rendre non uniforme la distribution sur la surface S. On considère dans cet exemple deux fonctions :

$$\begin{aligned} t_1(x,y) =& e^{-f(x,y)^2} \\ t_2(x,y) =& 3 - |3 - f(x,y)| \end{aligned}$$

Avec $f(x,y) = 6e^{-(x^2+y^2)}$, ces deux fonctions deviennent :

$$\begin{aligned} t_1(x,y) =& e^{-36e^{-2(x^2+y^2)}} \\ t_2(x,y) =& 3 - |3 - 6e^{-(x^2+y^2)}| \end{aligned}$$

On crée deux objets `t1` et `t2` de type `expression` :

```
t1<-expression(exp(-(6*exp(-x^2-y^2))^2))
t2<-expression(3-abs(3-6*exp(-x^2-y^2)))
```

Les fonctions $t_1(x,y)$ et $t_2(x,y)$ vont être utilisées pour le calcul de la fonction :

$$m_2(x,y) = t(x,y)\left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

On crée donc deux objets `m2.1` et `m2.2` avec les deux fonctions $t_1(x,y)$ et $t_2(x,y)$:

```
m2.1<-eval(t1,envir=DD)*(1+eval(dfdx,envir=DD)^2 + eval(dfdy,envir=DD)^2)^0.5
m2.2<-eval(t2,envir=DD)*(1+eval(dfdx,envir=DD)^2 + eval(dfdy,envir=DD)^2)^0.5
```

Le calcul de $M_2 = \max_D m_2(x,y)$ est simple :

```
M2.1<-max(m2.1)
M2.2<-max(m2.2)
```

Etape 4 : il reste à sélectionner le point $(x_i, y_i, f(x_i, y_i))$ dans l'échantillon final de points aléatoires sur la surface S si :

$$\omega(i) < \frac{m_2(x_i, y_i)}{M2}$$

```
n1<-numeric(N)
n2<-numeric(N)
for(i in 1:N){
if(w[i]<m2.1[i]/M2.1) n1[i]<-1
if(w[i]<m2.2[i]/M2.2) n2[i]<-1
}
f<-eval(fxy,envir=DD);frange<-range(f)
xylim<-c(-3,3)
S1<-data.frame(DD,f);S2<-S1
S1<-S1[n1==1, ];S2<-S2[n2==1, ]
```

On a maintenant deux objets (`S1` et `S2`) contenants les coordonnées $(x_i, y_i, f(x_i, y_i))$ des points simulés qui ont été sélectionnés, respectivement avec la fonction $t_1(x, y)$ et $t_2(x, y)$. Comme avant, on peut voir le résultat sur deux graphiques (Figures 3.2 et 3.3).

```
par(mfrow=c(1,2), pty="s")
scatterplot3d(S1,highlight.3d=TRUE,pch=20,xlim=xylim,ylim=xylim,zlim=frange)
scatterplot3d(S2,highlight.3d=TRUE,pch=20,xlim=xylim,ylim=xylim,zlim=frange)
par(mfrow=c(1,1))
```

FIG. 3.2 – Résultat de l'algorithme *NURDA* sur la surface $6e^{-(x^2+y^2)}$, avec $t_1(x,y) = e^{-f(x,y)^2}$

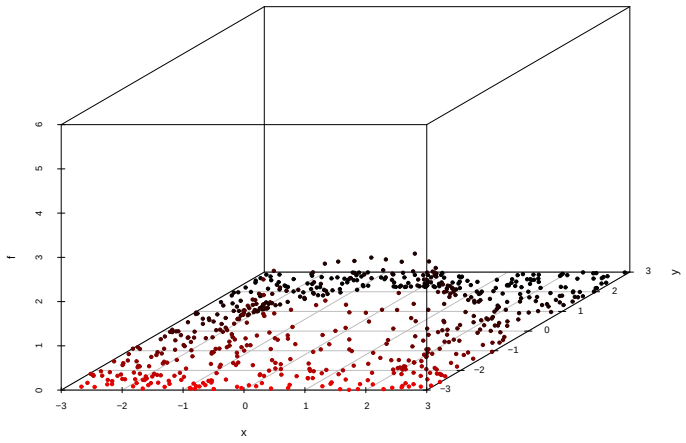
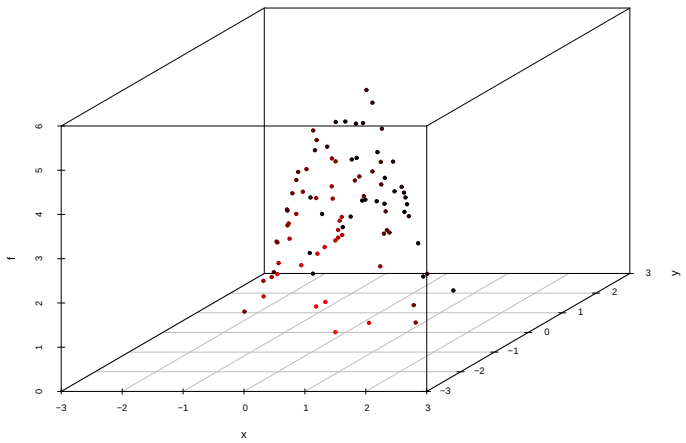


FIG. 3.3 – Résultat de l'algorithme *NURDA* sur la surface $6e^{-(x^2+y^2)}$, avec $t_2(x,y) = 3 - |3 - f(x,y)|$



Chapitre 4

Simulation de points non-indépendants sur surfaces

Dans ce chapitre on va voir un troisième algorithme de simulation de points aléatoires sur surfaces non planes. Contrairement aux deux premiers algorithmes (*URDA* et *NURDA*), qui simulaient des points indépendants, celui-ci permettra de simuler des points non-indépendants. Comme pour les deux algorithmes précédents, on va d'abord présenter les étapes de l'algorithme pour ensuite l'implémenter à l'aide du logiciel `R`.

4.1 Distribution non-indépendante : Algorithme *NIRDA*

L'algorithme *NIRDA*[1] a pour but de simuler des points non-indépendants, toujours sur des surfaces définies comme fonctions de deux variables. L'outil nécessaire est comme pour les deux algorithmes précédents un générateur de nombres pseudo-aléatoires uniformes standard $U(0,1)$.

La composante de la non-indépendance est introduite à travers l'utilisation d'une fonction positive $t_i(x,y)$ qui est directement proportionnelle à la probabilité de sélection d'un point. Cette fonction n'est pas constante et change lorsqu'un nouveau point est sélectionné dans l'échantillon final.

4.1.1 Description de l'algorithme

Les hypothèses de départ sont les mêmes que celles des deux algorithmes *URDA* et *NURDA*, c'est-à-dire l'existence d'une surface S définie comme une fonction différentiable f sur un ensemble compact $D \subset \mathbb{R}^2$:

$$S = \{(x, y, f(x,y)) \in \mathbb{R}^3 | (x,y) \in D\}$$

[1] En anglais : Non Independent Random Distribution Algorithm

Voici les étapes de l'algorithme 3 : (*NIRDA : Non Independent Random Distribution Algorithm*) :

Etape 0 : Initialiser $t_1(x, y) \equiv 1$ et $i = 1$.

Etape 1 : générer un point (x_i, y_i) dans D distribué uniformément, comme dans l'algorithme 1.

Etape 2 : affecter au point aléatoire généré (x_i, y_i) un nombre pseudo-aléatoire $\omega(i) \sim U(0, 1)$.

Etape 3 : en considérant la fonction :

$$m_{3,i}(x, y) = t_i(x, y) \cdot \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

définie sur D, calculer :

$$M_{3,i} = \max_D \{m_{3,i}(x, y)\}$$

Etape 4 : sélectionner le point $(x_i, y_i, f(x_i, y_i))$ dans l'échantillon final de points aléatoires dans S si :

$$\omega(i) < \frac{m_{3,i}(x_i, y_i)}{M_{3,i}}$$

Dans le cas contraire, retourner à l'étape 1.

Etape 5 : on met à jour $i := i + 1$ et $t_{i+1} = C(t_i, x_i, y_i)$, et on retourne à l'étape 1. C représente une opération appropriée de mise à jour qui tient compte du nouveau point sélectionné.

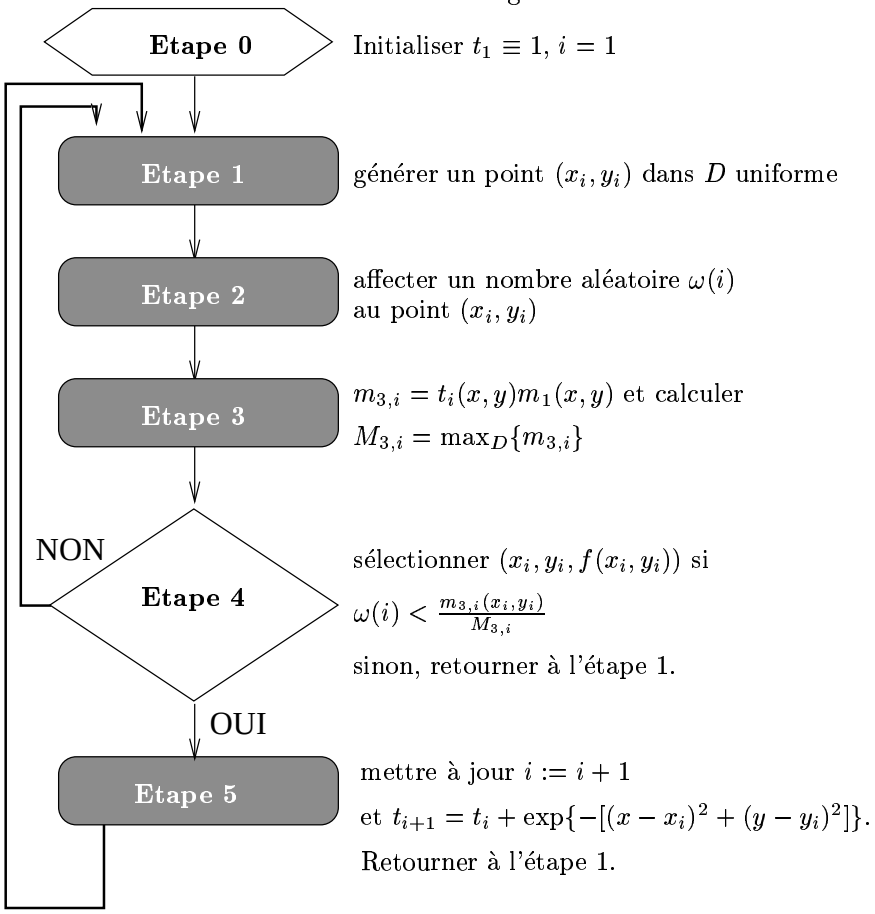
La procédure permet de simuler autant de points non-indépendants que l'on veut. Il faut quand-même dire que l'algorithme est beaucoup plus intensif au niveau computationnel et ceci à cause de la dépendance entre points qu'on a voulu introduire.

4.1.2 Remarques sur l'algorithme *NIRDA*

La logique de la non-indépendance est introduite dans l'algorithme à travers une fonction positive $t_i(x, y)$. Cette fonction n'est pas constante et change lorsqu'un nouveau point est sélectionné dans l'échantillon final sur la surface S. Une des grandes différences par rapport aux algorithmes *URDA* et *NURDA* est que dans ce cas on simule un point à la fois et non plus N points. On continue à simuler des points jusqu'au moment où un point satisfasse aux conditions de sélection-rejet. Dès qu'un point est sélectionné pour l'échantillon final de points sur S, on met à jour la fonction t_i, de façon à ce qu'elle tienne compte de l'information du dernier (ou des derniers) point(s) sélectionné(s). La fonction t_i mise à jour servira à la sélection du prochain point et elle donnera une probabilité de sélection supérieure aux points plus "proches" du point sélectionné précedemment.

Avant d'arriver à l'implementation de l'algorithme avec `R`, on présente l'algorithme sous forme de figure schématique (voir figure 4.1). Cette représentation va aider dans la compréhension de la structure de l'algorithme, afin de mieux comprendre les étapes nécessaires à son implémentation.

FIG. 4.1 – Schéma de l'algorithme *NIRDA*



Ce qui complique un peu la programmation de l'algorithme par rapport au deux précédents sont surtout les deux flèches qu'on voit partir des étapes 4 et 5 vers l'étape 1. Le retour de l'étape 4 vers l'étape 1 se passe jusqu'à la satisfaction de certaines conditions. A priori, le temps durant lequel l'algorithme doit tourner dépend de la nature des fonctions f et C et le temps peut aussi varier d'une simulation à l'autre. Dans les deux premiers algorithmes, on simulait N points aléatoires et on en choisissait une fraction inconnue à l'avance. Dans l'algorithme *NIRDA*, la logique est un peu différente dans le sens qu'on se pose l'objectif de simuler N points non-indépendants, mais ceci doit être fait à travers la programmation de deux boucles `while`, dont le temps d'exécution peut varier.

La fonction $t_i(x, y)$ est initialisée à une valeur égale à 1. Ensuite, on fait les étapes de 1 à 4 jusqu'au moment où un point est sélectionné pour l'échantillon final sur la surface S. Le premier point est sélectionné avec les mêmes critères de l'algorithme *URDA*, c'est-à-dire que, par unité de surface, en moyenne chaque point a la même probabilité d'être sélectionné. Dès que le premier point a été sélectionné, on passe à l'étape 5, où on doit mettre à jour la fonction t_i et augmenter l'indice i à $i + 1$. Ensuite on retourne à l'étape 1 et on effectue la

procédure "étape 1 - étape 4" comme avant, mais cette fois avec la nouvelle fonction t_i. La mise à jour de la fonction t_i doit tenir compte du (des) dernier(s) point(s) sélectionné(s), de façon à ce que les points "proches" de celui-ci (ceux-ci) aient une probabilité de sélection majeure.

L'opération de mise à jour C peut tenir compte de l'information concernant le dernier point sélectionné ou tous les points qui ont été sélectionnés précedemment. Pour augmenter la probabilité de sélection des points "proches" du dernier qui a été sélectionné, un exemple de fonction t_i peut être :

$$t_i(x,y) = \begin{cases} 1 & \text{pour } i = 1; \\ \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^2 & \text{pour } i = 2,\ldots,N. \end{cases}$$

Si on veut tenir compte non seulement du dernier point sélectionné, mais aussi des précédents, on peut par exemple considérer une fonction t_i qui se met à jour de la façon suivante :

$$t_i(x,y) = \begin{cases} 1 & \text{pour } i = 1; \\ t_{i-1}(x,y) + \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^2 & \text{pour } i = 2,\ldots,N. \end{cases}$$

Comme pour les algorithmes *URDA* et *NURDA*, on va présenter dans la prochaine section un exemple de l'implémentation de la procédure, selon un domaine D et des fonctions f et t_i particuliers, la procédure étant applicable à n'importe quel domaine et à n'importe quelles fonctions.

4.2 Implémentation de l'algorithme *NIRDA*

Dans un premier temps, on va présenter le code `R` pour l'algorithme *NIRDA* dans sa version avec la fonction t_i suivante :

$$t_{i+1} = \left(e^{-[(x-x_i)^2+(y-y_i)^2]}\right)^2$$

Avec une fonction pareille, la dépendance existe seulement entre un point sélectionné et le précedent. On présentera dans la suite une implémentation de l'algorithme où un point est sélectionné en tenant compte non seulement du dernier, mais aussi de tous les autres points qui ont été choisis pour l'échantillon final de points aléatoires sur la surface S. Cette dernière version utilisera une fonction t_i de la forme :

$$t_{i+1} = t_i + \left(e^{-[(x-x_i)^2+(y-y_i)^2]}\right)^2$$

4.2.1 Code `R` : Algorithme *NIRDA*, version 1

Avant de commencer avec les étapes de l'algorithme, il nous faut quelques opérations préliminaires à effectuer, comme la création d'une fonction `ti` qui sera utilisée dans l'algorithme :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^2
}
```

La fonction prend quatre arguments : `x, y` qui seront les coordonnées des points simulés et `xi, yi` les coordonnées du dernier point qui a été sélectionné. La fonction marche soit avec des arguments sous forme de vecteurs, soit avec des scalaires. Une autre opération préliminaire consiste dans la définition de la fonction décrivant la surface et de ces dérivées partielles (déjà vu avec les deux algorithmes précédents) :

```
fxy<-expression(exp(-x^2-y^2))
dfdx<-D(fxy,"x");dfdy<-D(fxy,"y");
```

On simule une grille de points (x, y) aléatoires dans D, qui vont servir à l'estimation du maximum de la fonction $m_{3,i}$ à chaque fois qu'elle vient mise à jour :

```
u12<-runif(2*1000);
i<-1:1000; i1<-2*i; i2<-2*i - 1;
x<-(u12[i1]*(bb-aa))+aa;
y<-(u12[i2]*(dd-cc))+cc;
grid.sim<-data.frame(x,y);rm(x,y);
```

L'objet créé `grid.sim` contient les coordonnées de 1000 points pseudo-aléatoires simulés dans D. La fonction :

$$m_1(x,y) = \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}}$$

peut être évaluée numériquement et stockée dans un objet `m1.temp` :

```
m1.temp<-(1+eval(dfdx,envir=grid.sim)^2+eval(dfdy,envir=grid.sim)^2)^0.5
```

La dernière opération préliminaire est la définition de deux vecteurs vides `xi` et `yi` de longueur N, le nombre de points non-indépendants qu'on veut simuler (dans ce cas $N = 10$) :

```
N<-10;
xi<-numeric(N); yi<-numeric(N);
```

Etape 0 : L'initialisation de $t_1 \equiv 1$ et de $i = 1$ se fait de la façon suivante :

```
t1<-1; i<-1;
```

Etape 1 : Il faut générer un point (pseudo)-aléatoire dans $D \subset \mathbb{R}^2$. On déclare les bornes du rectangle $(-3, 3) \times (-3, 3)$:

```
aa<-cc<- -3; bb<-dd<-3;
```

On génère deux nombres (pseudo)-aléatoires `u1` et `u2` uniformes standard :

```
u1<-runif(1);u2<-runif(1);
```

Par transformation affine on obtient un point (x_i, y_i) aléatoire uniforme dans D :

```
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)
```

On a à ce point trois objets : `x` contenant la coordonnée x du point aléatoire généré, `y` la coordonnée y et le `data.frame xy` un vecteur avec les deux coordonnées.

Etape 2 : On génère un nombre (pseudo)-aléatoire $\omega(i) \sim U(0,1)$, qu'on va appeler `w`, pour l'assigner au point (x_i, y_i) qui vient d'être créé :

```
w<-runif(1)
```

Etape 3 : A ce point, on doit considérer la fonction :

$$m_{3,i}(x,y) = t_i(x,y) \cdot \left(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2\right)^{\frac{1}{2}} = t_i(x,y) \cdot m_1(x,y)$$

et calculer son maximum :

$$M_{3,i} = \max_D \{m_{3,i}\}$$

Au premier "tour", la valeur de i et t_1 sont initialisées à 1, et on est à la recherche du premier point à sélectionner. La fonction $m_{3,i}(x,y)$ vaut, pour $i = 1$:

$$m_{3,i}(x,y) = m_{3,1}(x,y) = t_1(x,y) \cdot m_1(x,y) = 1 \cdot m_1(x,y) = m_1(x,y)$$

Avec le logiciel `R`, on évalue séparément la fonction $m_{3,i}$ d'une part avec le point candidat à la sélection pour l'échantillon (objet `m3i`) et d'autre part avec les coordonnées des 1000 points de la grille qu'on a créé avant (objet `m3i.temp`). Le maximum $M_{3,i} = \max_D\{m_{3,i}\}$ sera estimé à partir de cette grille de 1000 points aléatoires générés précedemment (objet `M3i`) :

```
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
```

Etape 4 : A ce stade, on a toutes les informations nécessaires pour décider si le point généré à l'étape 1 doit ou ne doit pas être sélectionné dans l'échantillon final de points sur la surface S. Il s'agit seulement de vérifier la condition :

$$\omega(i) < \frac{m_{3,i}(x_1, y_1)}{M_{3,1}}$$

Si cette condition est vraie, on peut passer à l'étape 5 et effectuer la mise à jour de la fonction t_i et de l'indice i ; dans le cas contraire il faudrait retourner à l'étape 1 et refaire la procédure "étape 1 - étape 4" jusqu'au moment où la condition ci-dessus soit satisfaite. On rappelle qu'on ne peut pas savoir à priori combien de fois il faut effectuer la procédure "étape 1 - étape 4" pour satisfaire la condition qui permet de passer à l'étape 5.

Etape 5 : On suppose ici que la condition de l'étape 4 a été satisfaite. Il faut mettre à jour $i = i + 1$ et la fonction

$$t_{i+1} = C(t_i, x_i, y_i) = \left(e^{-[(x-x_i)^2 + (y-y_i)^2]}\right)^2$$

où (x_i, y_i) sont les coordonnées du point généré à l'étape 1 et qui a satisfait la condition de l'étape 4. Après ces mises à jour, il faut retourner à l'étape

1 et effectuer la procédure "étape 1 - étape 4" avec la nouvelles valeurs i et t_i, jusqu'au moment où un point satisfait la condition de l'étape 4, et ainsi de suite.

On va maintenant compléter l'implémentation de l'algorithme à l'aide de deux boucles `while`, qui permettront l'execution correcte et automatique de la procédure.

Les étapes de 1 à 5 de l'algorithme doivent être exécutées selon la structure présentée dans la figure 4.1, page 21. Cette structure peut être implémentée avec l'utilisation de deux boucles `while` :

- Une première boucle `while` pour effectuer la procédure "étape 1 - étape 4" jusqu'à la satisfaction de la condition de l'étape 4 ;
- Une deuxième qui, après qu'on sort de la première boucle, permet les mises à jour nécessaires et retourne au début de la procédure "étape 1 - étape 4". Tout ceci jusqu'au moment où on termine de générer les N points non-indépendants sur la surface S.

La première boucle décrite ci-dessus peut être programmée avec la "stratégie" suivante : on crée, avant d'entrer dans la boucle, des objets `m3i, M3i` et `w` :

```
m3i<-1;M3i<-1;w<-1;
```

Le but des valeurs affectées à ces variables est seulement de rendre vraie la condition :

$$\omega(i) \geq \frac{m_{3,i}}{M_{3,i}}$$

Ensuite, on peut entrer dans la boucle :

```
while(w>=(m3i/M3i)){
u1<-runif(1);u2<-runif(1);
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)

w<-runif(1)

if(length(t1)==1){
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
}else{
m3i<-ti(x=x,y=y,xi=xi[i-1],yi=yi[i-1])*
(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5

m3i.temp<-ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=yi[i-1])*m1.temp
M3i<-max(m3i.temp)
}
}
```

La partie à l'intérieur de la boucle ne fait rien d'autre qu'effectuer la procédure "étape 1 - étape 4". Dans la boucle, on voit l'introduction d'une condition `if` : celle-ci sert à différencier la première entrée dans la boucle (première partie de la condition `if`) de toutes les autres fois qu'on entre dans la boucle. En effet, la

première fois on utilise $t_1(x, y) \equiv 1$ et $i = 1$: à ce point on est à la recherche du premier point qu'on veut sélectionner sur la surface S.

La boucle `while` ci-dessus, une fois executée, sortira quand un point simulé (x_1, y_1) rendra fausse la condition $\omega(1) \geq \frac{m_{3,1}}{M_{3,1}(x_1,y_1)}$. Pour permettre le passage à l'étape 5, on introduit une boucle en "dehors" de celle qu'on vient de présenter. En appelant la boucle ci-dessus "étape 1 - étape 4", la nouvelle boucle se présente de la façon suivante :

```
while(i<=N){
m3i<-1;M3i<-1;w<-1;

#########################
Boucle "ETAPE 1 - ETAPE 4"
#########################

t1<-c(t1,1)
xi[i]<-x;yi[i]<-y;
i<-i+1
}
```

La première ligne juste en dessous de la commande `while` a le seul but de rendre vraie la condition pour entrer dans la boucle "étape 1 - étape 4". Les trois dernières lignes représentent les mises à jour :

- la commande `t1<-c(t1,1)` sert uniquement à augmenter à chaque fois la longueur de l'objet `t1`. Celui-ci, quand de longeur différente de 1 (donc, après la sélection du premier point de l'échantillon final sur la surface S), fait en sorte que, dans la boucle "étape 1 - étape 4", la partie `else` de la condition `if` est executée ;
- la commande `xi[i]<-x ;yi[i]<-y ;` ne fait rien d'autre que stocker les coordonnées du dernier point sélectionné pour l'échantillon final dans les vecteurs créés précédemment.
- la dernière ligne, `i<-i+1`, met à jour l'indice i en l'augmentant de 1.

La fonction t_i est mise à jour à l'intérieur de la boucle "étape 1 - étape 4", à travers le nouvel indice i mis à jour. Si on reprend la partie `else` de la condition `if` :

```
...

}else{
m3i<-ti(x=x,y=y,xi=xi[i-1],yi=yi[i-1])*
(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5

m3i.temp<-ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=yi[i-1])*m1.temp
M3i<-max(m3i.temp)
}
```

on voit comme la fonction t_i (`ti` dans le code `R`) est mise à jour : elle tient compte du dernier point sélectionné en utilisant la fonction `ti` définie à l'extérieur de toute la procédure, à travers les arguments `xi=xi[i-1],yi=yi[i-1]`. Le code complet de la fonction `nirda()`, permettant l'exécution de l'algorithme, est présenté dans l'annexe B.3.

4.2.2 Code R : Algorithme *NIRDA*, version 2

La version de l'algorithme *NIRDA* vue ci-dessus est implémentée de façon à ce que la fonction t_i se met à jour de la manière suivante :

$$t_{i+1}(x,y) = \left(e^{-[(x-x_i)^2+(y-y_i)^2]}\right)^2$$

Avec une fonction de ce type, seulement l'information concernant le point sélectionné juste avant est considérée. Ce qu'on veut introduire dans cette version est une dépendance de tous les points qui ont été sélectionnés en précédence. La mise à jour de la fonction t_i sera de la forme :

$$t_{i+1} = t_i + \left(e^{-[(x-x_i)^2+(y-y_i)^2]}\right)^2$$

Avec une mise à jour pareille, à chaque nouveau point qu'on sélectionne, on tient compte de tous les points qui ont été sélectionnés jusqu'à ce moment.

Pour effectuer une mise à jour de la fonction $t_i(x,y)$ "additive", on crée un vecteur, qu'on nomme `t.old`, où, après la sélection d'un nouveau point, on ajoute l'information concernant le dernier point sélectionné. La fonction $t_i(x,y)$ mise à jour servira pour le calcul de la fonction $m_{3,i}$ sur la grille de 1000 points aléatoires simulés, ainsi que pour l'estimation de son maximum $M_{3,i}$. Cette opération peut être faite juste après la sélection d'un point pour l'échantillon final, en dehors de la boucle `while` pour la recherche d'un nouveau point candidat.
On crée aussi une variable `ti.old`, qui a le même rôle du vecteur `t.old`, mais sert au calcul de la fonction $m_{3,i}(x,y)$ pour le point candidat à la sélection pour l'échantillon final de points sur la surface S. La mise à jour de cette dernière variable, doit être executée à l'intérieur de la boucle `while` pour la recherche d'un nouveau point, parce qu'elle nécessite des coordonnées du point candidat, même dans le cas où ce dernier ne sera pas sélectionné.

Les opérations préliminaires restent les mêmes : définition d'une fonction $t_i(x,y)$ et $f(x,y)$, calcul de ces dérivées partielles, génération d'une grille de points aléatoires dans le domaine D, évaluation de la fonction $m_1(x,y)$ sur la grille, définition du nombre de points qu'on veut simuler, création des vecteurs qui vont contenir les coordonnées des points non-indépendants qu'on veut simuler. On reprend les commandes R :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^2
}

fxy<-expression(exp(-x^2-y^2))
dfdx<-D(fxy,"x");dfdy<-D(fxy,"y");

u12<-runif(2*1000);
i<-1:1000; i1<-2*i; i2<-2*i - 1;
```

```
x<-(u12[i1]*(bb-aa))+aa;
y<-(u12[i2]*(dd-cc))+cc;
grid.sim<-data.frame(x,y);rm(x,y);

m1.temp<-(1+eval(dfdx,envir=grid.sim)^2+eval(dfdy,envir=grid.sim)^2)^0.5

N<-10;
xi<-numeric(N); yi<-numeric(N);
```

Etape 0 : L'initialisation de $t_1 \equiv 1$ et de $i = 1$ se fait de la façon suivante :

```
t1<-1; i<-1;
```

On crée aussi le vecteur `t.old` et la variable `ti.old`, qui serviront pour la mise à jour de la fonction t_i de façon "additive", pour la grille et pour le point candidat respectivement :

```
t.old<-rep(1,1000);ti.old<-1;
```

Etape 1 : Exactement comme pour la version 1 de l'algorithme, on génère les coordonnées d'un point aléatoire dans $D = (a, b) \times (c, d) = (-3, 3) \times (-3, 3)$. On reprend uniquement les commandes `R` :

```
aa<-cc<- -3; bb<-dd<-3;
u1<-runif(1);u2<-runif(1);
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)
```

Etape 2 : On génère un nombre (pseudo)-aléatoire $\omega(i) \sim U(0, 1)$, qu'on va appeler `w`, pour l'assigner au point (x_i, y_i) qui vient d'être créé (exactement comme pour la version 1 de l'algorithme) :

```
w<-runif(1)
```

Etape 3 : Comme pour la version 1, on évalue la fonction $m_{3,i}(x, y)$ pour la grille et pour le point candidat, ainsi que l'estimation de son maximum $M_{3,i}$:

```
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
```

Etape 4 : A ce stade, on a toutes les informations nécessaires pour décider si le point généré à l'étape 1 doit ou ne doit pas être sélectionné dans l'échantillon final de points sur la surface S. Il s'agit seulement de vérifier la condition :

$$\omega(i) < \frac{m_{3,i}(x_1, y_1)}{M_{3,1}}$$

Si cette condition est vraie, on peut passer à l'étape 5 et effectuer la mise à jour de la fonction t_i et de l'indice i ; dans le cas contraire il faudrait retourner à l'étape 1 et refaire la procédure "étape 1 - étape 4" jusqu'au moment où la condition ci-dessus soit satisfaite. On rappelle qu'on ne peut pas savoir à priori combien de fois il faut effectuer la procédure "étape 1 - étape 4" pour satisfaire la condition qui permet de passer à l'étape 5.

Etape 5 : On suppose ici que la condition de l'étape 4 a été satisfaite. Il faut mettre à jour $i = i + 1$ et la fonction

$$t_{i+1} = C(t_i, x_i, y_i) = t_i + \left(e^{-[(x-x_i)^2+(y-y_i)^2]}\right)^2$$

où (x_i, y_i) sont les coordonnées du point généré à l'étape 1 et qui a satisfait la condition de l'étape 4. Après ces mises à jour, il faut retourner à l'étape 1 et effectuer la procédure "étape 1 - étape 4" avec les nouvelles valeurs i et t_i, jusqu'au moment où un point satisfait la condition de l'étape 4, et ainsi de suite. Une fois que le premier point aura été sélectionné, on mettra à jour le vecteur `t.old` avant d'entrer dans la boucle `while` pour la recherche d'un nouveau point candidat. On mettra à jour la variable `ti.old` à l'intérieur de la boucle (voir ci-après).

La structure de l'algorithme avec les deux boucles `while` reste la même que celle de la version 1 ; la seule chose qui change est la mise à jour de la fonction t_i.

La première boucle, qui effectue la procédure "étape 1 - étape 4", peut être résumée par les commandes suivantes :

```
while(w>=(m3i/M3i)){
u1<-runif(1);u2<-runif(1);
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)

w<-runif(1)

if(length(t1)==1){
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
}else{
ti.old<-1
for(k in 1:(i-1)){
#Step 5
ti.old<-ti.old+ti(x=x,y=y,xi=xi[i-k],yi=yi[i-k])
}
m3i<-ti.old*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
}
}
```

La chose importante à remarquer est la partie `else` de la condition `if` ci-dessus. En effet, c'est dans cette partie de code que la mise à jour de la fonction t_i a lieu. La procédure entre dans cette partie dès que le premier point a été sélectionné. La fonction t_i est évaluée pour le point candidat en question et tient compte de tous les points qui ont été sélectionnés précédemment (vecteur `xi`, voir ci-après). Pour tenir compte des points déjà sélectionnés, une boucle `for` est introduite. Quant à la mise à jour de la fonction t_i pour la grille, on peut l'introduire dans l'algorithme avant d'entrer dans la boucle `while` "étape 1 - étape 4", de la façon suivante :

```
while(i<=N){
m3i<-1;M3i<-1;w<-1;
if(length(t1)!=1){
#Step 5
t.old<-t.old+ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=yi[i-1])
m3i.temp<-t.old*m1.temp
M3i<-max(m3i.temp)
}

#########################
Boucle "ETAPE 1 - ETAPE 4"
#########################

t1<-c(t1,1)
xi[i]<-x;yi[i]<-y;
i<-i+1
}
```

La première ligne juste en dessous de la commande `while(i<=N){` a le seul but de rendre vraie la condition pour entrer dans la boucle "étape 1 - étape 4". La condition `if` juste en dessous, par contre, est évaluée seulement à partir de la sélection du deuxième point. La commande :

```
t.old<-t.old+ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=yi[i-1])
```

utilise les coordonnées du dernier point sélectionné et ajoute chaque fois la fonction t_i dans le vecteur `t.old`. Ce vecteur contiendra donc les valeurs de la fonction t_i mise à jour de façon additive, pour les points de la grille. Ces valeurs serviront à l'estimation du maximum de la fonction $m_{3,i}(x, y)$ (objet `M3i`).

Chapitre 5

Conclusions

Ce travail avait le but de continuer [4] et [8]. L'implémentation de l'algorithme *NIRDA* n'avait jamais été faite et les résultats obtenus dans ce travail sont satisfaisants. Cependant, il serait intéressant d'approfondir l'argument.
Des applications pratiques des algorithmes *URDA* et *NURDA* ont été faites dans [8]. Une suite logique consisterait dans l'application de l'algorithme *NIRDA* à des cas ou exemples réels.
De plus, on pourrait se concentrer sur l'étude de différentes fonctions t_i, afin de simuler la dépendance entre points de plusieurs manières.
Des idées pour des applications de l'algorithme *NIRDA*, dans ces deux versions, seraient par exemple :

- La simulation des déplacements aléatoires d'un animal d'une certaine espèce sur une surface particulière (algorithme *NIRDA, version* 1) ;
- La reforestation d'une espèce d'arbres ou le repeuplement d'une espèce d'animaux, sur une surface particulière (algorithme *NIRDA, version* 2).

Une option qu'on pourrait aussi étudier pour la suite serait de trouver une fonction t_i qui, au lieu de favoriser la génération de points dans les zones où d'autres points ont été sélectionnés, fairait presque le contraire, c'est-à-dire de pénalizer les zones déjà "peuplées".

Annexe A

Quelques résultats

A.1 Produit vectoriel et propriétés métriques

Soit a_1, a_2, a_3 et b_1, b_2, b_3 les composantes respectives des vecteurs $\mathbf{a}$ et $\mathbf{b}$. On appelle produit vectoriel de $\mathbf{a}$ et $\mathbf{b}$, et on note $\mathbf{a} \times \mathbf{b}$, le vecteur

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \tag{A.1}$$

Le produit vectoriel jouit des propriétés métriques suivantes :

1. Le produit vectoriel des vecteurs $\mathbf{a}$ et $\mathbf{b}$ est orthogonal à $\mathbf{a}$ et $\mathbf{b}$;
2. La norme du produit vectoriel $\mathbf{a} \times \mathbf{b}$ est $||\mathbf{a}|| \cdot ||\mathbf{b}|| \cdot \sin\theta$, où θ est l'angle de $\mathbf{a}$ et $\mathbf{b}$.

Il est intéressant de remarquer que la norme du produit vectoriel représente l'aire du parallélogramme construit sur des représentants de $\mathbf{a}$ et $\mathbf{b}$ d'origine commune (voir Figure A.2). En effet, l'aire du parallélogramme est égale à sa base multipliée par son hauteur. Comme base on peut prendre la longueur du vecteur $\mathbf{a}$ (c'est-à-dire sa norme) et comme hauteur la longueur de la projection orthogonale du vecteur $\mathbf{b}$ sur $\mathbf{a}$, qui mesure $||\mathbf{b}|| \sin\theta$ (où θ est la mesure de l'angle aigu formé par les deux vecteurs, voir Figure A.1).

FIG. A.1 – Aire du parallélogramme formé par les vecteurs $\mathbf{a}$ et $\mathbf{b}$

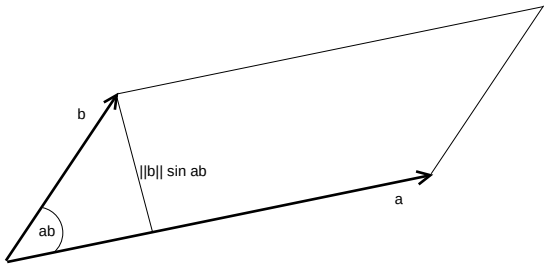
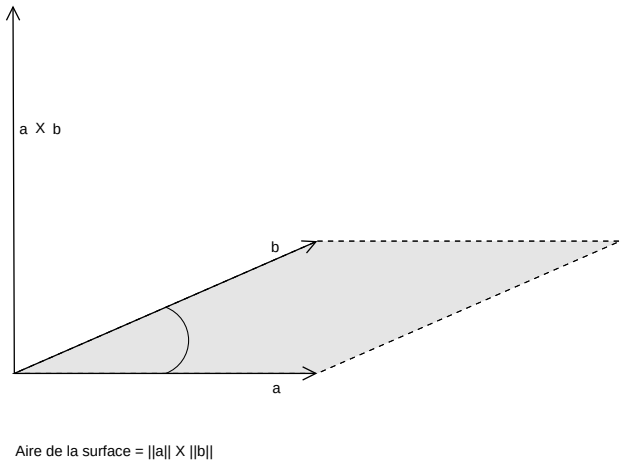


FIG. A.2 – Produit vectoriel et aire du parallélogramme



Les résultats montrés ci-dessus vont servir à l'explication du coefficient d'uniformisation présenté au point 3.2.1, page 11. Pour plus de détails, voir l'annexe A.2.

A.2 Projection d'une surface plane sur une surface non plane

On considère une surface S définie comme une fonction différentiable f sur un ensemble compact $D \subset \mathbb{R}^2$, c'est-à-dire :

$$S = \left\{(x, y, f(x, y)) \in \mathbb{R}^3 | (x, y) \in D\right\}$$

En partant d'un point (x_i, y_i) dans D et en se déplaçant de Δx en direction de x et de Δy en direction de y, on veut trouver l'aire de la projection du rectangle d'aire $\Delta x \Delta y$ sur la surface S. Les vecteurs définissants les incréments Δx et Δy appliqués au point (x, y) peuvent s'exprimer, dans le système à trois coordonnées par :

$$\mathbf{\Delta x} = \begin{pmatrix} \Delta x \\ 0 \\ 0 \end{pmatrix} \quad , \mathbf{\Delta y} = \begin{pmatrix} 0 \\ \Delta y \\ 0 \end{pmatrix}$$

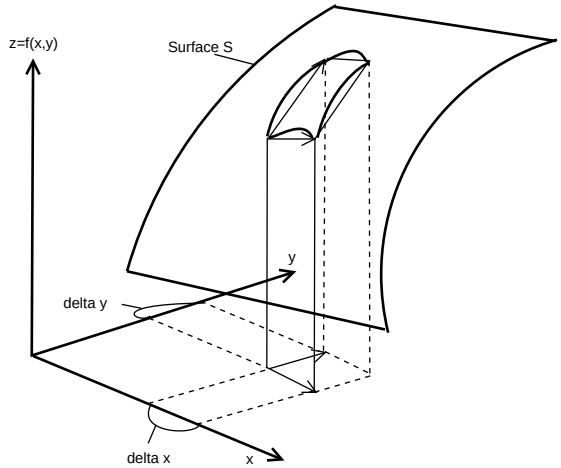
En projetant le rectangle d'aire $\Delta x \Delta y$ sur la surface S, on aura une surface dont l'aire ne sera pas égale à $\Delta x\, \Delta y$, mais soit plus grande, soit égale dans le cas d'un gradient nul. On peut estimer l'aire de la projection sur S en calculant l'aire du parallélogramme formé par la projection des vecteurs $\mathbf{\Delta x}$ et $\mathbf{\Delta y}$. Cette estimation correspond à l'approximation donnée par un développement de Taylor de la fonction "aire de la projection" tronqué au premier ordre. Ces vecteurs deviennent :

$$\begin{pmatrix} \Delta x \\ 0 \\ \Delta x f_x(x, y) \end{pmatrix} \quad , \begin{pmatrix} 0 \\ \Delta y \\ \Delta y f_y(x, y) \end{pmatrix}$$

L'aire du parallélogramme formé par ces deux vecteurs peut être calculée grâce au résultat présenté en A.1. Cette aire est égale à la norme du produit vectoriel de ces deux vecteurs, qui vaut :

$$\begin{pmatrix} \Delta x \\ 0 \\ \Delta x f_x(x,y) \end{pmatrix} \times \begin{pmatrix} 0 \\ \Delta y \\ \Delta y f_y(x,y) \end{pmatrix} = \begin{pmatrix} -\Delta x f_x(x,y)\Delta y \\ -\Delta x \Delta y f_y(x,y) \\ \Delta x \Delta y \end{pmatrix}$$

FIG. A.3 – Projection d'une surface $\Delta x\, \Delta y$ dans D sur une surface S non plane



La norme de ce dernier vecteur vaut :

$$\begin{aligned} \|\mathbf{\Delta x} \times \mathbf{\Delta y}\| &= \sqrt{(-\Delta x f_x(x,y)\Delta y)^2 + (-\Delta x \Delta y f_y(x,y))^2 + (\Delta x \Delta y)^2} \\ &= \sqrt{\Delta x^2 f_x(x,y)^2 \Delta y^2 + \Delta x^2 \Delta y^2 f_y(x,y)^2 + \Delta x^2 \Delta y^2} \\ &= \sqrt{\Delta x^2 \Delta y^2 (f_x(x,y)^2 + f_y(x,y)^2 + 1)} \\ &= \Delta x \Delta y \sqrt{f_x(x,y)^2 + f_y(x,y)^2 + 1} \end{aligned}$$

Le fait que

$$\lim_{\Delta x \to 0, \Delta y \to 0} \frac{\|\mathbf{\Delta x} \times \mathbf{\Delta y}\|}{\Delta x \Delta y} = \sqrt{(1 + \left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

prouve l'approximation de Taylor au premier ordre. L'erreur commis est de l'ordre de $(\Delta x \Delta y)^2$ et il tend à zéro quand $\Delta x \to 0$ et $\Delta y \to 0$.

Annexe B

Fonctions R complètes pour les algorithmes

Dans cet annexe on présente le code complet des fonctions qu'on a créé pour l'implémentation des trois algorithmes *URDA*, *NURDA* et *NIRDA*, ainsi que les explications nécessaires à l'utilisation de celles-ci. Le code est presque le même que celui présenté dans les chapitres du document, sauf que chaque fonction peut être utilisée pour n'importe quel domaine, fonction f, t, t_i. Les fonctions ont été programmées avec des arguments de défaut, qui peuvent être changés. A la fin de chaque algorithme, on présente quelques exemples avec différents arguments, afin d'aider la compréhension de l'utilisation des fonctions créées.

Pour pouvoir utiliser les fonctions dont le code R est présenté ci-dessous, il suffit de copier tout dans la ligne de commande du logiciel ; après les fonctions seront disponibles.

B.1 URDA

B.1.1 Code de la fonction `urda()`

```
#automated function for algorithm 1 (URDA)
urda<-function(N=1000,aa=-3,bb=3,cc=-3,dd=3,fxy=expression(6*
exp(-x^2-y^2))){
#creation of the compact set DD (aa,bb)x(cc,dd)
#Step 1: generation of N points in DD (uniformly distributed)
u12<-runif(2*N);i<-1:N; i1<-2*i; i2<-2*i - 1;
xi<-(u12[i1]*(bb-aa))+aa;
yi<-(u12[i2]*(dd-cc))+cc;
DD<-data.frame(xi,yi)
names(DD)<-c("x","y")

#Step 2: assignement of a uniform random number (0,1) to each
#random point generated in DD
w<-runif(N)
```

```
#Step 3: consider the function
#m1(x,y)=(1 + (d fxy/ d x)^2 + (d fxy/ d y)^2)^0.5 defined on DD
dfdx<-D(fxy,"x");dfdy<-D(fxy,"y");
m1<-sqrt(1+eval(dfdx,envir=DD)^2+eval(dfdy,envir=DD)^2)
#Compute M1=max_d(m1)
M1<-max(m1)

#Step 4: select the point (xi, yi, f(xi,yi)) in the final sample of
#random points on S if w_i<m1(x,y)/M1
i3<-numeric(N)
for(i in 1:N){
if(w[i]<m1[i]/M1) i3[i]<-1 else i3[i]<-0
}
f<-eval(fxy, envir=DD);frange<-range(f);
DD<-data.frame(DD,f);DDtot<-DD;DD<-DD[i3==1, ];

require(scatterplot3d)
s3d<-scatterplot3d(DD,highlight.3d=TRUE,pch=20,xlim=c(aa,bb),
ylim=c(cc,dd),
zlim=frange,main=fxy,sub=paste("Selected points: ",nrow(DD),
"over ",N))
list(DD=DD,selected=nrow(DD), prop.selected=nrow(DD)/N)
}
```

B.1.2 Explications pour l'utilisation de la fonction `urda()`

La fonction `urda()` qu'on vient de présenter prend les arguments suivants :

- `N` : le nombre de points qu'on veut simuler. Les points faisant partie de l'échantillon final seront seulement une partie (variable) de N. La valeur de défaut est `N=1000`;
- `aa, bb, cc, dd` : ce sont les coordonnées du rectangle $(a, b) \times (c, d)$ de $D \subset \mathbb{R}^2$. Les valeurs de défaut sont `aa=-3, bb=3, cc=-3, dd=3` qui correspondent au rectangle $(-3, 3) \times (-3, 3)$;
- `fxy` : la fonction $f(x, y)$ qui définit la surface S sur laquelle on veut simuler des points aléatoires (uniformes). La valeur de défaut est :
  `expression(6*exp(-x^2-y^2))`
  correspondant à la fonction $f(x, y) = 6e^{-(x^2+y^2)}$. Cet argument doit être introduit sous la forme d'un objet de type `expression`, pour permettre le calcul des dérivées partielles.

B.1.3 Exemples d'utilisation de la fonction `urda()`

En utilisant la fonction sans arguments, l'algorithme *URDA* est executé avec les valeurs de défaut. Donc, la commande :

```
urda()
```

simule $N = 1000$ points dans l'ensemble compact $D \subset \mathbb{R}^2$ (dans le carré $(-3, 3) \times (-3, 3)$), sur la surface définie par la fonction $f(x, y) = 6e^{-(x^2+y^2)}$. Seulement une partie de ces 1000 points est sélectionnée dans l'échantillon final

de points sur S.

On présente ci-dessous quelques exemples de la fonction `urda()`, en utilisant différentes fonctions f, domaines D et nombre de points qu'on simule. Par exemple, si on veut simuler 1500 points uniformes sur la surface définie par la fonction :

$$f(x, y) = x^2 + y^2$$

dans le domaine $(-1, 1) \times (-1, 1)$, il faut utiliser la commande suivante :

```
urda(N=1500, aa=-1, bb=1, cc=-1, dd=1, fxy=expression(x^2+y^2))
```

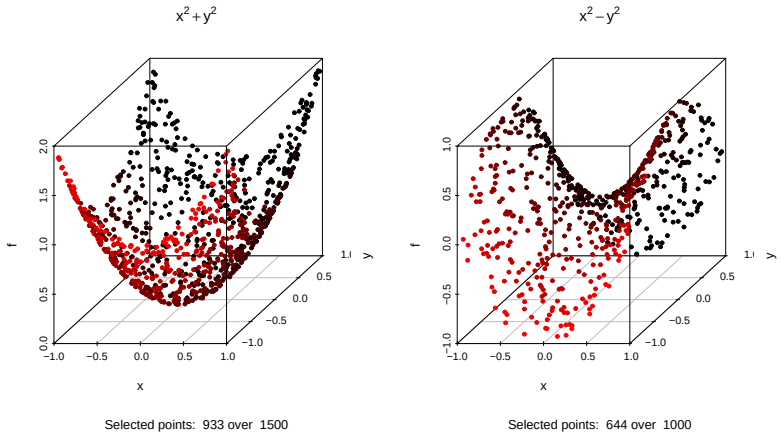
On peut aussi déclarer les arguments qu'on veut passer à la fonction séparément. Par exemple, les commandes :

```
l.bound<- -1; u.bound<- 1;
f<- expression(x^2-y^2);
urda(aa=l.bound,bb=u.bound,cc=l.bound,dd=u.bound,fxy=f)
```

vont simuler des points aléatoires uniformes sur la surface $f(x, y) = x^2 - y^2$ définie dans le domaine $D = (-1, 1) \times (-1, 1)$, en partant de la simulation de $N = 1000$ points dans D (argument de défaut).

Si on a installé la librairie `scatterplot3d`[3], la fonction crée automatiquement le graphique en trois dimensions (voir figure B.1).

FIG. B.1 – Résultat de l'algorithme *URDA* avec $f(x, y) = x^2 + y^2$ (à gauche) et $f(x, y) = x^2 - y^2$ (à droite) dans le domaine $D = (-1, 1) \times (-1, 1)$



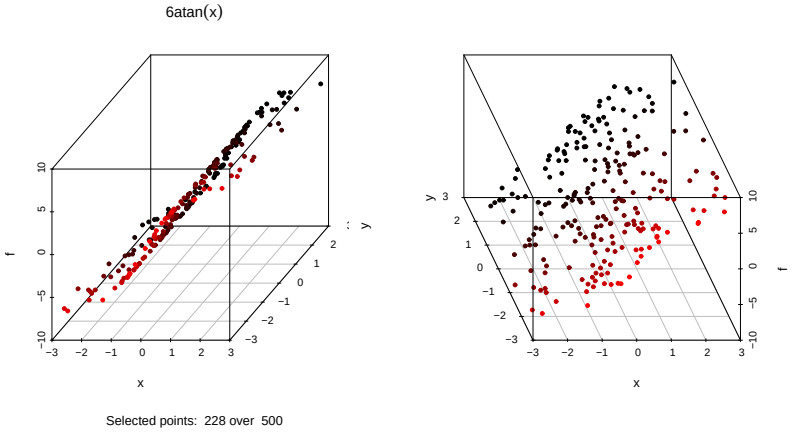
Une dernière chose à remarquer est la possibilité de mémoriser le résultat d'une simulation dans un objet `R`. Par exemple, la commande :

```
sim.atan<-urda(N=500, fxy=expression(6*atan(x)))
```

va stocker les résultats de la simulation sur la surface $f(x,y) = 6\arctan x$ (dans $D = (-3,3) \times (-3,3)$, le domaine de défaut) dans l'objet `sim.atan`. Ce dernier est un objet de type `list`, contenant les informations suivantes :

- `DDtot`, contenant les coordonnées (x,y) des points (pseudo)-aléatoires simulés initialement dans D, les valeurs de la fonction $f(x,y)$ correspondantes, les nombres $\omega \sim U(0,1)$ assignés, les probabilités de sélection $m_1(x,y)/M_1$ et l'information `TRUE / FALSE` indiquant si le point est ou non sélectionné dans l'échantillon final de points sur la surface S ;
- `DD`, contenant les coordonnées $(x, y, f(x,y))$ des points qui ont été sélectionnés dans l'échantillon final de points sur la surface S ;
- `selected`, indiquant le nombre de points qui ont été sélectionnés ;
- `prop.selected`, indiquant la proportion de points sélectionnés dans l'échantillon final.

FIG. B.2 – Résultat de l'algorithme *URDA* avec $f(x,y) = 6\arctan x$ dans $D = (-3,3) \times (-3,3)$, vue de différents points de vue



Un des avantages de mémoriser le résultat d'une simulation dans un objet est de pouvoir utiliser les résultats par exemple pour changer les coordonnées du graphique, ou d'exporter les résultats dans un fichier texte permettant l'utilisation des données avec d'autres logiciels.
Les compostantes de l'objet créé peuvent être invoquées avec le simbole `$`. Dans notre exemple, on peut voir les coordonnées des points sélectionnés dans l'échantillon final avec la commande :

```
sim.atan$DD
```

On peut par exemple faire le graphique en trois dimensions sous un autre point de vue (voir figure B.2) avec la commande[1] :

```
scatterplot3d(sim.atan$DD, angle=120, highlight.3d=TRUE,pch=20)
```

B.2 NURDA

B.2.1 Code de la fonction `nurda()`

```
#automated function for algorithm 2 (NURDA)
nurda<-function(N=1000, aa=-3, bb=3, cc=-3, dd=3,
fxy=expression(6*exp(-x^2-y^2)),
txy=expression(exp(-(6*exp(-x^2-y^2))^2))){
#creation of the compact set DD (aa,bb)x(cc,dd)
#Step 1: generation of N points in DD (uniformly distributed)
u12<-runif(2*N);i<-1:N; i1<-2*i; i2<-2*i - 1;
xi<-(u12[i1]*(bb-aa))+aa;
yi<-(u12[i2]*(dd-cc))+cc;
DD<-data.frame(xi,yi)
names(DD)<-c("x","y")

#Step 2: assignement of a uniform random number (0,1) to each
#random point generated in DD
w<-runif(N)

#Step 3: consider the function m2(x,y)=t(x,y).(1+.....) defined on D.
#Compute M2=maxD(m2(x,y))
dfdx<-D(fxy,"x"); dfdy<-D(fxy,"y");
m2<-eval(txy,envir=DD)*(1+eval(dfdx,envir=DD)^2 + eval(dfdy,envir
=DD)^2)^0.5
M2<-max(m2)

#Step 4: select the point (xi,yi,f(xi,yi)) in the final sample of
#random points on S if wi < m2(x,y)/M2
i3<-numeric(N)
for(i in 1:N){
if(w[i]<m2[i]/M2) i3[i]<-1 else i3[i]<-0
}

f<-eval(fxy,envir=DD);frange<-range(f)
DD<-data.frame(DD,f);DD<-DD[i3==1, ];

#three-dimensional plot of the selected simulated points on the surface
require(scatterplot3d)
scatterplot3d(DD,highlight.3d=TRUE,pch=20,xlim=c(aa,bb),ylim=
c(cc,dd),
zlim=frange,main=fxy,sub=paste("Selected points: ",nrow(DD),
```

[1] Après avoir chargé la librairie `scatterplot3d` avec la commande `require(scatterplot3d)` ou `library(scatterplot3d)`.

```
"over ",N))
list(DD=DD,selected=nrow(DD), prop.selected=nrow(DD)/N)
}
```

B.2.2 Explications pour l'utilisation de la fonction `nurda()`

La fonction `nurda()` présentée ci-dessus fonctionne de la même manière que `urda()`, sauf pour la présence d'un nouvel argument `txy`, qui correspond à la fonction $t(x, y)$. On rappelle que la fonction $t(x, y)$ représente un modèle de densité non-uniforme, à différence de comme c'était le cas pour l'algorithme *URDA*.

La fonction $t(x, y)$ doit être définie explicitement comme un objet de type `expression`. La valeur de défaut est :

```
txy=expression(exp(-(6*exp(-x^2-y^2))^2))
```

et correspond à la fonction :

$$t(x, y) = e^{-f(x,y)^2} = e^{-(6\exp(-x^2-y^2))^2}$$

La commande `nurda()` invoque la fonction avec les arguments de défaut :

- $D = (-3, 3) \times (-3, 3)$
- $f(x, y) = 6e^{-(x^2+y^2)}$
- $t(x, y) = e^{-f(x,y)^2} = e^{-(6\exp(-x^2-y^2))^2}$
- $N = 1000$

B.2.3 Exemples d'utilisation de la fonction `nurda()`

Les cinq premiers arguments (`N, aa, bb, cc, dd, fxy`) étant les mêmes que pour l'algorithme *URDA*, on va montrer uniquement comment modifier le nouvel argument `txy`.

Si on veut utiliser une fonction $t(x, y) = 3-|3-f(x, y)| = 3-|3-6e^{-(x^2+y^2)}|$, il faut la définir comme objet de type `expression`. La commande :

```
nurda(txy=expression(3-abs(3-6*exp(-x^2-y^2))))
```

simule 1000 points dans $D = (-3, 3)\times(-3, 3)$ et en choisis une partie. Ces points seront distribués selon le modèle de densité que cette fonction $t(x, y)$ représente, sur la surface définie par la fonction f de défaut $f(x, y) = 6e^{-(x^2+y^2)}$. Le modèle de densité $t(x, y) = 3-|3-f(x, y)|$ donne une probabilité de sélection supérieure aux points pour lesquels $f(x, y) = 3$
Le résultat graphique est le même que celui de la figure 3.3, page 18.

B.3 NIRDA, version 1

B.3.1 Code de la fonction `nirda()`

```
#temporary function ti
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^2
}

#automated function for algorithm 3 (NIRDA)
nirda<-function(N=10, aa=-3, bb=3, cc=-3, dd=3, fxy=expression
(exp(-x^2-y^2))){

dfdx<-D(fxy,"x");dfdy<-D(fxy,"y");
#temporary grid (simulated to evaluate maximum of m3i)
u12<-runif(2*N);
i<-1:N; i1<-2*i; i2<-2*i - 1;
x<-(u12[i1]*(bb-aa))+aa;
y<-(u12[i2]*(dd-cc))+cc;
grid.sim<-data.frame(x,y);rm(x,y);
m1.temp<-(1+eval(dfdx,envir=grid.sim)^2+eval(dfdy,envir=
grid.sim)^2)^0.5


#vector which will contain selected points (x_i,y_i)
xi<-numeric(N);yi<-numeric(N);

#Step 0: initialize t_1(x,y)=1 and i=1
t1<-1; i<-1;

j<-0;
while(i<=N){
m3i<-1;M3i<-1;w<-1;
while(w>=(m3i/M3i)){
u1<-runif(1);u2<-runif(1);
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)

w<-runif(1)

if(length(t1)==1){
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
}else{
m3i<-ti(x=x,y=y,xi=xi[i-1],yi=yi[i-1])*(1+eval(dfdx,envir
=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=
yi[i-1])*m1.temp
M3i<-max(m3i.temp)
```

```
}
print(c(m3i/M3i,w, w<(m3i/M3i)))
j<-j+1
}
t1<-c(t1,1)
xi[i]<-x;yi[i]<-y;
i<-i+1
}
selected<-data.frame(xi,yi)
names(selected)<-c("x","y")
f<-eval(fxy,envir=selected)
f.grid<-eval(fxy,envir=grid.sim)
frange<-range(f.grid)
frange.low<-floor(frange[1])
frange.high<-ceiling(frange[2])
frange<-c(frange.low,frange.high)
S<-data.frame(selected,f)
require(scatterplot3d)
par(mfrow=c(1,2), pty="s")
scatterplot3d(S,highlight.3d=TRUE,pch=20,xlim=c(aa,bb),
ylim=c(cc,dd),zlim=frange,main=fxy,sub=paste("N = ", N))
scatterplot3d(S,highlight.3d=TRUE,pch=20,xlim=c(aa,bb),
ylim=c(cc,dd),zlim=frange,main="Algorithm NIRDA, version 1",
sub=paste("Total iterations: ", j), type="b")
par(mfrow=c(1,1))
invisible(S)
}
```

B.3.2 Explications pour l'utilisation de la fonction `nirda()`

Avant d'utiliser la fonction `nirda()`, il faut créer séparément une fonction t_i qui définit la non-indépendance entre un point et le dernier sélectionné. La fonction t_i doit être définie comme objet de type `function` et doit prendre quatre arguments :

- `x` et `y` : les coordonnées des points simulés. Ils peuvent être soit sous forme de scalaire, soit sous forme de vecteurs de coordonnées ;
- `xi` et `yi` : les coordonnées des points sélectionnés. Au début le vecteur est vide de longeur N et au fur et à mesure qu'un nouveau point est sélectionné, les deux vecteurs se "remplissent".

Par exemple, la fonction :

$$t_i(x,y) = \begin{cases} 1 & \text{pour } i = 1; \\ \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^2 & \text{pour } i = 2,\ldots,N. \end{cases}$$

doit être définie dans `R` comme suit :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^2
}
```

La fonction `ti` qu'on définit séparément sera utilisée à l'intérieur de la fonction `nirda()`.

La fonction `nirda()`, dont le code a été présenté ci-dessus, prend les arguments suivants :

- `N` : le nombre de points non-indépendants qu'on veut simuler sur la surface S ;
- `aa, bb, cc, dd` : les bornes $(a, b) \times (c, d)$ de l'ensemble compact $D \subset \mathbb{R}^2$, comme pour les fonctions `urda()` et `nurda()` présentées avant ;
- `fxy` : un objet de type `expression` qui définit la fonction $f(x, y)$, comme pour les fonctions `urda()` et `nurda()` présentées avant.

Les valeurs de défaut que prend la fonction sont :

```
N=10
aa=-3, bb=3, cc=-3, dd=3
fxy=expression(exp(-x^2-y^2))
```

B.3.3 Exemples d'utilisation de la fonction `nirda()`

En utilisant la fonction `nirda()` sans arguments, l'algorithme *NIRDA*, version 1, est executé avec les valeurs de défaut. On peut changer les arguments comme pour les deux algorithmes précédents. Par exemple, si on veut simuler $N = 50$ points sur le plan $f(x, y) = 1$ dans le domaine $D = (-1, 1) \times (-1, 1)$, avec la fonction $t_i = \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^4$, il faut d'abord définir la fonction `ti` :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^4
}
```

et ensuite utiliser la fonction `nirda()` de la façon suivante :

```
nirda(N=50,aa=-1,bb=1,cc=-1,dd=1,fxy=expression(1))
```

La fonction crée deux graphiques des points non-indépendants simulés sur la surface S en question. Le deuxième graphique relie les points séquentiellement par des lignes. On peut voir le résultat de deux simulations dans les figures B.3 et B.4, à la page 47.

Si on veut effectuer une simulation de 50 points sur la surface définie par $f(x, y) = \exp(-x^2 - y^2)$ (fonction `fxy` de défaut), dans le domaine $D = (-3, 3) \times (-3, 3)$ avec la fonction $t_i = \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^{10}$, il faut utiliser les commandes :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^10
}

nirda(N=50)
```

On peut voir le résultat de deux simulations dans les figures B.5 et B.6 à la page 48.

B.4 NIRDA, version 2

B.4.1 Code de la fonction `nirda2()`

```
#automated function for algorithm 3 (NIRDA)
nirda2<-function(N=10, aa=-3, bb=3, cc=-3, dd=3, fxy=
expression(exp(-x^2-y^2))){

dfdx<-D(fxy,"x");dfdy<-D(fxy,"y");
#temporary grid (simulated to evaluate maximum of m3i)
u12<-runif(2*1000);
i<-1:1000; i1<-2*i; i2<-2*i - 1;
x<-(u12[i1]*(bb-aa))+aa;
y<-(u12[i2]*(dd-cc))+cc;
grid.sim<-data.frame(x,y);rm(x,y);
m1.temp<-(1+eval(dfdx,envir=grid.sim)^2+eval(dfdy,envir=grid.sim)^2)^0.5


#vector which will contain selected points (x_i,y_i)
xi<-numeric(N);yi<-numeric(N);
t.old<-rep(1,1000);
#Step 0: initialize t_1(x,y)=1 and i=1
t1<-1; i<-1;ti.old<-1;

#j is created to count total iterations performed
j<-0;
while(i<=N){
m3i<-1;M3i<-1;w<-1;
if(length(t1)!=1){
t.old<-t.old+ti(x=grid.sim$x,y=grid.sim$y,xi=xi[i-1],yi=yi[i-1])
m3i.temp<-t.old*m1.temp
M3i<-max(m3i.temp)
}

while(w>=(m3i/M3i)){
u1<-runif(1);u2<-runif(1);
x<-u1*(bb-aa)+aa; y<-u2*(dd-cc)+cc;
xy<-data.frame(x,y)

w<-runif(1)

if(length(t1)==1){
m3i<-t1*(1+eval(dfdx,envir=xy)^2+eval(dfdy,envir=xy)^2)^0.5
m3i.temp<-t1*m1.temp
M3i<-max(m3i.temp)
}else{
ti.old<-1
for(k in 1:(i-1)){
ti.old<-ti.old+ti(x=x,y=y,xi=xi[i-k],yi=yi[i-k])
}
```

```
m3i<-ti.old*(1+eval(dfdx,envir=xy)^2
+eval(dfdy,envir=xy)^2)^0.5

print(c(m3i/M3i,w, w<m3i/M3i))
j<-j+1
}
}
t1<-c(t1,1)
xi[i]<-x;yi[i]<-y;
i<-i+1
}
selected<-data.frame(xi,yi)
names(selected)<-c("x","y")
f<-eval(fxy,envir=selected)
frange<-range(f)
S<-data.frame(selected,f)
require(scatterplot3d)
scatterplot3d(S,highlight.3d=TRUE,pch=20,xlim=c(aa,bb),ylim=c(cc,dd),
zlim=frange,main=fxy,sub=paste("N = ", N))
par(mfrow=c(1,1))
invisible(S)
}
```

B.4.2 Explications pour l'utilisation de la fonction `nirda2()`

Comme pour la version 1, il faut d'abord définir une fonction `ti` :

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^2
}
```

Une fois que la fonction `ti` a été définie, la fonction `nirda2()` peut être invoquée exactement de la même manière de la version 1.

B.4.3 Exemples d'utilisation de la fonction `nirda2()`

La commande `nirda2()` exécute l'algorithme *NIRDA*, version 2, avec les arguments de défaut, c'est-à-dire :

- $N = 10$;
- $D = (-3, 3) \times (-3, 3)$;
- $f(x, y) = e^{-(x^2+y^2)}$

Le résultat de deux simulations avec $N = 50$ points et les autres arguments de défaut est présenté dans les figures B.7 et B.8 à la page 49[2].

Dans ce cas aussi on peut effectuer la simulation sur le plan $f(x, y) = 1$ dans le domaine $D = (-1, 1) \times (-1, 1)$, avec la fonction t_i qui se met à jour de la façon suivante :

$$t_i = t_{i-1} + \left(e^{-[(x-x_{i-1})^2+(y-y_{i-1})^2]}\right)^4$$

Les commandes :

[2]Les graphiques ont été obtenus avec la commande `nirda2(N=50)`.

```
ti<-function(x,y,xi,yi){
ti<-exp(-((x-xi)^2+(y-yi)^2))^4
}

nirda2(N=50, aa=-1,bb=1,cc=-1,dd=1, fxy=expression(1))
```

exécutées deux fois, produisent les graphiques des figures B.9 et B.10 à la page 50.

FIG. B.3 – Résultat de l'algorithme *NIRDA, version 1* avec $f(x, y) = 1$ dans $D = (-1, 1) \times (-1, 1)$

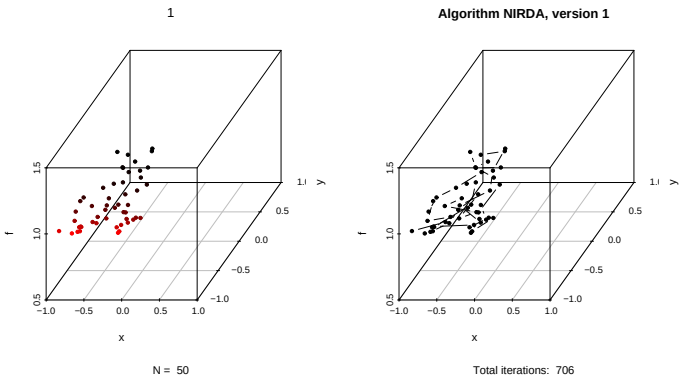


FIG. B.4 – Résultat d'une autre simulation *NIRDA, version 1* avec $f(x, y) = 1$ dans $D = (-1, 1) \times (-1, 1)$

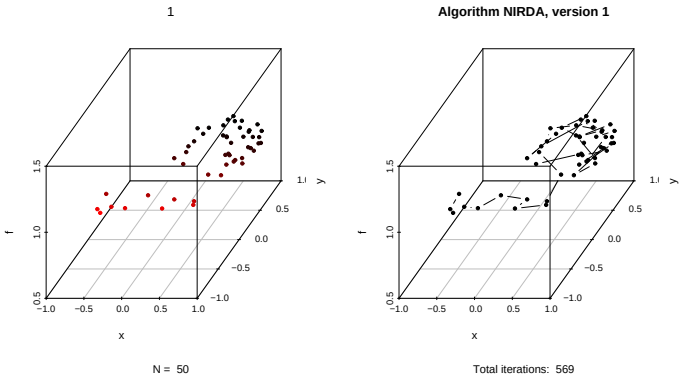


FIG. B.5 – Résultat de l'algorithme *NIRDA, version 1* avec $f(x, y) = \exp(-x^2 - y^2)$ dans $D = (-3, 3) \times (-3, 3)$

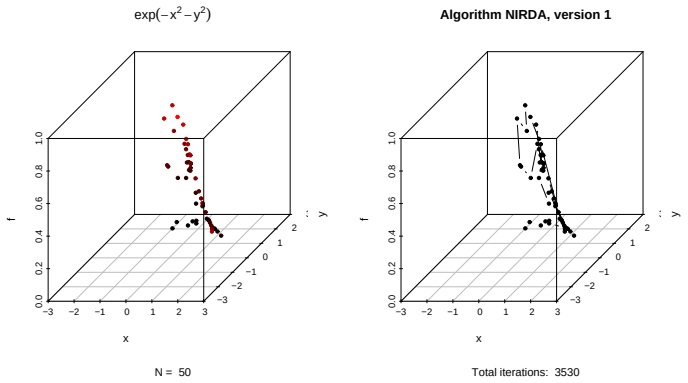


FIG. B.6 – Résultat d'une autre simulation *NIRDA, version 1* avec $f(x, y) = \exp(-x^2 - y^2)$ dans $D = (-3, 3) \times (-3, 3)$

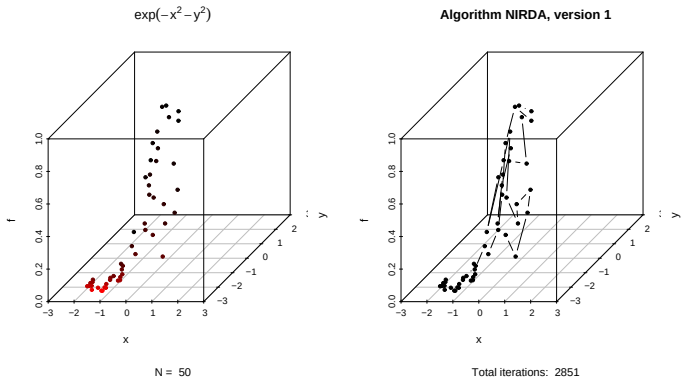


FIG. B.7 – Résultat de l'algorithme *NIRDA, version 2* avec $f(x, y) = \exp(-x^2 - y^2)$ dans $D = (-3, 3) \times (-3, 3)$

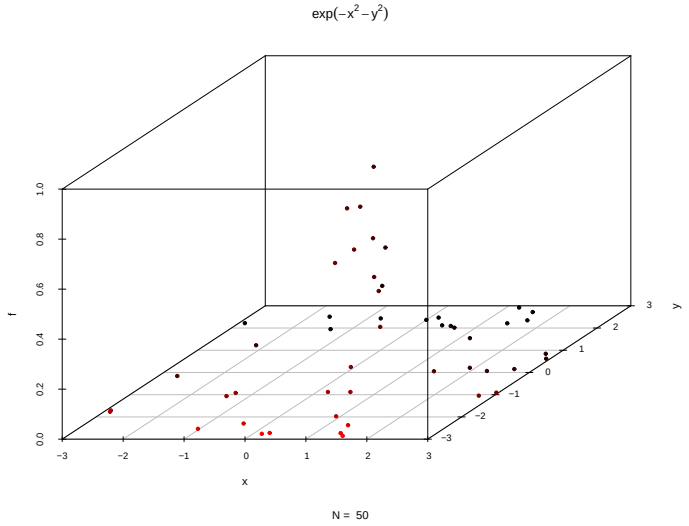


FIG. B.8 – Résultat d'une autre simulation *NIRDA, version 2* avec $f(x, y) = \exp(-x^2 - y^2)$ dans $D = (-3, 3) \times (-3, 3)$

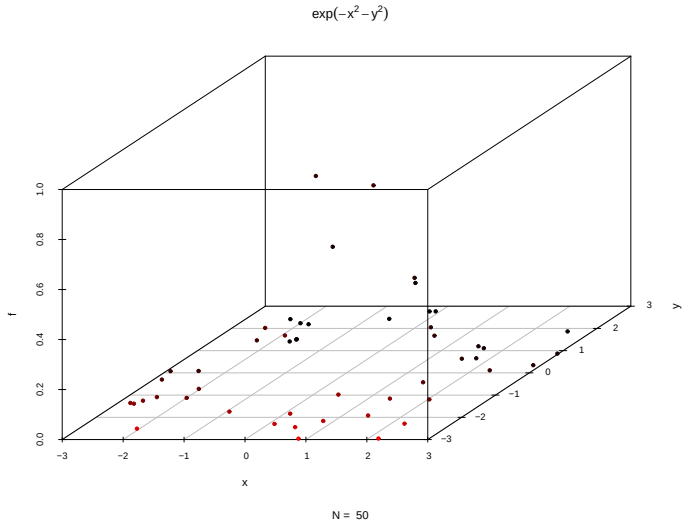


FIG. B.9 – Résultat de l'algorithme *NIRDA, version 2* avec $f(x, y) = 1$ dans $D = (-1, 1) \times (-1, 1)$

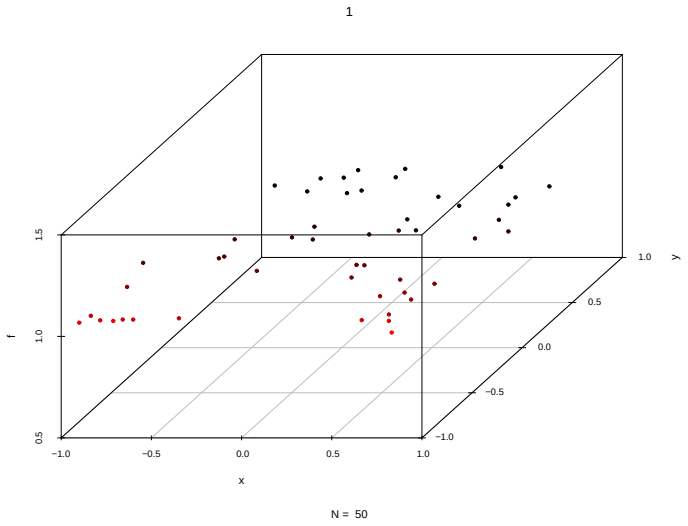
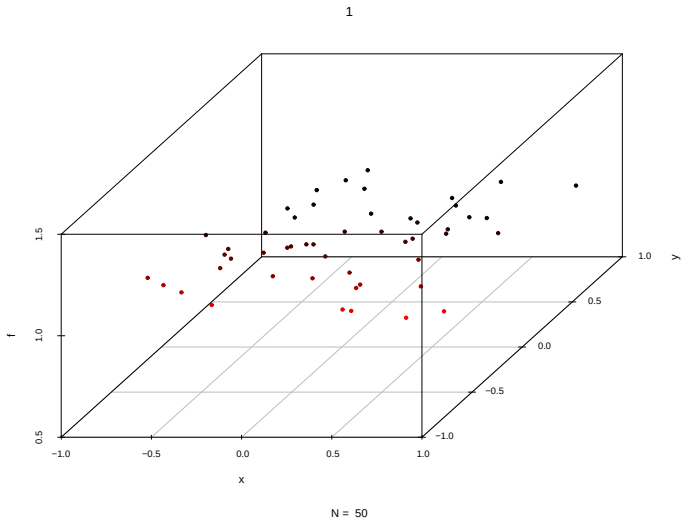


FIG. B.10 – Résultat d'une autre simulation *NIRDA, version 2* avec $f(x, y) = 1$ dans $D = (-1, 1) \times (-1, 1)$



Bibliographie

[1] Wilfred Kaplan. *Advanced Calculus.* Addison-Wesley, 1959.

[2] Donald Knuth. *Seminumerical Algorithms*, volume 2 of *The art of computer programming.* Addison-Wesley, Reading, Massachussets, 1981. ISBN 0-201-89684-2.

[3] Uwe Ligges and Martin Mächler. Scatterplot3d : An R package for visualizing multivariate data. *Journal of Statistical Software*, 8(11) :1–20, 2003.

[4] Giuseppe Melfi and Gabriella Schoier. Simulation of random distributions on surfaces. pages 173–176, Bari, 2004. Società Italiana di Statistica (SIS), Atti della XLII Riunione Scientifica.

[5] R Development Core Team. *R : A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria, 2004. ISBN 3-900051-07-0.

[6] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods.* Springer Texts in Statistics, New York, 1999. ISBN 0-387-98707-X.

[7] Deepayan Sarkar. *Lattice : Lattice Graphics*, 2004. R package version 0.10-16.

[8] Alessandra Zacchigna. Analisi statistica spaziale : Algoritmi di simulazione di punti su superfici, Università degli Studi di Trieste, Facoltà di economia, Tesi di Laurea in Statistica ed Informatica per l'azienda, Anno accademico 2003-2004.