Université Gustave Eiffel

SIMULATION ET COPULES

PROJET 1

Marie MEYER et Shuaib MOHUN • *4 Novembre 2024*

INTRODUCTION

Dans ce projet, nous allons explorer l'estimation de l'intégrale de la fonction $e^{\sin(x \cdot y)}$ à l'aide de techniques de simulation.

L'objectif est de comprendre comment appliquer des méthodes numériques pour approximativement évaluer cette intégrale sur un domaine donné.

Chargement des bibliothèques nécessaires

Avant de commencer, nous devons charger les bibliothèques sur R qui faciliteront nos calculs :

```
library(rgl)
library(pracma)  # Pour les integrations numeriques
library(randtoolbox)   # Pour generer des points de Halton
```

Estimation de l'intégrale de $e^{(x \cdot y)}$

Nous cherchons à estimer l'intégrale de la fonction $e^{\sin(x \cdot y)}$ sur un domaine donné.

> *Remarque 1 :* Le domaine d'intégration, noté Ω, est donné par :

$$\Omega = \{(x, y) \mid a \leqslant x \leqslant b,\ c \leqslant y \leqslant d\}$$

soit :

$$\Omega = \{(x, y) \mid 0 \leqslant x \leqslant 0{,}5,\ 2 \leqslant y \leqslant 3\}$$

Plus précisément, nous souhaitons calculer :

$$I = \int_a^b \int_c^d e^{\sin(x \cdot y)}\, dy\, dx$$

Pour ce faire, nous avons dans un premier temps réalisé le code R suivant :

```
# Definition des limites des variables
# Nous avons etabli les bornes de l'integration afin de delimiter le rectangle sur le plan (x,y)
# dans lequel nous allons evaluer la fonction
a <- 0    # xmin
b <- 0.5  # xmax
c <- 2    # ymin
d <- 3    # ymax

# Nombre de simulations
# Nous avons fixe le nombre total de points a simuler a 1000, ainsi que le nombre d'iterations
# a 500 pour chaque methode d'estimation
n_simulations <- 1000  # Total de points a simuler
n_iterations <- 500      # Nombre d'iterations pour chaque methode

# Fonction a integrer
# On definie notre fonction, ce qui permet de calculer ses valeurs pour chaque paire (x,y)
# du domaine d'integration.
f_xy <- function(x, y) {
  exp(sin(x * y))  # Calcul de f(x, y)
```

```
}

# Creation de la grille de points pour le graphique 3D
valeurs_x <- seq(a, b, length.out = 100)
valeurs_y <- seq(c, d, length.out = 100)

# Calcul des valeurs Z en utilisant outer pour appliquer f_xy sur chaque
# Combinaison de x et y
# Grace a la fonction outer, nous avons applique f(x,y) sur chaque combinaison de valeurs x et y
# de la grille creee. Ca a abouti a une matrice de valeurs Z, representant les resultats de la
# fonction pour le domaine considere.
valeurs_z <- outer(valeurs_x, valeurs_y, f_xy)

# Affichage du graphique 3D
# Nous avons enfin utilise la fonction persp3d pour creer le graphique 3d, permettant d'illuster
# la forme de la fonction sur le domaine d'integration
persp3d(valeurs_x, valeurs_y, valeurs_z, col = "lightblue",
xlab = "X", ylab = "Y", zlab = "f(x, y)",
phi = 30, theta = 30,
lit = TRUE, axes = TRUE)
```

Remarque 2 : Plus il y a de simulations, plus l'estimation sera précise.
Nous avons donc décidé de fixer le nombre total de points à simuler à 1000 pour l'ensemble des méthodes dans un premier temps. Nous avons testé ensuite avec 10 000 simulatoins à la Figure 12.

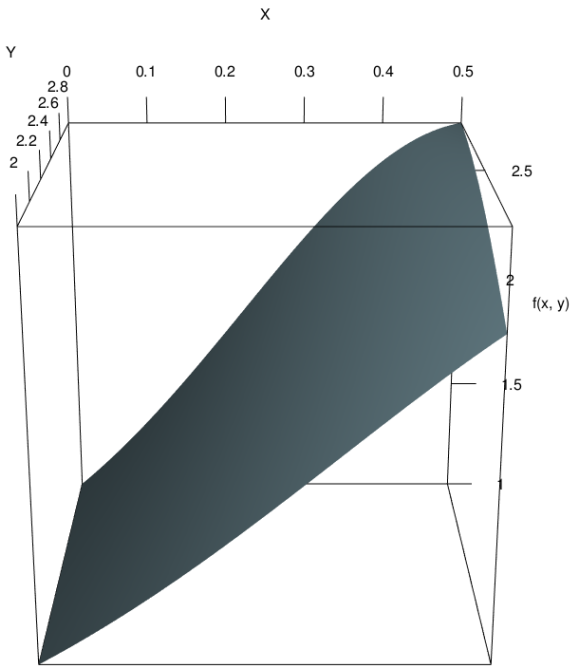


Figura 1: Représentation 3d

La **Figure 1** obtenue est une représentation tridimensionnelle de la fonction $e^{\sin(x \cdot y)}$ sur l'intervalle $[0, 0{,}5]$ pour x et $[2, 3]$ pour y.
L'axe X correspond aux valeurs de la variable x dans l'intervalle $[0, 0{,}5]$. L'axe Y correspond aux valeurs de la variable y dans l'intervalle $[2, 3]$. L'axe Z ($f(x, y)$) représente les valeurs de la fonction $f(x, y)$.

La surface visible représente les valeurs de la fonction $e^{\sin(x \cdot y)}$ pour chaque combinaison des valeurs de x et y.

On observe que la surface est courbée, ce qui reflète la variation non linéaire de la fonction en fonction des deux variables x et y.

Lorsque x et y augmentent, la valeur de la fonction change de manière non linéaire. C'est dû en effet à la nature oscillatoire de la fonction sinus, appliquée à $x \cdot y$, puis exponentiée, ce qui amplifie les variations de $\sin(x \cdot y)$. La surface apparaît plus inclinée et plus élevée pour des valeurs de y plus grandes et des x proches de 0,5, ce qui s'explique par l'augmentation de $x \cdot y$, entraînant une plus grande valeur de $\sin(x \cdot y)$, et donc une augmentation de $e^{\sin(x \cdot y)}$.

La courbe montre des pics dans certaines régions où $\sin(x \cdot y)$ atteint des valeurs élevées, et des creux là où $\sin(x \cdot y)$ est plus faible.

MÉTHODE 1 : MÉTHODE DE VOLUME

La première méthode effectué est la **Méthode de Volume** qui consiste à estimer le volume sous la surface définie par la fonction, dans un espace tridimensionnel défini par les limites d'intégration pour x et y.

Le code réalisé pour cette méthode est le suivant :

```
# Definition des limites pour la hauteur (z)
# Nous avons determine les limites pour la variable de hauteur z.
z_min <- 0  # Valeur minimale pour z fixe a 0
z_max <- exp(sin(b * d))  # Valeur maximale pour z

# Tirage de points dans l'espace de volume
# Nous avons genere des echantillons aleatoires pour la hauteur z en utilisant la fonction runif
# ce qui a permis de tirer des points uniformement distribues dans l'intervalle [z_min, z_max]
# Les echantillons ont ete organises sous forme de matrice pour faciliter les calculs ulterieurs.
z_ech <- matrix(runif(n_simulations * n_iterations, z_min, z_max), ncol = n_iterations)

# Tirage de points pour x et y
# Des echantillons pour les variables x et y ont egalement ete tires de maniere uniforme dans
# leurs intervalles respectifs [a,b] et [c,d], ce qui a permis de couvir l'ensemble du domaine
# d'integration.
x_ech <- matrix(runif(n_simulations * n_iterations, a, b), ncol = n_iterations)
y_ech <- matrix(runif(n_simulations * n_iterations, c, d), ncol = n_iterations)

# Calculer f(x, y) pour les echantillons tires
# Pour chaque paire d'echantillons (x,y), nous avons evalue la fonction f(x,y) afin de determiner
# la valeur de la surface a chaque point tire.
z_surface <- f_xy(x_ech, y_ech)

# Verifier combien de points sont sous la surface
# Nous avons ensuite verifie combien de points (x,y,z) se trouvaient sous la surface definie
# par f(x,y). Cette verification a ete effectuee en comparant les valeurs de z  echantillonnees
# avec les valeurs de z_surface
points_en_dessous_surface <- z_ech <= z_surface

# Approximation de l'integrale par la methode du volume
facteur_volume <- (b - a) * (d - c) * z_max
Tn_evol <- cumsum(points_en_dessous_surface[, 1]) / (1:n_simulations) * facteur_volume

# Calcul de Tn pour toutes les simulations
# Nous avons enfin defini une fonction qui calcule la moyenne des points sous la surface,
# multipliee par le facteur de volume, afin d'obtenir une estimation de l'integrale pour
# chaque iteration.
calculer_estimation_volume <- function(points) {
  mean(points) * facteur_volume
}

# Les resultats finaux ont ete stokes dans un vecteur pour une analyse ulterieure
Tn_vecteur <- apply(points_en_dessous_surface, MARGIN = 2, FUN = calculer_estimation_volume)
```

Dans un premier temps, les limites pour la hauteur z sont définies : la valeur minimale z_{min} est fixée à 0, correspondant au plan $z = 0$ et la valeur maximale z_{max} est calculée à l'aide de la fonction exponentielle appliquée à $\sin(x_{max} \cdot y_{max})$.
Ensuite, des points aléatoires sont tirés dans l'espace de volume. Cela inclut la génération d'une matrice contenant des valeurs aléatoires pour z, uniformément réparties entre z_{min} et z_{max}, ainsi

que pour x et y, qui sont également générés de manière uniforme dans leurs intervalles respectifs.
Puis, la fonction `f_xy` est appliquée aux paires de valeurs x et y pour calculer les valeurs de z sur la surface. Pour évaluer combien de points se trouvent sous cette surface, une matrice est créée, indiquant si chaque point (x, y, z) est situé en dessous de `z_surface`.

L'intégrale T_n est approximée en utilisant la formule suivante :

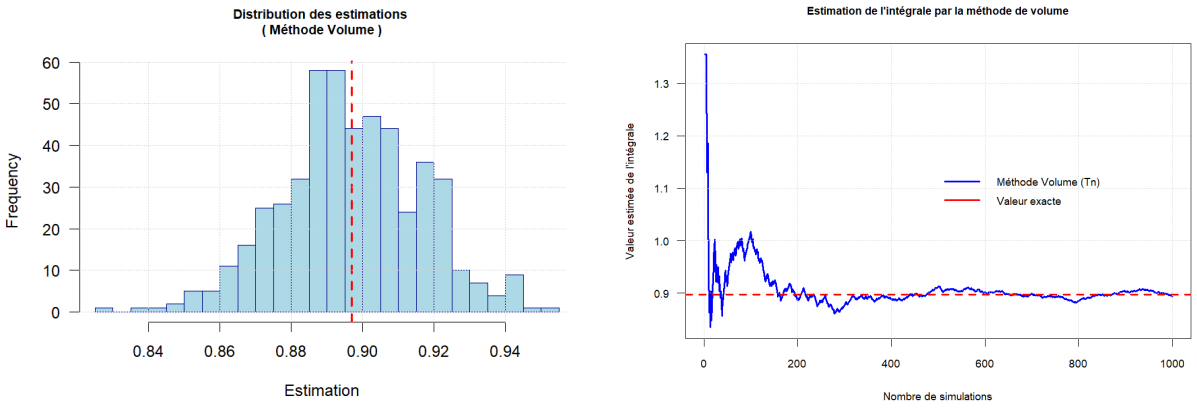
$$\frac{1}{n} \sum \mathbf{1}_{\{\varphi(U_i, V_i) \geqslant W_i\}} \to \frac{I}{[a, b] \times [c, d] \times K}$$

avec :

- $U_i \to \bigcup_{[a,b]}$, correspondant à x_ech, qui sont les échantillons tirés pour x dans l'intervalle $[a, b]$.
- $V_i \to \bigcup_{[c,d]}$, correspondant à y_ech, qui sont les échantillons tirés pour y dans l'intervalle $[c, d]$.
- $W_i \to \bigcup_{[0,K]}$, correspondant à z_ech, qui sont les échantillons tirés pour z dans l'intervalle $[0, z\_max]$.
- $K = \sup_{(x,y)\in[a,b]\times[c,d]} \varphi(x, y)$, correspondant à z_max qui est donc défini comme la valeur maximale de la fonction $\varphi(x, y) = e^{\sin(x \cdot y)}$ sur le domaine $[a, b] \times [c, d]$.

Pour finir, une fonction est définie pour estimer le volume moyen des points sous la surface, et cette fonction est appliquée à chaque itération pour produire un vecteur de résultats `Tn_vecteur`.

Nous obtenons les résultats suivant :



(a) Distribution des estimations

(b) Estimation de l'intégrale

Figura 2: Résultats Méthode Volume

L'Histogramme représente la distribution des différentes valeurs obtenues lors de l'estimation de l'intégrale et nous donne une visualisation de la qualité des estimations obtenues. La distribution semble centrée autour de la valeur exacte de l'intégrale (en rouge). Cela suggère que la plupart des estimations sont proches de cette valeur et qu'il y a de moins en moins d'estimations au fur et à mesure que l'on s'éloigne de celle-ci.
La forme relativement étroite et centrée de l'histogramme suggère que la méthode du volume

est relativement précise. Le graphique de droite nous donnera plus de précision à ce sujet.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe bleue . On observe une certaine instabilité au début, puis les estimations semblent se stabiliser autour de la valeur exacte (ça converge). On remarque que même avec un grand nombre de simulations, il subsiste une certaine dispersion des estimations.

MÉTHODE 2 : MÉTHODE DE MONTE CARLO BASIQUE

La deuxième méthode effectué est la **Méthode de Monte Carlo Basique** qui consiste à tirer un certain nombre d'échantillons aléatoires dans le domaine d'intégration et de calculer la moyenne des valeurs de la fonction à ces points.

Le code réalisé pour cette méthode est le suivant :

```
# Calcul de f(x, y) pour chaque echantillon tire
# valeurs_z_mc est une matrice ou chaque element correspond a la valeur de la fonction pour un
# echantillon specifique.
valeurs_z_mc <- f_xy(x_ech, y_ech)

# Approximation de l'integrale par Monte Carlo basique
# Nous calculons l'aire du rectangle delimite par les limites d'integration pour x et y.
# L'aire est calculee en multipliant la largeur par la hauteur
aire_rectangle <- (b - a) * (d - c)
# On calcule l'evolution de l'estimation de l'integrale a chaque iteration.
Sn_evol <- cumsum(valeurs_z_mc[, 1]) / (1:n_simulations) * aire_rectangle

# Calcul de Sn pour toutes les simulations
# Nous definissons une fonction qui calcule l'estimation de l'integrale pour un ensemble donne
# de valeurs.
calculer_estimation_mc <- function(valeurs) {
  mean(valeurs) * aire_rectangle
}

# On applique la fonction calculer_estimation_mc a chaque colonne de la matrice valeurs_z_mc
Sn_vecteur <- apply(valeurs_z_mc, MARGIN = 2, FUN = calculer_estimation_mc)
```

On génère dans un premier temps aléatoirement des points (x, y) dans le domaine d'intégration, comme dans la méthode du volume.
Ensuite, pour chaque point (x, y) généré, on calcule la valeur de la fonction $f(x, y)$ en ce point. La fonction $f_{xy}(x_{ech}, y_{ech})$ est utilisée pour évaluer les valeurs de z pour chaque paire d'échantillons tirés aléatoirement des variables x et y. Les résultats sont stockés dans `valeurs_z_mc`.
L'aire du rectangle englobant la zone d'intégration est ensuite calculée à partir des limites maximales et minimales des variables x et y.

L'intégrale est alors approximée en utilisant la formule suivante :

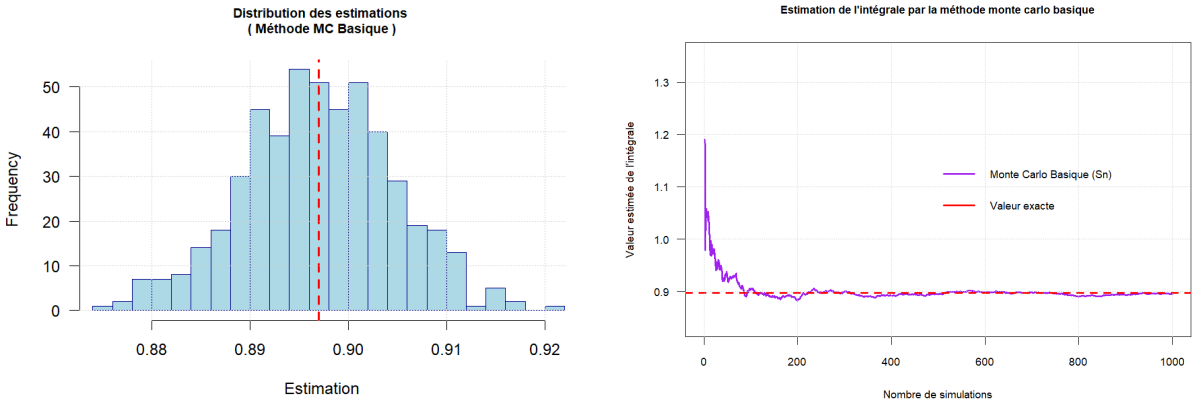
$$\frac{1}{n} \sum \varphi(U_i, V_i) \cdot (b-a)(d-c) \cdot \mathbf{1}_{[a,b]\times[c,d]}(U_i, V_i) \to \mathbb{E}[\varphi(U_i, V_i)]$$

Les résultats sont stockés dans Sn_evol.

Enfin, une fonction $calculer\_estimation\_mc$ est définie pour calculer l'estimation de l'intégrale en prenant la moyenne des valeurs de z pour chaque simulation, également multipliée par l'aire

du rectangle. La fonction est appliquée à chaque colonne des résultats de $valeurs\_z\_mc$, et les estimations résultantes sont stockées dans $Sn\_vecteur$.

Nous obtenons les résultats suivant :



(a) Distribution des estimations

(b) Estimation de l'intégrale

Figura 3: Résultats Méthode Monte Carlo Basique

D'après l'Histogramme, la dispersion des estimations est relativement faible, ce qui indique une bonne précision. La distribution des estimations semble approximativement normale (en forme de cloche). La méthode converge car les estimations obtenues se rapprochent de la valeur exacte de l'intégrale lorsque le nombre de simulations augmente.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe violette . Comme pour la méthode de Volume, on observe une certaine instabilité au début, puis les estimations semblent se stabiliser autour de la valeur exacte. Cette méthode semble plus efficace que la méthode de Volume car l'estimation se stabilise plus autour de la valeur exacte de l'intégrale.

MÉTHODE 3 : MÉTHODE DE VARIANCE RÉDUITE

Cette troisième méthode nommé **Méthode de variance réduite** utilise une technique de variance réduite pour améliorer l'estimation de l'intégrale. L'objectif est de réduire la variance de l'estimation, ce qui permet d'obtenir une approximation plus précise de l'intégrale avec moins de simulations.

Nous avons effectué le code suivant :

```
# Definir plusieurs valeurs pour y : c (y_min), la mediane, et d (y_max)
# Creation d'un vecteur y_fixe contenant c (min), la mediane, et d (max)
y_fixe <- c(c, (c + d) / 2, d)

# Calculer les valeurs de f(x, y_fixe) pour chaque combinaison de x et y
# Creation d'une matrice ou chaque colonne correspond e f(x, y) pour une valeur de y fixee
f_x_yfixe <- sapply(y_fixe , function(y) sapply(valeurs_x, function(x) f_xy(x, y)))

# Tracer la courbe pour y = c (correspondant e y_min) avec une ligne noire pointillee
plot(valeurs_x, f_x_yfixe[, 1], type = "l", col = "black", lwd = 2, lty = 2,
     xlab = "X", ylab = "f(X, Y)",
     main = "Graphique de f(x, y) pour y : c, mediane, d",  # Titre plus clair
     ylim = c(1, 3))  # Definir les limites de l'axe Y de 1 a 3

# Ajouter la courbe pour y = d (correspondant e y_max) avec une ligne noire pointillee
lines(valeurs_x, f_x_yfixe[, 3], col = "black", lwd = 2, lty = 2)

# Colorier la region entre c (y_min) et d (y_max) en bleu transparent
polygon(c(valeurs_x, rev(valeurs_x)),
        c(f_x_yfixe[, 1], rev(f_x_yfixe[, 3])),
        col = rgb(0.1, 0.1, 0.8, 0.2), border = NA)  # Remplissage en bleu transparent

# Ajouter la courbe pour la valeur mediane de y avec une ligne rouge
lines(valeurs_x, f_x_yfixe[, 2], col = "red", lwd = 2)

# Calculer la fonction p1 a tracer en bleu
p1_valeurs <- 8 / 7 * (1 + 3 * valeurs_x)  # Definir p1 en fonction de x
lines(valeurs_x, p1_valeurs, col = "blue", lwd = 2)  # Tracer la ligne de p1 en bleu

# Ajouter une legende pour identifier les courbes
legend("bottomright",
       legend = c(paste("y =", c),  # Indiquer la valeur de c (y_min)
                  paste("y =", (c + d) / 2),  # Indiquer la valeur mediane
                  paste("y =", d),  # Indiquer la valeur de d (y_max)
                  "p1 = 8/7(1 + 3x)"),  # Indiquer la formule de p1
       col = c("black", "red", "black", "blue"),  # Couleurs pour la legende
       lwd = 2, lty = c(2, 1, 2, 1))  # Styles de lignes pour la legende
```

Le code commence par définir plusieurs valeurs pour la variable y en utilisant trois points : le minimum (c), la médiane ($(c + d)/2$), et le maximum (d). Ces trois valeurs forment un vecteur appelé y_fixe. L'objectif est d'étudier comment la fonction $f(x, y)$ évolue en fonction de x pour ces trois valeurs de y.

Nous obtenons le grahique 4 ci-dessous :

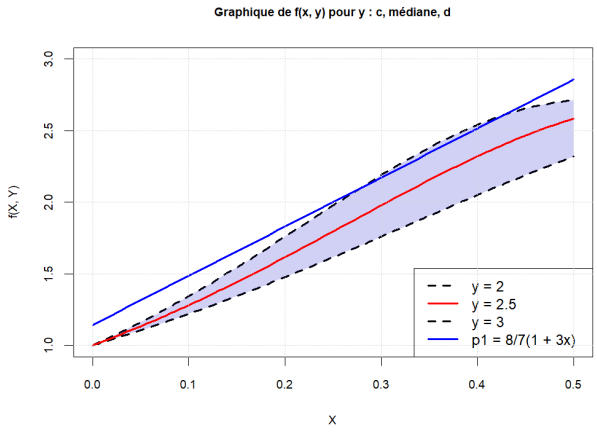


Figura 4: Evolution de f(x, y) en fonction de x pour trois valeurs spécifiques de y : y = c, y = (c + d) / 2 (médiane), et y = d

Le graphique montre l'évolution de $f(x, y)$ en fonction de x pour trois valeurs spécifiques de y :
$y = 2$, $y = 2{,}5$ et $y = 3$.

On voit que la fonction croît de manière similaire pour ces trois valeurs, mais les écarts entre elles se réduisent à mesure que x augmente. La région ombrée en bleu clair montre l'intervalle de variation de $f(x, y)$ entre les deux extrêmes $y = 2$ et $y = 3$, tandis que la droite bleue représente la fonction $p_1(x)$.

Pour obtenir le fonction de répartition et la fonction inverse, nous avons fait les calculs suivants :

- **Fonction de densité** $p_1(x)$:

$$p1(x) = \frac{8}{7}(1 + 3x)\,dx$$

- **Fonction de répartition** $F_1(t)$:

$$\int \frac{8}{7}(1+3t)\,dt = \frac{8}{7}\int (1+3t)\,dt = \frac{8}{7}\left(t + \frac{3}{2}t^2\right) + C = \left[\frac{8}{7}\left(t + \frac{3}{2}t^2\right)\right]_0^t = \frac{8}{7}\left(t + \frac{3}{2}t^2\right)$$

$$F_1(t) = \frac{4}{7}\left(2t + 3t^2\right)$$

- **Fonction Inverse** $F1\_inverse(z)$: Nous voulons trouver la fonction inverse $F_1^{-1}(z)$ telle que $F_1(t) = z$.

On a :

$$z = \frac{4}{7}(2t + 3t^2) <=> \frac{7}{4}z = 2t + 3t^2$$

Soit :

$$3t^2 + 2t - \frac{7}{4}z = 0$$

On applique la formule quadratique, où $a = 3$, $b = 2$, et $c = -\frac{7}{4}z$:

$$b^2 - 4ac = 2^2 - 4 \cdot 3 \cdot \left(-\frac{7}{4}z\right) = 4 + 21z$$

On trouve :

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-2 \pm \sqrt{4 + 21z}}{6}$$

Nous cherchons une valeur positive de t, donc nous choisissons la solution positive :

$$F_1^{-1}(z) = \frac{-1 + \sqrt{1 + \frac{21}{4}z}}{3}$$

Nous inversons la fonction de répartition pour générer des échantillons de X pour transformer des échantillons uniformes (qui suivent une distribution uniforme entre 0 et 1) en échantillons de la distribution p1.

Pour continuer, nous avons le code ci-dessous :

```r
# Definir p1, la densite de probabilite pour X
p1 <- function(x) 8 / 7 * (1 + 3 * x)

# Fonction de repartition de p1 (cumulee)
F1 <- function(t) {
  return(4 / 7 * (2 * t + 3 * t^2))  # Fonction F1(x) correspondant a la FdR de p1
}

# Inverse de la fonction de repartition F1 (permet de generer des echantillons de X selon p1)
F1_inverse <- function(z) {
  return((-1 + sqrt(1 + 21 / 4 * z)) / 3)  # Resolution analytique de l'inverse de F1
}

# Histogramme des valeurs generees par la fonction inverse de F1 avec une distribution uniforme
hist(F1_inverse(runif(10000)))
```

L'histogramme montre la distribution des valeurs générées pour X selon la densité $p_1(x)$. On observe une augmentation des fréquences lorsque X augmente, ce qui est conforme à la forme de la densité qui est croissante en x.

Les valeurs faibles de X (vers 0) apparaissent moins fréquemment, alors que les valeurs plus proches de 0,5 apparaissent plus fréquemment, indiquant une densité qui favorise les valeurs plus élevées de X dans cet intervalle. Ca correspond bien à la distribution attendue pour $p_1(x)$, où la probabilité est plus élevée pour les valeurs de x proches de 0,5.

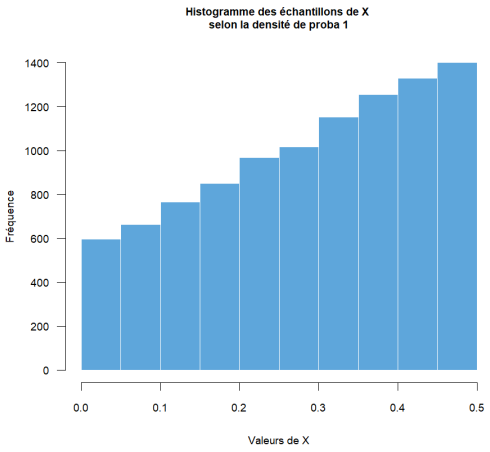


Figura 5: Histogramme des échantillons de X selon la densité de probabilité 1

Cet histogramme valide ainsi que l'inverse de la fonction de répartition $F_1^{-1}(z)$ est correctement utilisée pour échantillonner selon la densité p_1.

Enfin, le code se termine par :

```r
# Definir plusieurs valeurs pour x :
# a (correspondant a x_min), la mediane, et b (correspondant a x_max)
x_fixe <- c(a, (a + b) / 2, b)  # Creation d'un vecteur contenant a, la mediane, et b

# Calculer les valeurs de f(x_fixe, y) pour chaque combinaison de y et x
# Creation d'une matrice ou chaque colonne correspond a f(x, y) pour une valeur de x fixee
f_xfixe_y <- sapply(x_fixe, function(x) sapply(valeurs_y, function(y) f_xy(x, y)))

# Tracer la courbe pour x = a (correspondant a x_min) avec une ligne noire pointillee
plot(valeurs_y, f_xfixe_y[, 1], type = "l", col = "black", lwd = 2, lty = 2,
     xlab = "Y", ylab = "f(X, Y)",
     main = "Graphique de f(x, y) pour x : a, mediane, b",  # Titre plus clair
```

```
     ylim = c(1, 3))  # Definir les limites de l'axe Y de 1 a 3

# Ajouter la courbe pour x = b (correspondant a x_max) avec une ligne noire pointillee
lines(valeurs_y, f_xfixe_y[, 3], col = "black", lwd = 2, lty = 2)

# Colorier la region entre a (x_min) et b (x_max) en bleu transparent
polygon(c(valeurs_y, rev(valeurs_y)),
        c(f_xfixe_y[, 1], rev(f_xfixe_y[, 3])),
        col = rgb(0.1, 0.1, 0.8, 0.2), border = NA)  # Remplissage en bleu transparent

# Ajouter la courbe pour la valeur mediane de x avec une ligne rouge
lines(valeurs_y, f_xfixe_y[, 2], col = "red", lwd = 2)
# Ajouter une legende pour identifier les courbes
legend("bottomright", legend = c(paste("x =", a),  paste("x =", (a + b) / 2), paste("x =", b)),
col = c("black", "red", "black"), lwd = 2, lty = c(2, 1, 2))

# La visualisation avec une couleur bleue pour la region entre x=a et x=b, ainsi qu'une ligne
# rouge pour la mediane est efficace pour montrer les differences dans f(x,y) en fonction de y.
# Definir p2, la densite de probabilite uniforme pour Y
p2 <- function(y) 1  # Uniforme, donc constante entre c et d

# Definir p, la fonction de densite conjointe
p = function(x, y) p1(x) * p2(y)  # Densite conjointe p(x, y) = p1(x) * p2(y)

# Generer des echantillons pour X et Y
x_ech = F1_inverse(runif(n_simulations))  # Echantillons pour X selon p1 via F1_inverse
y_ech = runif(n_simulations, c, d)  # Echantillons pour Y selon une loi uniforme entre c et d

# Calculer l'evolution de la variance reduite (VR_evol)
# Moyenne cumulative ponderee par la fonction de densite p(x, y)
VR_evol = cumsum(f_xy(x_ech, y_ech) / p(x_ech, y_ech)) / (1:n_simulations)
```

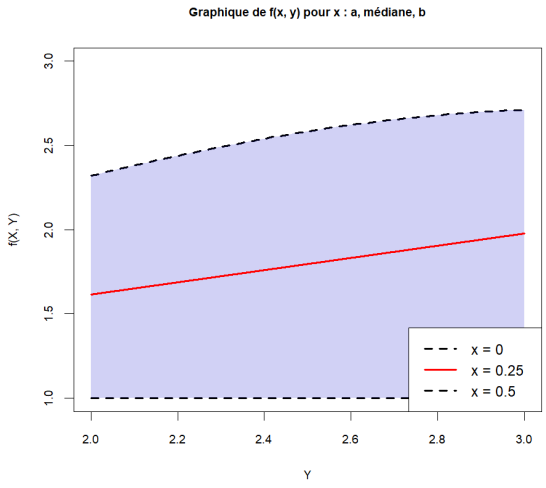
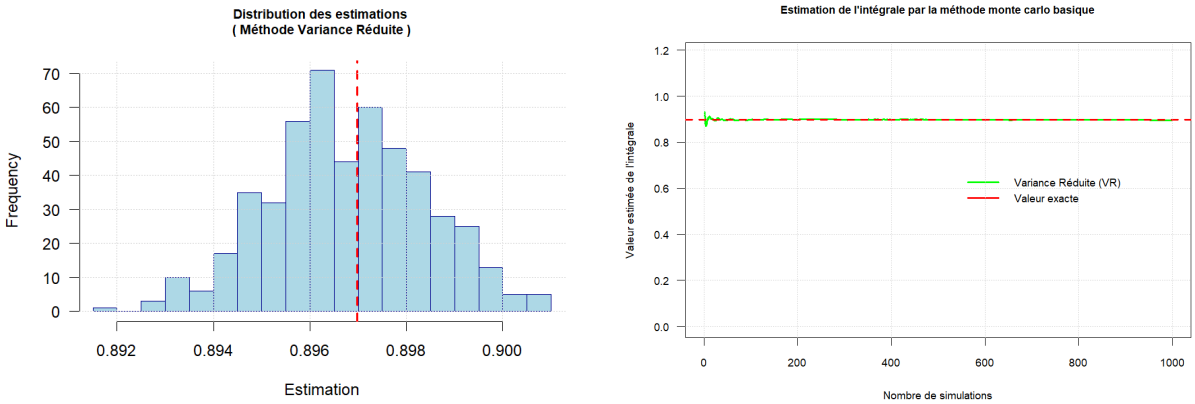


Figura 6: Graphique de f(x,y) pour x : a, médiane, b

Le graphique obtenu visualise la fonction $f(x, y)$ pour 3 valeurs de x (minimum, médiane et maximum) en fonction de y.

En observant la zone ombrée, on peut voir l'étendue des valeurs possibles de $f(x, y)$ pour les différentes valeurs de y.

Nous obtenons donc les résultats suivant :



(a) Distribution des estimations

(b) Estimation de l'intégrale

Figura 7: Résultats Méthode Variance Réduite

D'après l'Histogramme, la dispersion des estimations est faible, ce qui indique une très bonne précision.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe verte . On observe que les estimations se stabilise "presque"parfaitement autour de la valeur exacte, on frole la perfection . Cette méthode est très efficace et ça va être compliqué de faire mieux.

MÉTHODE 4 : MÉTHODE DES COPULES (AVEC LA MARGE DE LA MÉTHODE 2)

Cette quatrième méthode, nommée **Méthode avec copules et marges méthode 2**, utilise une copule de Clayton pour modéliser la dépendance entre les variables X et Y. L'objectif est de capturer la structure de dépendance entre les variables afin d'améliorer l'estimation de l'intégrale en tenant compte des corrélations existantes. En appliquant une copule, nous générons des échantillons dépendants qui reproduisent mieux la distribution jointe cible.

Nous avons effectué le code suivant :

```
# Fonction a integrer avec rotation de 180 degres
# Cette fonction applique une transformation a deux variables x et y
# en utilisant une fonction exponentielle et la fonction sinus,
# afin de creer une surface qui depend des parametres a, b, c et d.
f_xy_rotated <- function(x, y) {
  exp(sin((a + b - x) * (c + d - y)))  # Transformation des coordonnees
}

# Calcul des valeurs Z en utilisant 'outer' pour appliquer f_xy_rotated
# 'outer' cree une matrice de resultats en appliquant f_xy_rotated a chaque combinaison
# des valeurs x et y. Cela produit une surface Z correspondante.
valeurs_z_rotated <- outer(valeurs_x, valeurs_y, f_xy_rotated)

# Affichage du graphique 3D avec la fonction transformee
# On utilise 'persp3d' pour tracer la surface 3D avec les valeurs generees
persp3d(valeurs_x, valeurs_y, valeurs_z_rotated,
        col = "lightblue",              # Couleur de la surface
        xlab = "X",                     # Etiquette de l'axe X
        ylab = "Y",                     # Etiquette de l'axe Y
```

```
        zlab = "f(x, y)",                # Etiquette de l'axe Z
        phi = 30,                        # Angle de la vue en elevation
        theta = 30,                      # Angle de la vue en rotation
        lit = TRUE,                      # Eclairage de la surface
        axes = TRUE)                     # Afficher les axes

# Parametres et correlation cible
tau_cible <- 0.5 # Valeur a ajuster pour la dependance souhaitee entre X et Y

# Fonction pour minimiser la difference entre tau cible et tau genere
# Cette fonction calcule la difference absolue entre le tau calcule a partir de la copule
# et le tau_cible, afin de trouver le parametre de dependance optimal.
min_diff_tau <- function(theta) {
  copule <- claytonCopula(param = theta, dim = 2)  # Creer la copule de Clayton
  tau_calc <- tau(copule)  # Calcul du tau pour la copule de Clayton
  return(abs(tau_calc - tau_cible))  # Retourner la difference
}

# Optimisation pour trouver le theta optimal
# On utilise 'optimize' pour minimiser la difference entre tau calcule et tau cible
res <- optimize(min_diff_tau, interval = c(0.01, 10))  # Plage pour theta
theta_clayton <- res$minimum  # Extraire la valeur de theta optimale

# Affichage du theta optimal trouve
cat("Theta optimal pour atteindre la dependance cible : ", theta_clayton, "\n")

# Generation d'echantillons
n <- n_simulations * n_iterations  # Nombre total d'echantillons a generer

# Generer deux series de variables uniformes independantes
U <- runif(n)  # Premiere variable uniforme
W <- runif(n)  # Variable auxiliaire pour generer la dependance

# Calcul de V en utilisant la dependance de la copule de Clayton
V <- (U^(-theta_clayton) * (W^(-theta_clayton / (1 + theta_clayton)) - 1) + 1)^(-1 /
theta_clayton)

# Assemblage des paires (U, V)
echantillons_uv <- cbind(U, V)

# Transformation des echantillons (u, v) vers (x, y)
# La transformation convertit les marges uniformes [0, 1] en [a, b] pour X et [c, d] pour Y
x_ech <- matrix(qunif(echantillons_uv[, 1], min = a, max = b), ncol = n_iterations)
# Transforme U en X
y_ech <- matrix(qunif(echantillons_uv[, 2], min = c, max = d), ncol = n_iterations)
# Transforme V en Y

# Calcul de f_rotated(x, y) pour chaque echantillon
valeurs_z_copules <- f_xy_rotated(x_ech, y_ech)

# Calcul des probabilites marginales
u <- punif(x_ech, min = a, max = b)  # Probabilite pour X uniforme sur [a, b]
v <- punif(y_ech, min = c, max = d)  # Probabilite pour Y uniforme sur [c, d]

# Calcul des densites marginales
f_X <- dunif(x_ech, min = a, max = b)  # Densite de X
f_Y <- dunif(y_ech, min = c, max = d)  # Densite de Y

# Fonction de la densite de la copule de Clayton
c_clayton_density <- function(u, v, theta) {
  (1 + theta) * (u * v)^(-(1 + theta)) * (u^(-theta) + v^(-theta) - 1)^(-(2 + 1/theta))
```

```
}

# Calcul de la densite des echantillons selon Sklar
valeurs_p_copules <- c_clayton_density(u, v, theta_clayton) * f_X * f_Y

# Approximation de l'integrale avec copules
# Cn_evol utilise la methode des sommes cumulees pour approximer une integrale
Cn_evol <- cumsum(valeurs_z_copules[, 1] / valeurs_p_copules[, 1]) / (1:n_simulations)

# Calcul de l'estimation de Cn pour toutes les simulations
# Fonction pour calculer l'estimation de Cn en utilisant les valeurs de Z et de P
calculer_estimation_copules <- function(valeurs_z, valeurs_p) {
  mean(valeurs_z / valeurs_p)  # Calcul de l'estimation
}

# Appliquer la fonction de calcul sur chaque colonne de la matrice
Cn_vecteur <- sapply(1:n_iterations, function(i) {
  calculer_estimation_copules(valeurs_z_copules[, i], valeurs_p_copules[, i])
  # Calcul pour chaque iteration
})
```

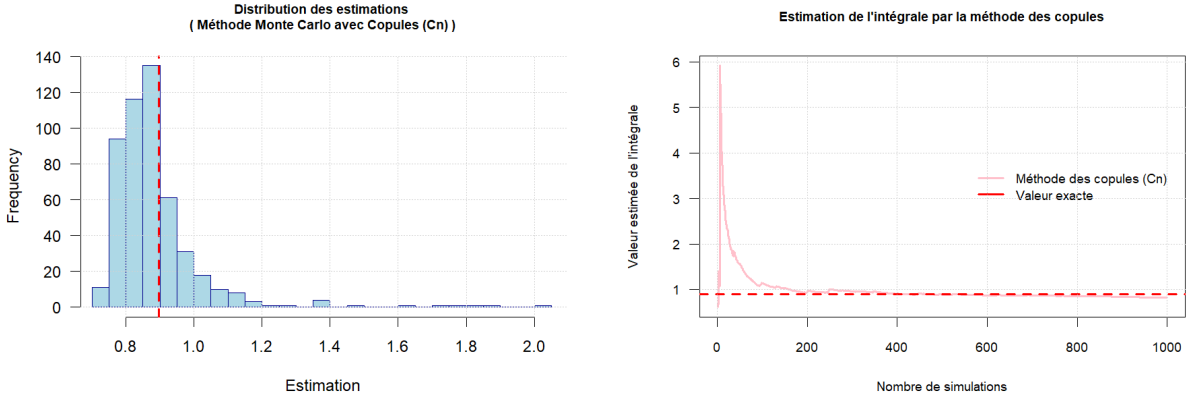
La fonction `f_xy_rotated` représente l'intégrale à estimer avec une transformation de rotation de 180 degrés appliquée aux variables x et y. Nous avons fait cette rotation pour avoir des x et y petits (pour que ça ressemble à une Claython).
Cette fonction utilise une combinaison exponentielle et sinus pour créer une surface dépendant des paramètres a, b, c, et d. Ensuite on utilise la fonction outer pour appliquer `f_xy_rotated` à toutes les combinaisons de valeurs de x et y, générant ainsi une matrice `valeurs_z_rotated`.

Le paramètre de dépendance est défini en fixant une valeur cible de Kendall's tau (tau_cible). La fonction `min_diff_tau` minimise la différence entre le tau généré par la copule de Clayton et ce tau cible afin d'optimiser le paramètre θ de la copule pour atteindre la dépendance souhaitée. Le paramètre optimal est ensuite trouvé à l'aide de la fonction optimize.

Deux séries de variables uniformes indépendantes, U et W, sont générées pour simuler des paires dépendantes (U, V) via la copule de Clayton. Le vecteur V est calculé pour capturer la dépendance définie par θ, et les paires sont assemblées dans echantillons_uv.

Les échantillons uniformes U et V sont transformés aux marges $[a, b]$ pour X et $[c, d]$ pour Y, via la fonction `qunif`. Les résultats sont stockés dans `x_ech` et `y_ech`, représentant les valeurs de X et Y transformées. La fonction `f_xy_rotated` est appliquée aux échantillons transformés, et les résultats sont stockés dans `valeurs_z_copules`, correspondant aux valeurs de $f(x, y)$ pour chaque paire d'échantillons. Les probabilités cumulées u et v et les densités marginales f_X et f_Y sont calculées pour les échantillons x et y. Ces valeurs sont utilisées pour normaliser les échantillons et estimer la densité jointe.
La fonction `c_clayton_density` calcule la densité de la copule de Clayton pour chaque paire (u, v), en fonction du paramètre θ. Cette densité est combinée avec les densités marginales pour obtenir les valeurs `valeurs_p_copules` nécessaires à l'estimation de l'intégrale.
Enfin, la somme cumulative `Cn_evol` est calculée pour approximer l'intégrale en divisant les valeurs `valeurs_z_copules` par `valeurs_p_copules`. La fonction `calculer_estimation_copules` applique cette estimation pour chaque itération, et `Cn_vecteur` stocke les résultats pour toutes les simulations, permettant d'analyser la convergence de l'estimation.

Nous obtenons donc les résultats suivant :



(a) Distribution des estimations **(b)** Estimation de l'intégrale

Figura 8: Résultats Méthode des copules avec la marge de la méthode 2

D'après l'Histogramme, la dispersion des estimations est assez faible, ce qui indique une bonne précision.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe rose . On observe que les estimations se stabilise autour de la valeur exacte à partir d'environ 200 simulations.

MÉTHODE 5 : MÉTHODE DES COPULES (AVEC LA MARGE DE LA MÉTHODE 3)

La **Méthode 5** repose également sur l'utilisation des copules pour intégrer la fonction f_{xy}, en suivant des étapes similaires à la **Méthode 4**. Comme pour la Méthode 4, cette approche implique la **copule de Clayton** pour modéliser la dépendance entre les variables X et Y, ainsi que l'utilisation de marges spécifiques. Cependant, cette méthode utilise les marges de la Méthode 3 pour X et Y, plutôt qu'une configuration marginale distincte.

Nous avons effectué le code suivant :

```
# Calculer les valeurs de f_xy_rotated(x, y_fixe) pour chaque combinaison de x et y
f_x_yfixe_rotated <- sapply(y_fixe, function(y) sapply(valeurs_x, function(x) f_xy_rotated(x, y))
# Creation d'une matrice ou chaque colonne correspond a f_xy_rotated(x, y) pour une valeur de y
# fixee

# Tracer la courbe pour y = c (correspondant a y_min) avec une ligne noire pointillee
plot(valeurs_x, f_x_yfixe_rotated[, 1], type = "l", col = "black", lwd = 2, lty = 2,
     xlab = "X", ylab = "f_rotated(X, Y)",
     main = "Graphique de f_rotated(x, y) pour y : c, mediane, d",  # Titre plus explicite
     ylim = c(1, 3),
     cex.main = 0.8,   # Taille du titre reduite
     cex.lab = 0.8,    # Taille des etiquettes des axes reduite
     cex.axis = 0.8)   # Taille des etiquettes des axes reduite

# Ajouter la courbe pour y = d (correspondant a y_max) avec une ligne noire pointillee
lines(valeurs_x, f_x_yfixe_rotated[, 3], col = "black", lwd = 2, lty = 2)

# Colorier la region entre y = c (y_min) et y = d (y_max) en bleu transparent
polygon(c(valeurs_x, rev(valeurs_x)),
        c(f_x_yfixe_rotated[, 1], rev(f_x_yfixe_rotated[, 3])),
        col = rgb(0.1, 0.1, 0.8, 0.2), border = NA)  # Remplissage en bleu transparent
```

```r
# Ajouter la courbe pour la valeur mediane de y avec une ligne rouge
lines(valeurs_x, f_x_yfixe_rotated[, 2], col = "red", lwd = 2)

# Calculer et tracer la fonction p1 en bleu
p1_rotated_valeurs <- 8 / 7 * (2.5 - 3 * valeurs_x)  # Definition de p1 en fonction de x
lines(valeurs_x, p1_rotated_valeurs, col = "blue", lwd = 2)  # Tracer la courbe de p1 en bleu

# Ajouter une legende pour identifier les courbes
legend("topright",
       legend = c(paste("y =", c),          # Courbe pour y = c (y_min)
                  paste("y =", (c + d) / 2), # Courbe pour la valeur mediane de y
                  paste("y =", d),           # Courbe pour y = d (y_max)
                  "p1_rotated = 8/7 * (2.5 - 3x)"),  # Formule pour p1
       col = c("black", "red", "black", "blue"),  # Couleurs des courbes dans la legende
       lwd = 2, lty = c(2, 1, 2, 1))  # Styles de lignes dans la legende

# Afficher une grille sur le graphique
grid()

# Definition de p1_rotated, la densite de probabilite pour X
p1_rotated <- function(x) 8 / 7 * (2.5 - 3 * x)

# Fonction de repartition cumulee de p1_rotated
F1_rotated <- function(t) {
  return(4 / 7 * (5 * t - 3 * t^2))  # F1_rotated(t) correspond a la fonction de repartition
  # cumulee de p1_rotated
}

# Inverse de la fonction de repartition F1_rotated (utile pour generer des echantillons de X
# selon p1_rotated)
F1_rotated_inverse <- function(z) {
  return((20 - sqrt(400 - 336 * z)) / 24)  # Resolution analytique de l'inverse de F1_rotated
}

# Histogramme des valeurs generees par la fonction inverse de F1_rotated avec une distribution
# uniforme
hist(F1_rotated_inverse(runif(10000)),
     col = rgb(0.1, 0.5, 0.8, 0.7),  # Couleur des barres avec transparence
     border = "white",  # Couleur de la bordure des barres
     xlab = "Valeurs de X",  # Etiquette de l'axe X
     ylab = "Frequence",  # Etiquette de l'axe Y
     main = paste("Histogramme des echantillons de X", "\nselon la densite de proba 1"),
     # Titre avec saut de ligne
     xlim = c(0, 0.5),  # Limites de l'axe X (ajuster selon votre intervalle)
     ylim = c(0, 1400),  # Limites de l'axe Y (ajuster selon votre distribution)
     las = 1,  # Orientation des etiquettes de l'axe Y
     freq = TRUE,
     cex.main = 0.8,   # Taille du titre reduite
     cex.lab = 0.8,    # Taille des etiquettes des axes reduite
     cex.axis = 0.8)  # Frequence ou densite (TRUE pour frequence)

# Calculer les valeurs de f_xy_rotated(x, y_fixe) pour chaque combinaison de x et y
f_xfixe_y_rotated <- sapply(x_fixe, function(x) sapply(valeurs_y, function(y) f_xy_rotated(x, y)))
# Creation d'une matrice ou chaque colonne correspond a f_xy_rotated(x, y) pour une valeur de x
# fixee

# Tracer la courbe pour x = a (correspondant a x_min) avec une ligne noire pointillee
plot(valeurs_y, f_xfixe_y_rotated[, 1], type = "l", col = "black", lwd = 2, lty = 2,
     xlab = "Y", ylab = "f_rotated(X, Y)",
     main = "Graphique de f_rotated(x, y) pour x : a, mediane, b",  # Titre plus clair
```

```
     ylim = c(1, 3),
     cex.main = 0.8,   # Taille du titre reduite
     cex.lab = 0.8,    # Taille des etiquettes des axes reduite
     cex.axis = 0.8)  # Definir les limites de l'axe Y de 1 a 3

# Ajouter la courbe pour x = b (correspondant a x_max) avec une ligne noire pointillee
lines(valeurs_y, f_xfixe_y_rotated[, 3], col = "black", lwd = 2, lty = 2)

# Colorier la region entre a (x_min) et b (x_max) en bleu transparent
polygon(c(valeurs_y, rev(valeurs_y)),
        c(f_xfixe_y_rotated[, 1], rev(f_xfixe_y_rotated[, 3])),
        col = rgb(0.1, 0.1, 0.8, 0.2), border = NA)  # Remplissage en bleu transparent

lines(valeurs_y, f_xfixe_y_rotated[, 2], col = "red", lwd = 2)
# Ajouter la courbe pour la valeur mediane de x avec une ligne rouge
legend("bottomright", legend = c(paste("x =", a),  paste("x =", (a + b) / 2), paste("x =", b)),
col = c("black", "red", "black"), lwd = 2, lty = c(2, 1, 2))
# Ajouter une legende pour identifier les courbes

# Definir p2_rotated, la densite de probabilite uniforme pour Y
p2_rotated <- function(y) 1  # Uniforme, donc constante entre c et d

# Definir p_rotated, la fonction de densite conjointe
p_rotated = function(x, y) p1_rotated(x) * p2_rotated(y)
# Densite conjointe p_rotated(x, y) = p1_rotated(x) * p2_rotated(y)

# Generer deux series de variables uniformes independantes
U <- runif(n)  # Premiere variable uniforme
W <- runif(n)  # Variable auxiliaire pour generer la dependance

# Calcul de V en utilisant la dependance de la copule de Clayton
V <- (U^(-theta_clayton) * (W^(-theta_clayton / (1 + theta_clayton)) - 1) + 1)^(-1 /
theta_clayton)

# Assemblage des paires (U, V)
echantillons_uv <- cbind(U, V)

# Transformation des echantillons (u, v) vers (x, y)
# La transformation convertit les marges uniformes [0, 1] en [a, b] pour X et [c, d] pour Y
x_ech <- matrix(F1_rotated_inverse(echantillons_uv[, 1]), ncol = n_iterations)
# Transforme U en X
y_ech <- matrix(qunif(echantillons_uv[, 2], min = c, max = d), ncol = n_iterations)
# Transforme V en Y

# Calcul de f_rotated(x, y) pour chaque echantillon
valeurs_z_copules2 <- f_xy_rotated(x_ech, y_ech)

# Calcul des probabilites marginales
u <- F1_rotated(x_ech)  # Probabilite pour X uniforme sur [a, b]
v <- punif(y_ech, min = c, max = d)  # Probabilite pour Y uniforme sur [c, d]

# Calcul des densites marginales
f_X <- p1_rotated(x_ech)  # Densite de X
f_Y <- dunif(y_ech, min = c, max = d)  # Densite de Y

# Calcul de la densite des echantillons selon Sklar
valeurs_p_copules2 <- c_clayton_density(u, v, theta_clayton) * f_X * f_Y

# Approximation de l'integrale avec copules
# Cn_evol utilise la methode des sommes cumulees pour approximer une integrale
Cn_2_evol <- cumsum(valeurs_z_copules2[, 1] / valeurs_p_copules2[, 1]) / (1:n_simulations)
```

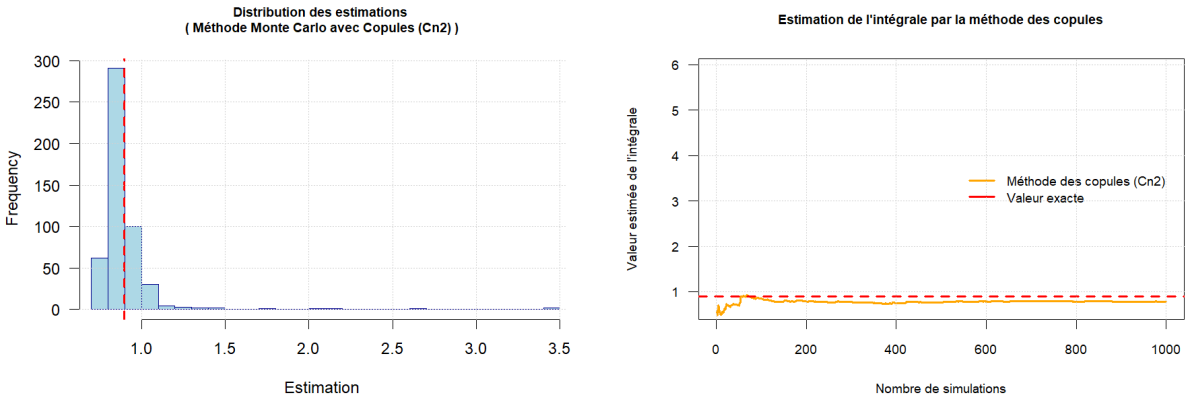
```
# Appliquer la fonction de calcul sur chaque colonne de la matrice
Cn_vecteur2 <- sapply(1:n_iterations, function(i) {
  calculer_estimation_copules(valeurs_z_copules2[, i], valeurs_p_copules2[, i])
  # Calcul pour chaque iteration
})
```

La **Méthode 5** repose également sur l'utilisation des copules pour intégrer la fonction f_{xy}, en suivant des étapes similaires à la **Méthode 4** mais avec quelques ajustements. Comme en Méthode 4, cette approche implique la **copule de Clayton** pour modéliser la dépendance entre les variables X et Y, ainsi que l'utilisation de marges spécifiques. Cependant, cette méthode utilise les marges de la Méthode 3 pour X et Y, plutôt qu'une configuration marginale distincte.

Les étapes clés de la Méthode 5 comprennent :

1. **Calcul de** $f_{xy\_rotated}$: Pour chaque combinaison de x et y, nous calculons $f_{xy\_rotated}(x, y)$, de la même manière que dans la Méthode 4. Une matrice est ainsi constituée, où chaque colonne correspond à une valeur fixée de y.
2. **Densité marginale** $p1\_rotated$: Nous définissons la densité $p1\_rotated(x)$, similaire à celle en Méthode 4, avec une fonction de répartition cumulée et son inverse, utiles pour générer des échantillons.
3. **Génération d'échantillons** : Deux variables uniformes indépendantes U et W sont générées, et V est calculée en appliquant la dépendance de la copule de Clayton. Les échantillons U et V sont ensuite transformés pour obtenir x_ech et y_ech via les marges.
4. **Estimation de l'intégrale** : À l'aide des valeurs calculées pour $f_{xy\_rotated}(x, y)$ et les densités marginales, l'intégrale est approchée par la méthode Monte Carlo. En appliquant des cumuls de rapports $\frac{f_{xy\_rotated}}{p\_rotated}$ au fur et à mesure des simulations, nous obtenons une série d'estimations progressives, tracées dans le graphique.

Nous obtenons ainsi les résultats suivants :



(a) Distribution des estimations **(b)** Estimation de l'intégrale

Figura 9: Résultats Méthode des Copules avec la marge de la méthode 3

D'après l'Histogramme, la dispersion des estimations est faible, ce qui indique une bonne précision.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe orange . On observe que les estimations ont du mal à se stabilier autour de la valeur exacte. Il faudra tester avec plus de simulations (voir 12).

Méthode Quasi-Monte Carlo (avec séquences de Halton)

Pour finir, la sixième méthode, nommée **Méthode Quasi-Monte Carlo (avec séquences de Halton)**, exploite la méthode Quasi-Monte Carlo pour estimer l'intégrale, en utilisant des séquences de Halton pour générer des points quasi-aléatoires dans le domaine d'intégration. Contrairement aux méthodes Monte Carlo standards qui reposent sur des points aléatoires, les séquences de Halton améliorent la couverture uniforme de l'espace de simulation, ce qui peut réduire la variance de l'estimation.
Nous avons effectué le code suivant :

```
# Generation de points de Halton pour le nombre total de simulations et d'iterations
halton_points <- halton(n_simulations * n_iterations, dim = 2)

# Mise a l'echelle des points de Halton pour les limites definies
# x_ech et y_ech sont des matrices ou chaque colonne correspond a une iteration
x_ech <- matrix(a + (b - a) * halton_points[, 1], ncol = n_iterations)
y_ech <- matrix(c + (d - c) * halton_points[, 2], ncol = n_iterations)

# Calcul de f(x, y) pour chaque echantillon tire
valeurs_z_halton <- f_xy(x_ech, y_ech)

# Approximation de l'integrale par Quasi-Monte Carlo
# aire_rectangle est le volume du domaine d'integration
aire_rectangle <- (b - a) * (d - c)
Hn_evol <- cumsum(valeurs_z_halton[, 1]) / (1:n_simulations) * aire_rectangle

# Fonction pour calculer l'estimation de l'integrale
calculer_estimation_qmc <- function(valeurs) {
  mean(valeurs) * aire_rectangle  # Retourne la moyenne des valeurs multipliee par le volume
}

# Calcul des estimations pour toutes les simulations
Hn_vecteur <- apply(valeurs_z_halton, MARGIN = 2, FUN = calculer_estimation_qmc)
```

Les étapes principales de cette méthode sont décrites ci-dessous :

1. **Génération de points de Halton** : Les points de Halton sont générés pour le nombre total de simulations et d'itérations dans un espace de dimension 2 (pour les variables X et Y). Ces séquences fournissent une meilleure répartition des points dans l'intervalle que des points aléatoires classiques.

2. **Mise à l'échelle des points de Halton** : Chaque point généré est mis à l'échelle pour correspondre aux limites du domaine d'intégration, définies par les bornes $a \leqslant x \leqslant b$ et $c \leqslant y \leqslant d$. Les échantillons x_ech et y_ech sont donc transformés afin que chaque colonne de la matrice corresponde à une itération.

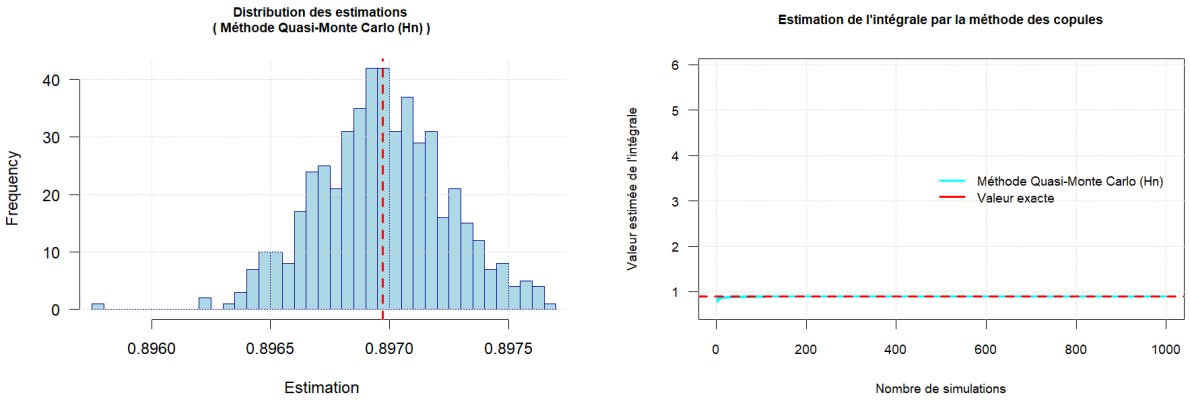
3. **Calcul de $f(x, y)$ pour chaque échantillon** : Les valeurs de la fonction $f(x, y)$ sont calculées pour chaque couple (x, y) issu des points de Halton. Ces valeurs sont stockées dans la matrice $valeurs\_z\_halton$.

4. **Approximation de l'intégrale par Quasi-Monte Carlo** : La valeur de l'intégrale est approchée en multipliant la moyenne des valeurs de $f(x, y)$ par l'**aire du rectangle** qui représente

le volume du domaine d'intégration, donné par $(b-a)\times(d-c)$. L'évolution des estimations au fil des simulations est stockée dans le vecteur Hn_evol.

5. **Calcul de l'estimation finale de l'intégrale** : Une fonction `calculer_estimation_qmc` est définie pour calculer l'estimation finale de l'intégrale. Elle prend la moyenne des valeurs calculées pour $f(x,y)$ et la multiplie par l'aire du domaine, renvoyant ainsi l'estimation de l'intégrale. Cette estimation est appliquée à chaque simulation pour obtenir un vecteur $Hn\_vecteur$ des estimations à chaque itération.

Nous obtenons les résultats suivants :



(a) Distribution des estimations **(b)** Estimation de l'intégrale

Figura 10: Résultats Méthode Quasi-Monte Carlo

D'après l'Histogramme, la dispersion des estimations est faible, ce qui indique une très bonne précision.

A droite nous avons l'évolution de l'estimation de l'intégrale en fonction du nombre de simulations effectuées, représentée par la courbe cyan . On observe que les estimations se stabilise parfaitement autour de la valeur exacte. Cette méthode est très efficace.

RÉSULTATS

Calcul des risques quadratiques

```
# Estimation de l'integrale par un calcul numerique
I <- integral2(f_xy, a, b, c, d) $ Q

# Risques quadratiques
RQ_Tn <- (mean(Tn_vecteur) - I)^2 + sd(Tn_vecteur)^2
RQ_Sn <- (mean(Sn_vecteur) - I)^2 + sd(Sn_vecteur)^2
RQ_VR <- (mean(VR_vecteur) - I)^2 + sd(VR_vecteur)^2
RQ_Cn <- (mean(Cn_vecteur) - I)^2 + sd(Cn_vecteur)^2
RQ_Cn2 <- (mean(Cn_vecteur2) - I)^2 + sd(Cn_vecteur2)^2
RQ_Hn <- (mean(Hn_vecteur) - I)^2 + sd(Hn_vecteur)^2
```

Ici, la fonction $integral2$ est utilisée pour effectuer des intégrations numériques sur des fonctions à deux variables et d'estimer l'intégrale de notre fonction f_xy sur le domaine défini par les limites $xmin$, $xmax$, $ymin$ et $ymax$.

Nous calculons ensuite les risques quadratiques en utilisant la formule :

$$\mathrm{RQ} = Variance + \mathrm{biais}^2$$

Nous obtenons :

	Méthode de Volume (Tn)	**Méthode de Monte Carlo Basique (Sn)**	**Méthode de Variance Réduite (VR)**	**Méthode Monte Carlo avec Copules (Cn)**	**Méthode Monte Carlo avec Copules (Cn2)**	**Méthode Quasi-Monte Carlo (Hn)**
Valeur approximée par la méthode	0.8969417	0.8967804	0.8969748	0.8949798	0.8931295	0.8969679
Risque Quadratique	3.962171×10^{-4}	5.666369×10^{-5}	2.49247×10^{-6}	3.918406×10^{-2}	2.495568×10^{-2}	7.387663×10^{-8}

Les résultats obtenus montrent que la Méthode de Volume (Tn) et la Méthode de Variance Réduite (VR) offrent des estimations proches de la valeur exacte, avec la VR affichant le risque quadratique le plus faible, ce qui en fait une méthode particulièrement efficace pour des estimations précises.

En revanche, les méthodes de Monte Carlo utilisant des copules (Cn et Cn2) présentent des estimations moins précises et des risques quadratiques élevés, ce qui peut limiter leur application dans des contextes nécessitant une grande précision.

La Méthode Quasi-Monte Carlo (Hn) se distingue par une estimation précise et un risque quadratique très faible.

Soit :

$$\mathrm{Hn} < \mathrm{VR} < \mathrm{Sn} < \mathrm{Tn} < \mathrm{Cn} < \mathrm{Cn2}$$

Les visualisations suivantes vont nous permettre de confirmer ces affirmations.

Résultats et visualisation

Les **Figures 11** et **12** ci-dessous montre une comparaison des méthodes d'estimation de l'intégrale avec N=1 000 et N=10 000.

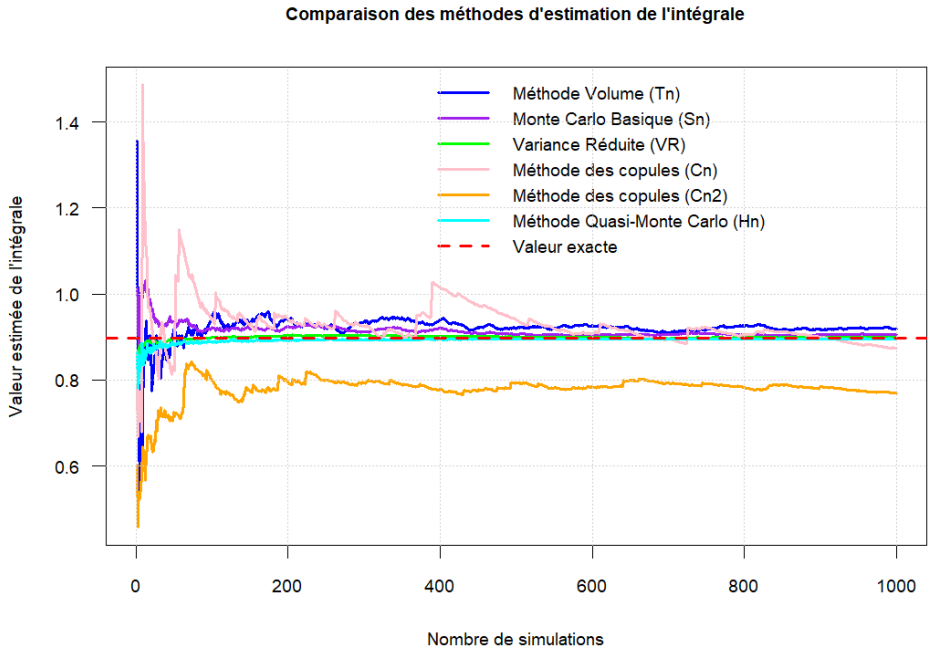


Figura 11: Comparaison des méthodes d'estimations de l'intégrale avec N= 1 000

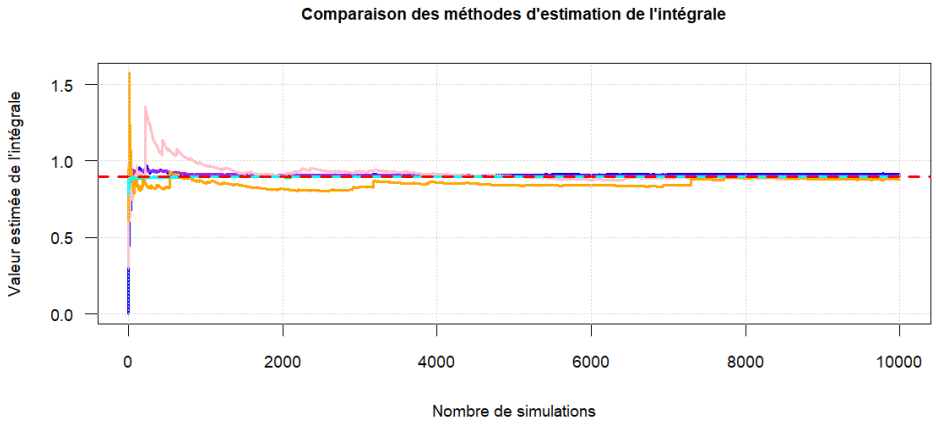


Figura 12: Comparaison des méthodes d'estimations de l'intégrale avec N= 10 000

On constate que la Méthode Quasi-Monte Carlo est celle qui converge le plus efficacement vers la valeur exacte, suivie par la méthode de Variance Réduite, la méthode Monte Carlo Basique, la méthode de Volume, et enfin les méthodes basées sur les copules.
Toutes les méthodes atteignent une certaine stabilité à partir d'environ 7000 simulations.

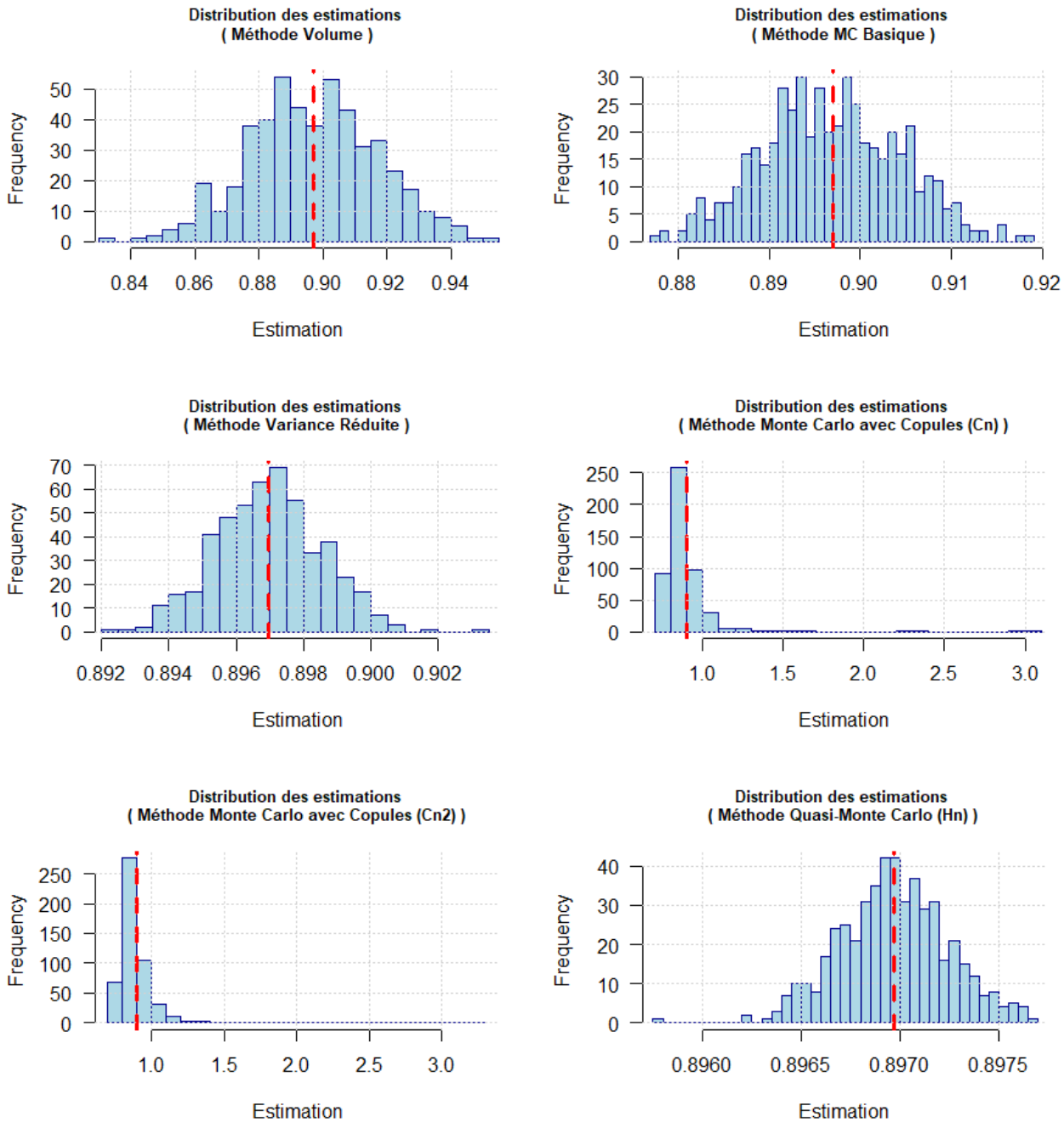


Figura 13: Comparaison de l'estimation avec la vraie fonction f pour diverses valeurs de N

Conclusion

Dans ce projet, nous avons exploré diverses méthodes d'estimation de l'intégrale de la fonction $\exp(\sin(x \cdot y))$, chacune ayant des avantages et des inconvénients propres, que nous résumons dans le tableau suivant.

	Avantages	**Inconvénients**
Méthode de Volume (Tn)	- Simplicité de mise en œuvre - Estimation d'erreurs - Parallélisation	- Sensibilité aux générateurs de nombres aléatoires - Convergence lente - Sensibilité aux générateurs de nombres aléatoires
Méthode de Monte Carlo Basique (Sn)	- Bonne généralisation pour différentes distributions - Simple et adaptable	- Convergence lente - Nécessite un grand nombre de simulations
Méthode de Variance Réduite (VR)	- Réduction significative de la variance - Amélioration de la précision	- Implémentation plus complexe - Nécessite une fonction de contrôle bien choisie
Méthode des Copules avec la marge de la méthode 2	- Capture les dépendances entre variables - Flexible avec les copules choisies	- Difficulté de choix de la copule - Paramétrage complexe
Méthode des Copules avec la marge de la méthode 3	- Captures plus efficacement les dépendances - Optimisation des marges	- Implémentation complexe - Sensibilité au paramétrage
Méthode Quasi-Monte Carlo (avec séquences de Halton)	- Convergence plus rapide que Monte Carlo classique - Précision accrue pour faibles dimensions	- Moins efficace pour les dimensions élevées - Nécessite un générateur de séquences déterministe

Chaque méthode a donc ses particularités et son domaine d'application optimal. Les méthodes de Monte Carlo classiques ont montré des avantages en termes de simplicité, mais leur convergence reste limitée. Les techniques de copules ont permis d'incorporer des relations de dépendance, augmentant la précision dans des contextes de dépendance forte. Enfin, l'approche Quasi-Monte Carlo avec les séquences de Halton s'est avérée efficace pour réduire l'erreur dans des cas de faible dimension, en exploitant des séquences déterministes bien réparties.

Ce projet a mis en évidence les différentes approches d'intégration numérique et l'importance de choisir la méthode adaptée aux caractéristiques de la fonction et aux contraintes pratiques.