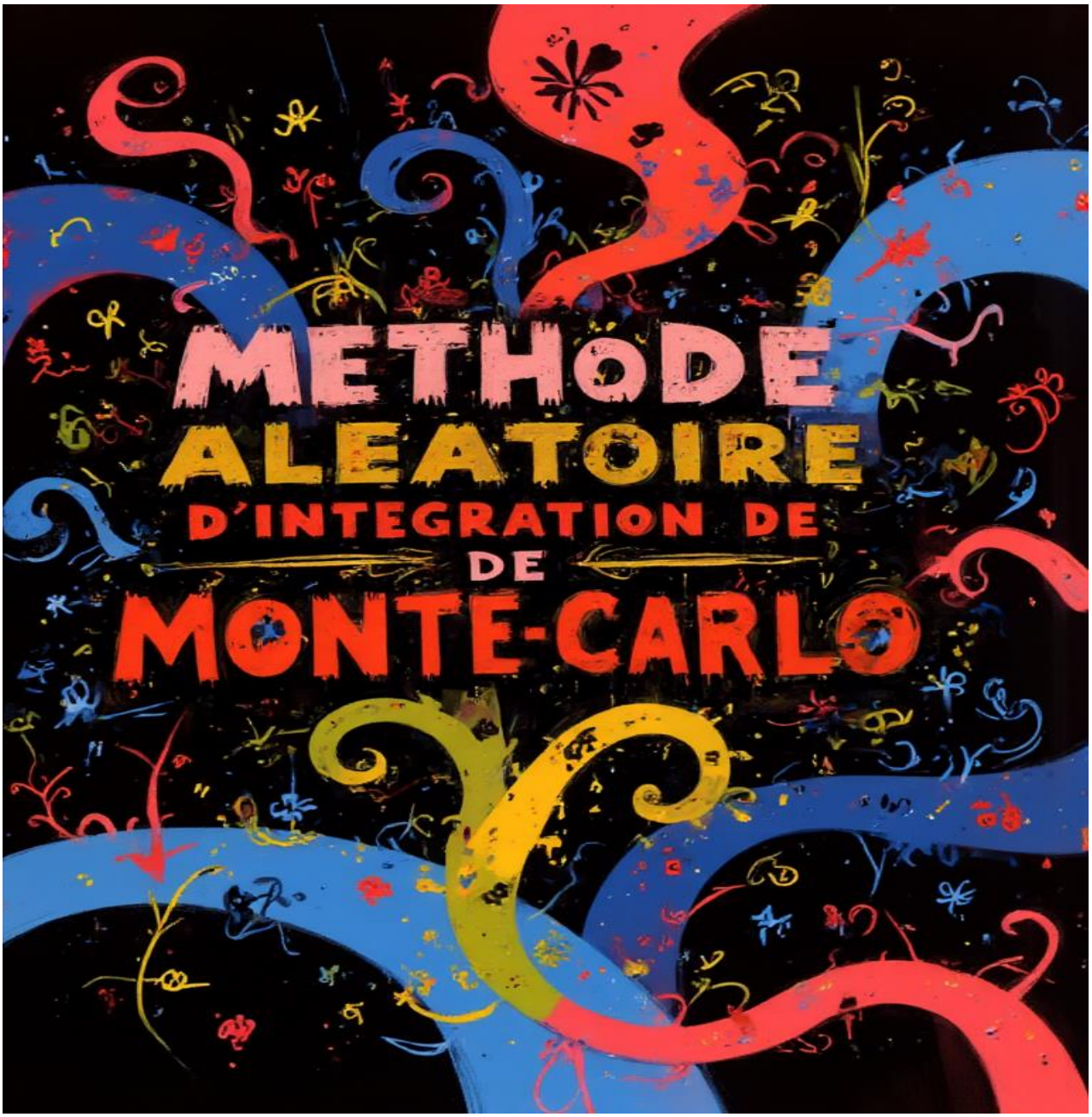
Université Gustave Eiffel

Le 06/05/2024



Par Anny IRANKUNDA

L2 MIASHS

Université Gustave Eiffel

Table des matières

Introduction

Dans le cadre de ce rapport, notre objectif est de réaliser une analyse approfondie de la méthode de Monte-Carlo. Nous nous attacherons à expliquer en détail le processus de calcul d'une intégrale à travers cette méthode innovante. De plus, nous effectuerons une comparaison avec la méthode des rectangles afin de mettre en évidence les avantages significatifs offerts par la méthode de Monte-Carlo, surtout dans des espaces de dimension élevée.

Nous explorerons également la mise en pratique de ces méthodes en utilisant le langage de programmation Python. Notre démarche consistera à programmer ces méthodes, les tester et analyser les résultats obtenus. Nous accorderons une attention particulière aux erreurs et aux performances temporelles de chaque méthode.

I. Présentation de la méthode Monte-Carlo

La méthode de Monte-Carlo tire son nom du célèbre casino de Monte-Carlo à Monaco, qui est associé à des jeux de hasard et de probabilités. Cette méthode a été développée dans les années 1940 par des scientifiques travaillant sur le projet Manhattan, notamment Stanislaw Ulam et John von Neumann, pour résoudre des problèmes liés à la diffusion neutronique dans les réacteurs nucléaires. Ils ont remarqué que les problèmes de probabilité complexes pouvaient être résolus de manière efficace en utilisant des simulations aléatoires.

Le nom "Monte-Carlo" a été choisi par Ulam en référence au casino, pour refléter le caractère probabiliste et aléatoire de la méthode.

L'analogie avec le jeu de roulette de Monte-Carlo est souvent citée pour expliquer la méthode. Dans un jeu de roulette, la bille rebondit de manière aléatoire sur le plateau avant de s'arrêter sur un numéro. De la même manière, dans la simulation de Monte-Carlo, des valeurs aléatoires sont générées pour représenter les différentes variables ou paramètres d'un problème, et ces valeurs sont utilisées pour estimer le résultat final.

La méthode de Monte-Carlo est une approche numérique qui repose sur la génération de nombres aléatoires pour résoudre des problèmes déterministes ou stochastiques. Elle est souvent utilisée pour estimer des valeurs d'intégrales en échantillonnant aléatoirement des points dans l'espace d'intégration. En calculant une moyenne des valeurs de la fonction à intégrer sur ces points, on obtient une estimation de l'intégrale.

Dans notre étude, nous distinguerons deux méthodes : la méthode naïve et la méthode non naïve.

II. Méthode Naïve de Monte-Carlo

A. Principe de la méthode

Dans cette approche, on assimile l'intégrale d'une fonction à une probabilité. Pour calculer $I = \int_a^b f(x)dx$, où f est une fonction définie sur $[a, b]$ renvoyant des valeurs dans $[c, d]$, on cherche à estimer la probabilité qu'un point (x, y) de loi uniforme dans le rectangle $[a, b] \times [c, d]$ tombe sous la courbe de f. Cette probabilité correspond à l'intégrale souhaitée :

$$I = P[(y \leq f(x) \cap y \leq 0) \cup (y \geq f(x) \cap y \leq 0)]$$

Explication simplifiée avec une liste des étapes :

Pour estimer l'intégrale $I = \int_a^b f(x)dx$ en utilisant la méthode de Monte-Carlo naïve, on suit les étapes suivantes :

1. **Définir le rectangle d'intégration :** Choisir les bornes a et b sur l'axe x et c et d sur l'axe y, formant ainsi le rectangle $[a, b] \times [c, d]$.
2. **Générer des points aléatoires :** Produire n points aléatoires $(x_i, y_i), i \in \{1, n\}$, dans le rectangle $[a, b] \times [c, d]$ en utilisant une distribution uniforme.
3. **Compter les points sous la courbe :** Pour chaque point (x_i, y_i), vérifier si $y_i \leq f(x_i)$. Si c'est le cas, le point est considéré comme tombant sous la courbe.

Attention ! Dans les régions où la fonction est négative, un point (x_i, y_i) est considéré comme tombant sous la courbe quand $y_i \geq f(x_i)$.

Il faut **compter séparément** les points (x_i, y_i) sous la courbe tels que $y_i \geq 0$ (points de l'aire positive) et ceux sous la courbe tels que $y_i \leq 0$ (points de l'aire négative).

4. **Calculer la probabilité estimée :** Pour l'aire positive et l'aire négative **séparément**, diviser le nombre de points tombant sous la courbe par le nombre total de points générés pour obtenir une estimation de la probabilité $P[y \leq f(x)]$.
5. **Estimer l'intégrale :** Utiliser la formule suivante pour estimer les aires positive et négative :

$$Estimation aire\ positive/négative \approx (b-a) \times (d-c) \times \frac{nombre de points sous la courbe (région\ positive/négative)}{n}$$

6. **Formule finale** : l'intégrale d'une fonction étant une somme algébrique, on obtient le résultat final de cette méthode soustrayant l'estimation de l'aire négative à celle de l'aire positive :

$$I \approx estimation aire positive - estimation aire négative$$

Cette méthode repose sur le principe que la proportion de points tombant sous la courbe de f par rapport au nombre total de points tirés dans le rectangle $[a,b]\times[c,d]$ approche la valeur de l'intégrale. Plus le nombre de points tirés est grand, plus l'estimation sera proche de la valeur réelle, conformément à la loi des grands nombres. Cependant, pour des fonctions complexes ou des espaces d'intégration de grande dimension, cette méthode peut nécessiter un très grand nombre de points pour obtenir une estimation précise, ce qui peut être inefficace en termes de temps de calcul.

B. Utilisation concrète de la méthode : estimer pi

La méthode naïve peut être utilisée pour estimer la valeur de π. Pour cela, on se place dans le carré $[0,1]^2$ dans lequel on trace un quart de disque de centre $(0,0)$ *[voir figure 1]*. Comme on se sert de la méthode naïve, on tirera n points (x,y) aléatoires uniformément dans ce plan, puis on calculera la proportion de points tombant dans le quart de disque par rapport au nombre total de points tirés.

- **Loi uniforme sur $[0,1]^2$:**

La v.a.r. $Z\in\Re^2$ suit une loi uniforme sur $[0,1]^2$ si :

$$\forall[a,b]\times[c,d]\subset[0,1]^2, P(Z\in[a,b]\times[c,d])=(b-a)(d-c)$$

On peut retrouver cette formule à partir de la définition d'une loi uniforme sur $[0,1]$.

- **Loi uniforme sur $[0,1]$:**

Une variable aléatoire réelle (v.a.r.) X suit une loi uniforme sur $[a,b](-\infty<a<b<+\infty)$ si elle a une densité constante sur $[a,b]$ et nulle ailleurs. Sa densité s'écrit alors :

$$f(t)=\frac{1}{b-a}\times 1_{[a,b]}(t), \forall t\in\Re$$

Ainsi, $P(X\in[x_0,x_0+\delta]\subset[a,b])=\int_{x_0}^{x_0+\delta}f(t)dt=\frac{1}{b-a}\int_{x_0}^{x_0+\delta}1dt=[t]_{x_0}^{x_0+\delta}=\frac{x_0+\delta-x_0}{b-a}=\frac{\delta}{b-a}$

Soit X et Y deux v.a.r. qui suivent une loi uniforme sur $[0,1]$. X et Y sont indépendantes. On a, d'après la définition précédente :

- $P(X\in[a,b]\subset[0,1])=\frac{1}{1-0}\int_a^b 1_{[0,1]}(t)dt=\int_a^b 1dt=[t]_a^b=b-a$
- de même, $P(Y\in[c,d]\subset[0,1])=\frac{1}{1-0}\int_c^d 1_{[0,1]}(t)dt=d-c.$

Soit $Z=(X,Y)\in[0,1]^2$. On a ainsi :

$P(Z\in[a,b]\times[c,d]\subset[0,1]^2)=P(X\in[a,b]\cap Y\in[c,d])=P(X\in[a,b])\times P(Y\in[c,d])$car X et Y sont indépendantes.

On obtient :

$$P(Z\in[a,b]\times[c,d]\subset[0,1]^2)=(b-a)(d-c).$$

Cette équation correspond à la définition d'une v.a.r. suivant une loi uniforme sur $[0,1]^2$.

D'après la loi des grands nombres, pour $n \to \infty$, on a la convergence :

$$P\big((x,y) \in quartdedisque\big) = \frac{nombredepointsdans\leq disque}{nombretotaldepointstirés} \to \frac{airedudisque}{aireducarré}.$$

Or, on a, avec $D = \frac{\pi}{4}$ l'aire du quart de disque et $C = 1^2 = 1$, l'aire du carré $[0,1]^2$:

$$\frac{D}{C} = \frac{\pi}{4} \Leftrightarrow \pi = 4 \times \frac{D}{C}$$

Le programme Python pour estimer $\frac{D}{C}$, puis π est le suivant :

```
import random

def monte_carlo_pi(num_samples):
    inside_circle = 0

    for _ in range(num_samples):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)

  # Vérifier si le point (x, y) est à l'intérieur du cercle unité
        if x**2 + y**2 <= 1:
            inside_circle += 1

  # Calculer l'estimation de π en utilisant la proportion des points à l'intérieur du cercle

    pi_estimate = 4 * inside_circle / num_samples
    return pi_estimate


num_samples = 10000000  # Utilisation de 10000000 échantillons
estimated_pi = monte_carlo_pi(num_samples)

print("Estimation de π avec", num_samples, "échantillons :", estimated_pi)
# Doit donner environ 3,14159
```

Résultats : Voici quelques résultats de la démarche après avoir lancé le programme plusieurs fois.

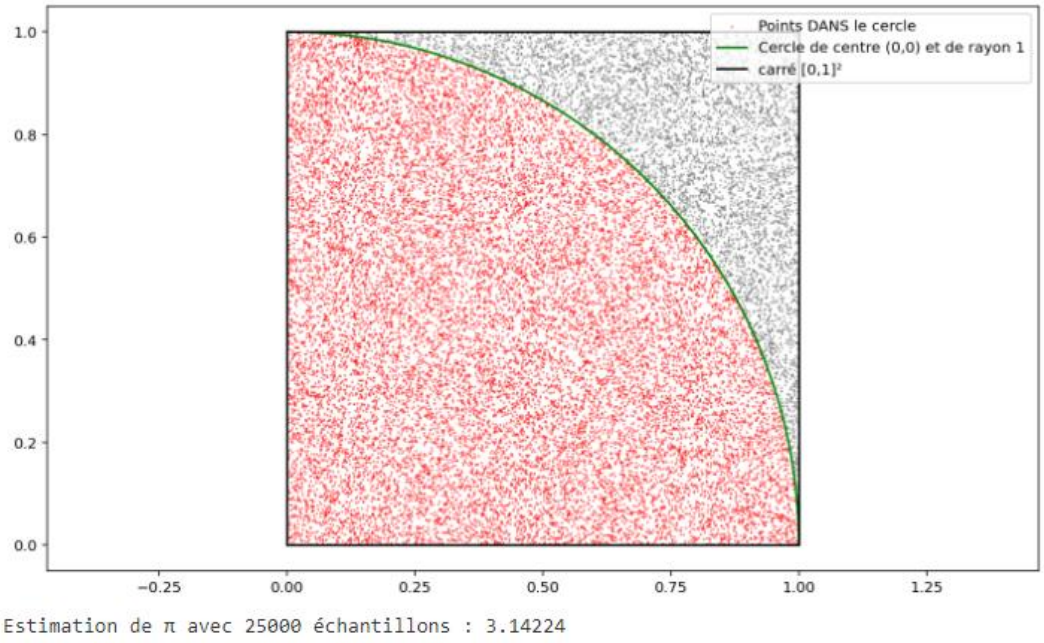
Pour 100 000 échantillons :

```
Out[10]: Estimation de π avec 100000 échantillons : 3.1402
Out[11]: Estimation de π avec 100000 échantillons : 3.146
Out[12]: Estimation de π avec 100000 échantillons : 3.13688
```

Pour 10 000 000 échantillons (100 fois plus que précédemment) :

```
Out[5]: Estimation de π avec 10000000 échantillons : 3.1426644
Out[6]: Estimation de π avec 10000000 échantillons : 3.142394
Out[7]: Estimation de π avec 10000000 échantillons : 3.1421044
```

Out[28]:



Estimation de π avec 25000 échantillons : 3.14224

Figure 1 : *Graphique représentatif de la démarche*

Observations :

Même pour 10 millions de points tirés, la valeur de π obtenue par la méthode naïve reste loin de la valeur exacte : sur les 3 essais affichés ici, seules les 2 premières décimales sont correctes ($\pi = 3.14159265\ldots$) et l'erreur est d'environ 7.95×10^{-4} en moyenne.

Pour 100 000 de points tirés, la valeur obtenue est encore moins précise que pour 10 millions de points tirés : ici aussi, seules les 2 premières décimales sont correctes mais la démarche permet de déterminer moins de ces décimales (3 à 5 contre 6 ou 7

précédemment). De plus, l'erreur moyenne pour ce nombre de points est d'environ 3.5×10^{-3}, soit de l'ordre de 10 fois plus importante que pour 10 millions de points.

En conclusion, en multipliant le nombre d'essais par 100, on divise l'erreur par 10.

C. Intégrale d'une fonction de signe quelconque :

Soit $f(x)$ une fonction continue définie sur l'intervalle $[a, b]$. L'intégrale de f sur cet intervalle est donnée par : $I = \int_a^b f(x)dx$. On peut se servir de la méthode naïve de Monte-Carlo pour estimer I à l'aide de points aléatoires.

Nous testerons la démarche pour 4 fonctions distinctes :

- $f(x) = 2x$ entre -3 et 10 où la fonction change de signe et l'intégrale n'est pas nulle ;
- $g(x) = \frac{1}{x}$ entre 1 et e où la fonction est strictement positive ;
- $h(x) = sin(x)$ entre 0 et 2π où la fonction change de signe et l'intégrale est nulle ;
- $v(x) = 3x^2$ entre -5 et 10 où la fonction est positive.

Nous avons stratégiquement choisi d'utiliser ces 4 fonctions sur des intervalles particuliers car nous savons déjà déterminer les valeurs de ces intégrales à la main. Cela nous permettra de comparer les résultats de la méthode naïve de Monte-Carlo aux valeurs exactes afin de discuter de l'erreur pour cette démarche.

Nous réutiliserons ces 4 mêmes fonctions pour tester et observer les résultats de la méthode non-naïve de Monte-Carlo et de celles de rectangles.

Le programme Python pour estimer l'intégrale d'une fonction de signe quelconque est alors le suivant :

```
import math
import random
import matplotlib.pyplot as plt

# Fonctions à tester :

def f(x):
    return 2*x

def g(x):
    return 1/x

def h(x):
    return math.sin(x)

def v(x):
    return 3*x**2
```

```
# Programme pour calculer le max et le min de la fonction étudiée (entre les bornes d'intégration) :

def max_fonction(f,k,a,b): # Dépend de la fonction étudiée, du pas de balayage, et des bornes d'intégration (a<b)
    x = a
    maxi = f(x)
    while x <= b :
        x += k # On regarde les valeurs de la fonction pour des x espacés de k sur [a,b]
        image = f(x)
        if image > maxi :
            maxi = image # On met à jour la valeur du max
    return (maxi)

def min_fonction(f,k,a,b): # On refait la même chose pour trouver le minimum
    x = a
    mini = f(x)
    while x <= b :
        x += k
        image = f(x)
        if image < mini :
            mini = image
    return (mini)


# Programme pour intégrer une fonction sur [a,b] selon la méthode de Monte-Carlo naïve

def monte_carlo_naive(f,a,b,N,k):
        # Bornes du domaine des y
    c = min_fonction(f,k,a,b) - 1 # On crée une marge de 1 pour être sûr de bien contenir la fonction
    d = max_fonction(f,k,a,b) + 1

    if c > 0 and d > 0 :
        c = 0
    elif d < 0 and c < 0 :
        d = 0
    # Quand le min est > 0, les points ne tomberont jamais entre 0 et min et donc on perd le rectangle [a,b]x[0, min] dans le calcul de l'intégrale ; pareil quand max < 0 ; ici, on prévient ces cas-là

        # Initialisation des comptes
```

```
    positive_area = 0
    negative_area = 0
    total_points = 0

    for i in range(N):
        x = random.uniform(a,b)
        y = random.uniform(c,d) # On pioche des points (x,y) aléatoirement dans le
rectangle [a,b]x[c,d]
        total_points += 1

        if y >= 0 and y <= f(x) : # Vérifier si le point est au-dessus de l'axe des x et en-
dessous de la courbe de f
            positive_area += 1
        elif y < 0 and y >= f(x) : # Vérifier l'inverse
            negative_area += 1

        # Calcul approché des aires positive et négative séparément
    positive_area_approx = (positive_area / total_points)*(b-a)*(d-c)
    negative_area_approx = (negative_area / total_points)*(b-a)*(d-c)

        # Calcul approché final de l'intégrale totale
    integral_approx = positive_area_approx - negative_area_approx

    print ("Min des y :", c, "; Max des y :", d, "; Pas de balayage :", k)
    print ("Intégrale entre", a, "et", b, "pour", N, "tirages :", integral_approx)

    # Test avec nos différentes fonctions
N = 1000000 # Nombre de points à générer
k = 0.0001 # On pose cette valeur car elle permet des valeurs suffisamment
approchées du max et du min

monte_carlo_naive(f,-3,10,N,k) # Doit donner 91
monte_carlo_naive(g,1,math.exp(1),N,k) # Doit donner 1
monte_carlo_naive(h,0,2*math.pi,N,k) # Doit donner 0
monte_carlo_naive(v,-5,10,N,k) # Doit donner 1125
```

Résultats : Voici les résultats de la démarche pour les 4 fonctions à tester avec 1 million de points aléatoires.

- $f(x) = 2x$:

Out[17]: Min des y : -7 ; Max des y : 21.000199999983877 ; Pas de balayage : 0.0001
Intégrale entre -3 et 10 pour 1000000 tirages : 91.02249015594758

- $g(x) = \frac{1}{x}$:

Out[18]: Min des y : -0.63212301806295 ; Max des y : 2.0 ; Pas de balayage : 0.0001
Intégrale entre 1 et 2.718281828459045 pour 1000000 tirages : 0.9978497328512861

- $h(x) = sin(x)$:

Out[19]: Min des y : -1.9999999999392841 ; Max des y : 1.9999999999932538 ; Pas de balayage : 0.0001
Intégrale entre 0 et 6.283185307179586 pour 1000000 tirages : -0.017995042719459065

- $v(x) = 3x^2$:

Out[23]: Min des y : -1.0 ; Max des y : 301.00600002950307 ; Pas de balayage : 0.0001
Intégrale entre -5 et 10 pour 1000000 tirages : 1126.0897723100081

Graphiques représentatifs : Pour 100 000 points aléatoires.

Tous les graphiques suivants de ce document ont été générés avec Python à l'aide des librairies *matplotlib.pyplot* et *numpy*.

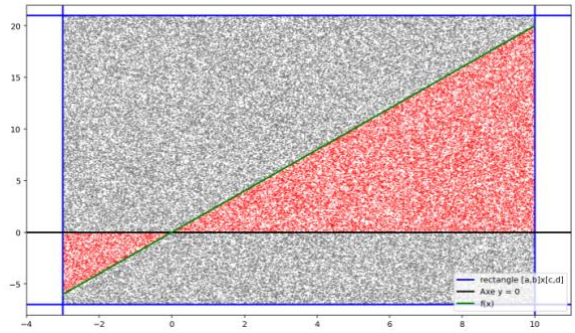


Figure 2 : *Graphique de* $f(x) = 2x$

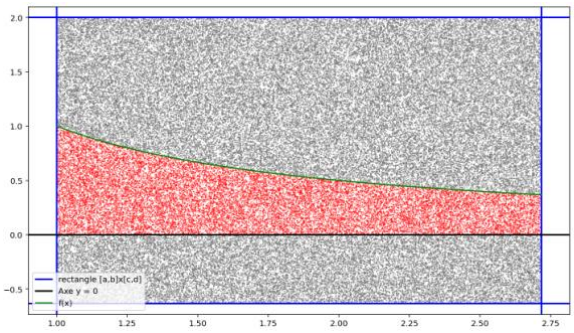


Figure 3 : *Graphique de* $g(x) = \frac{1}{x}$

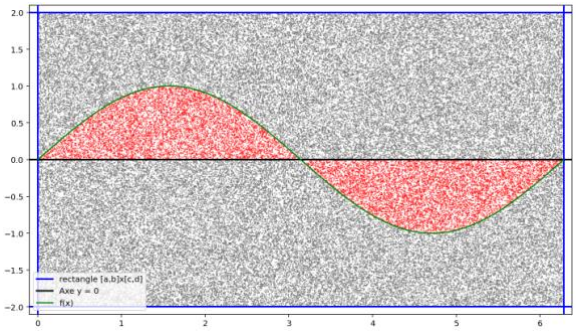


Figure 4 : *Graphique de* $h(x) = sin(x)$

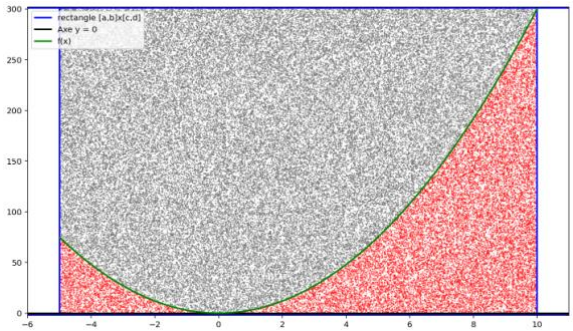


Figure 5 : *Graphique de* $v(x) = 3x^2$

Observations :

La méthode naïve de Monte-Carlo est rapide à réaliser pour une fonction positive aux variations peu complexes. Pour ces cas-là, elle consiste principalement à générer un certain nombre de points aléatoires dans le plan et compter le nombre de points tombant sous la courbe de la fonction.

Cependant, elle peut nécessiter un grand nombre de tirages pour atteindre une bonne précision, ce qui augmente sa complexité de mise en œuvre. Effectivement, pour les 4 résultats retenus plus haut, avec 1 million de points aléatoires, le programme a su approcher au mieux les valeurs exactes des intégrales à seulement 10^{-3} près (pour la fonction g) et est même allé jusqu'à obtenir une estimation d'intégrale erronée de plus d'une unité (pour la fonction v)

De plus, pour les fonctions dont les variations sont plus complexes, certaines étapes de la démarche deviennent plus compliquées à réaliser, comme le calcul des extrema pour déterminer le domaine des y. Contrairement aux fonctions simples en variation, comme $f(x) = 2x$, pour lesquelles on peut facilement trouver les extrema locaux à la main, pour d'autres fonctions aux variations moins régulières, on devra estimer les extrema par balayage. Cela dégrade d'autant plus la précision du résultat final de la démarche.

III. Méthode Non Naïve de Monte-Carlo

A. Principe de la méthode

La méthode de Monte-Carlo est une technique de simulation utilisée pour estimer des valeurs numériques par l'échantillonnage aléatoire. Dans le cadre de la méthode non naïve de Monte-Carlo, nous utiliserons une loi uniforme comme exécuté pour la méthode naïve.

Contrairement à la méthode naïve, on tirera des points aléatoires sur un segment au lieu d'un rectangle.

Il faut noter que pour améliorer l'efficacité et la précision de nos estimations, il est possible d'utiliser des techniques avancées telles que l'échantillonnage d'importance ou la stratification.

L'échantillonnage d'importance modifie la distribution des échantillons pour se concentrer dans les régions où la fonction a des valeurs significatives, augmentant ainsi la précision de l'estimation et réduisant la variance.

La stratification divise l'intervalle d'intégration en sous-intervalles, générant des échantillons indépendamment dans chaque strate pour garantir une représentation uniforme de l'intégrale totale, réduisant également la variance.

Ces techniques avancées améliorent l'efficacité et la précision des estimations de Monte-Carlo, mais sont plus compliquées et longues à coder. On a donc préféré utiliser une loi uniforme

.

Explication simplifiée avec une liste des étapes :

Pour illustrer cette méthode, considérons l'estimation d'une intégrale $\int_a^b f(x)dx$ à l'aide de la méthode de Monte-Carlo. Voici les étapes :

1. **Définir l'intervalle d'intégration** en choisissant les bornes a et b de l'intégrale
2. **Générer n points aléatoires** x_i dans l'intervalle $[a, b]$
3. **Calculer la valeur de la fonction en chaque point** x_i pour obtenir $f(x_i)$
4. **Calculer la moyenne** des valeurs de la fonction en ces points
5. **Estimer l'intégrale** en utilisant la formule suivante :

$$\int_a^b f(x)dx = \frac{(b-a)}{n}\sum_{i=1}^{n} f(x_i)$$

B. Test pour plusieurs fonctions de signe quelconque

Le programme suivant met en oeuvre la démarche non-naïve de Monte-Carlo pour approcher les intégrales des 4 mêmes fonctions étudiées avec la méthode naïve précédemment, pour les mêmes intervalles d'intégration :

```python
import math
import random
from scipy import integrate as intg # Servira à calculer la valeur exacte de l'intégrale plus tard


# Fonctions à tester :

def f(x):
    return 2*x

def g(x):
    return 1/x

def h(x):
    return math.sin(x)

def v(x):
    return 3*x**2

# Calculer l'intégrale d'une fonction f entre a et b par la méthode de Monte-Carlo non-naïve pour nb_ech échantillons :

def monte_carlo_non_naive(f,a,b,nb_ech):
    somme_fx = 0

    for s in range(nb_ech):
        x = random.uniform(a,b) # On choisit aléatoirement les x entre les bornes a et b
        somme_fx += f(x) # On somme les f(x) entre eux

    # On calcule une valeur approchée de l'intégrale
    integrale_approx = (somme_fx / nb_ech)*(b-a)

    print("Intégrale entre", a, "et", b, "selon la méthode non-naïve pour", nb_ech, "échantillons :", integrale_approx)

    # Calcul de la valeur exacte pour comparer
    integrale, err = intg.quad(f,a,b) # La fonction quad renvoie 2 résultats : l'intégrale et l'erreur
    print ("Valeur exacte :", integrale)

monte_carlo_non_naive(f,-3,10,1000000) # Doit donner 91
monte_carlo_non_naive(g,1,math.exp(1),1000000) # Doit donner 1
monte_carlo_non_naive(h,0,2*math.pi,1000000) # Doit donner 0
```

```
monte_carlo_non_naive(v,-5,10,1000000) # Doit donner 1125
```

Résultats : Avec 1 million de points aléatoires.

- $f(x) = 2x$:

```
Out[5]: Intégrale entre -3 et 10 selon la méthode non-naïve pour 1000000 échantillons : 91.02353080413954
        Valeur exacte : 91.0
```

- $g(x) = \frac{1}{x}$:

```
Out[6]: Intégrale entre 1 et 2.718281828459045 selon la méthode non-naïve pour 1000000 échantillons : 0.9999258243393271
        Valeur exacte : 0.9999999999999998
```

- $h(x) = sin(x)$:

```
Out[7]: Intégrale entre 0 et 6.283185307179586 selon la méthode non-naïve pour 1000000 échantillons :
        0.0023482678205142497
        Valeur exacte : 2.221501482512777e-16
```

- $v(x) = 3x^2$:

```
Out[8]: Intégrale entre -5 et 10 selon la méthode non-naïve pour 1000000 échantillons : 1127.7368505522215
        Valeur exacte : 1125.0
```

Graphiques représentatifs : Pour 50 points aléatoires

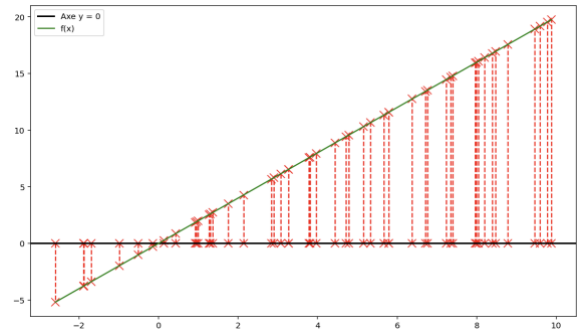


Figure 6 : $f(x) = 2x$

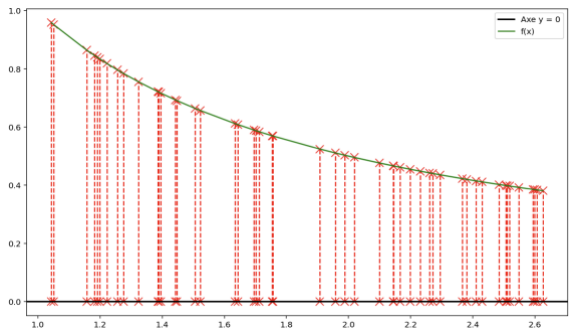


Figure 7 : $g(x) = \frac{1}{x}$

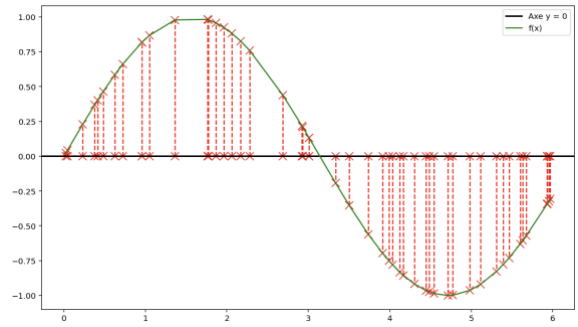


Figure 8 : $h(x) = sin(x)$

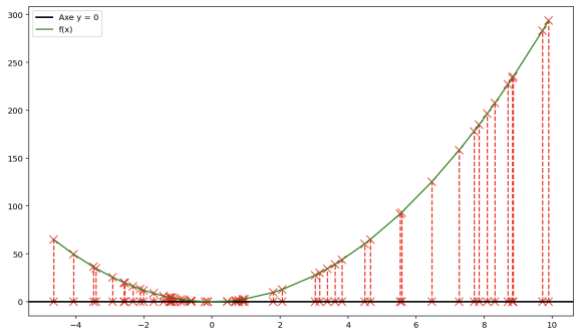


Figure 9 : $v(x) = 3x^2$

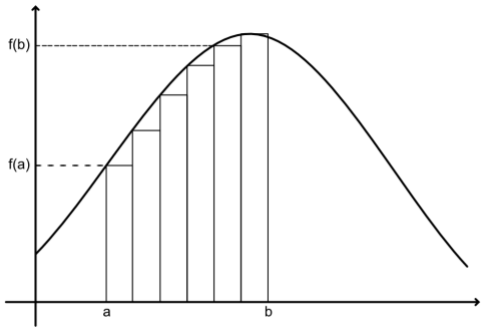
Observations :

La méthode non naïve de Monte-Carlo utilise un échantillonnage aléatoire uniforme pour estimer les intégrales, optimisant ainsi la précision et réduisant la variance des estimations. Les résultats montrent que cette méthode fournit des estimations proches des valeurs exactes même avec un nombre limité de points. Les graphiques démontrent une distribution uniforme des points, assurant une couverture complète de la fonction à intégrer. Cette méthode est plus efficace et précise que la méthode naïve.

IV. Méthode des rectangles

A. Principe de la méthode

La méthode des rectangles est une approche classique pour calculer des intégrales en découpant l'intervalle d'intégration en petits segments (voir image ci-contre) et en approximant l'aire sous la courbe par des rectangles. Elle offre une introduction intuitive au calcul d'intégrales mais peut être imprécise pour les fonctions complexes.



Comparer la méthode de Monte-Carlo à celle des rectangles permet de mettre en lumière les différences de précision et d'introduire les concepts statistiques sous-jacents à la méthode de Monte-Carlo. La méthode des rectangles est similaire à celle de Monte-Carlo non naîve en ce qu'elle prend une moyenne des points, mais elle utilise des points équidistants plutôt que des points aléatoires, illustrant ainsi leur applicabilité respective à des problèmes réels.

B. Exemple d'utilisation de la méthode

```
import math

# Fonctions à tester :

def f(x):
    return 2*x

def g(x):
    return 1/x

def h(x):
    return math.sin(x)

def v(x):
    return 3*x**2


def rectangle_integration(f, a, b, n):
    # Largeur de chaque rectangle
    L = (b - a) / n

    # Initialisation de la somme des aires des rectangles
    total_area = 0

    # Calcul de l'aire de chaque rectangle et ajout à la somme totale
    for i in range(n):
        x_left = a + i * L
        x_right = a + (i + 1) * L
        height = f(x_left)  # On aurait pu calculer f(x_right) à la place
        area = height * L
        total_area += area

    print("Intégrale entre", a, "et", b, "avec", n, "rectangles :", total_area)


# Calcul de l'approximation de l'intégrale des 4 fonctions pour 1 million de rectangles
rectangle_integration(f,-3,10,1000000) # Doit donner 91
rectangle_integration(g,1,math.exp(1),1000000) # Doit donner 1
rectangle_integration(h,0,2*math.pi,1000000) # Doit donner 0
rectangle_integration(v,-5,10,1000000) # Doit donner 1125
```

Résultats : Pour 1 million de rectangles.

- $f(x) = 2x$:

```
Out[23]: Intégrale entre -3 et 10 avec 1000000 rectangles : 90.99983100000001
```

- $g(x) = \frac{1}{x}$:

```
Out[22]: Intégrale entre 1 et 2.718281828459045 avec 1000000 rectangles : 1.0000005430808407
```

- $h(x) = sin(x)$:

```
Out[21]: Intégrale entre 0 et 6.283185307179586 avec 1000000 rectangles : 1.6976962142473453e-16
```

- $v(x) = 3x^2$:

```
Out[20]: Intégrale entre -5 et 10 avec 1000000 rectangles : 1124.998312501682
```

Graphiques représentatifs : Pour 50 rectangles

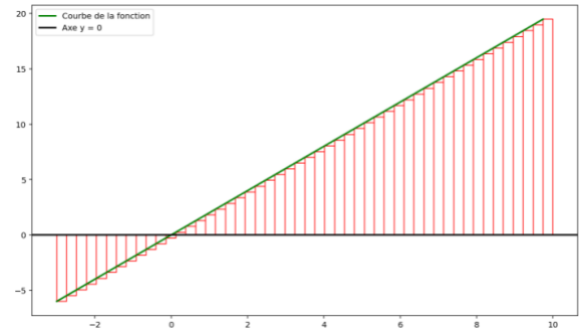


Figure 10 : $f(x) = 2x$

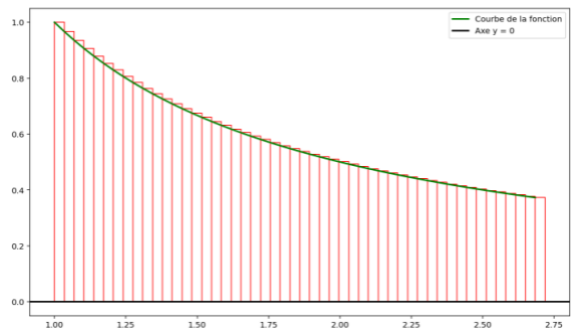


Figure 11 : $g(x) = \frac{1}{x}$

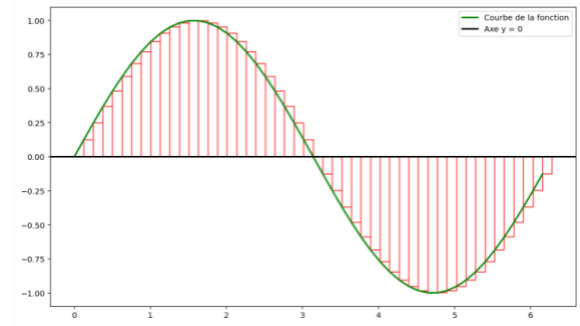


Figure 12 : $h(x) = sin(x)$

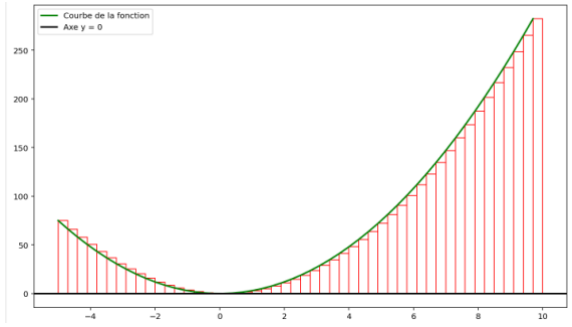


Figure 13 : $v(x) = 3x^2$

Observations :

La méthode des rectangles est généralement considérée comme plus simple que les deux précédentes à mettre en œuvre car elle repose sur une découpe régulière de la région d'intégration en rectangles de taille égale. Elle fournit aussi des résultats visiblement plus précis que les méthodes de Monte-Carlo. Nous verrons en étudiant et comparant les niveaux d'erreurs de chacune des méthodes en quoi cela est vrai.

V. Comparaison des méthodes

A. Analyse des erreurs

Pour cette étape, le code utilisé est une combinaison des précédents codes de calcul d'intégrales avec les trois méthodes, enrichi d'un nouveau segment pour étudier les erreurs. Pour éviter que le programme soit trop long, nous avons choisi de ne présenter que les parties nouvelles du code qui traitent des erreurs.

```
# Paramètres
a = 0
b = np.pi
c = 0
d = 1  # Valeur maximale de sin(x) sur [0, pi]
exact_integral = 2  # Valeur exacte de l'intégrale de sin(x) entre 0 et PI
N_values = np.arange(10000)  # Nombre d'échantillons pour tester

# Calcul des intégrales et des erreurs pour chaque méthode
monte_carlo_errors_nn = []
naive_errors = []
rectangle_errors = []

for N in N_values:
    monte_carlo_integral_non_naive = monte_carlo_integration(sin_function, a, b, N)
    monte_carlo_integral_naive = monte_carlo_naive(sin_function, a, b, c, d, N)
    rectangle_integral = rectangle_integration(sin_function, a, b, N)

    monte_carlo_errors_nn.append(np.abs(monte_carlo_integral_non_naive -
exact_integral))
    naive_errors.append(np.abs(monte_carlo_integral_naive - exact_integral))
    rectangle_errors.append(np.abs(rectangle_integral - exact_integral))

# Calcul des moyennes des erreurs
moy_nn = np.nanmean(monte_carlo_errors_nn)
moy_naive = np.nanmean(naive_errors)
moy_rectangle = np.nanmean(rectangle_errors)

print(f"Moyenne des erreurs Monte-Carlo Non Naïve : {moy_nn}")
print(f"Moyenne des erreurs Monte-Carlo Naïve : {moy_naive}")
print(f"Moyenne des erreurs Rectangles : {moy_rectangle}")
```

```
print(f"Estimation Monte-Carlo Naïve pour N={N_values[-1]} :
{monte_carlo_naive(sin_function, a, b, c, d, N_values[-1])}")
```

Résultats :

```
Moyenne des erreurs Monte Carlo Non Naïve : 0.015572422181813176
Moyenne des erreurs Monte Carlo Naïve : 0.023666813417124573
Moyenne des erreurs Rectangles : 0.0003084463903258344
Estimation Monte Carlo Naïve pour N=9999 : 1.9963676088518398
```

Observations :

Dans l'analyse des erreurs, il est immédiatement évident que la méthode des rectangles est beaucoup plus précise et fiable que les deux autres méthodes. Son erreur est de l'ordre de 100 fois plus petite que pour les méthodes de Monte-Carlo. Pour $N = 10000$, les erreurs de la méthode des rectangles sont de $\frac{1}{10000}$ comparées à $\frac{1}{100}$ pour Monte-Carlo ($\frac{1}{\sqrt{}}$) d'où le facteur 100 mentionné. Si N est différent, le coefficient changera également.

La différence d'erreurs entre la méthode naïve et la méthode non naïve de Monte-Carlo n'est pas significative. Nous comparerons donc les variances de ces deux méthodes. Cette comparaison n'est pas applicable à la méthode des rectangles car elle n'implique pas d'éléments aléatoires.

.

B. Analyse de la variance

Pour cette étape, le code utilisé est aussi une combinaison des précédents codes de calcul d'intégrales avec les trois méthodes, enrichi d'un nouveau segment pour étudier les variances. Pour éviter que le programme soit trop long, nous avons choisi de ne présenter que les parties nouvelles du code qui traitent des variances.

```
def calculate_variance(method, M, n, exact_value):
    """Calcul de la variance des estimations par rapport à la valeur exacte."""
    estimates = [method(n) for _ in range(M)]
    mean_estimate = sum(estimates) / M
    variance = sum((estimate - exact_value) ** 2 for estimate in estimates) / M
    return variance

# Nombre de points pour chaque estimation (puissances de 2)
points = [2**i for i in range(8)]  # De 1 à 128

# Nombre de calculs pour estimer la variance
M = 100

# Valeur exacte de l'intégrale de sin(x) de 0 à pi
exact_value = 2

# Calcul des variances pour chaque méthode
variances_naive = [calculate_variance(monte_carlo_naive, M, n, exact_value) for n in
points]
variances_non_naive = [calculate_variance(monte_carlo_non_naive, M, n,
exact_value) for n in points]

# Affichage des résultats
for n, var_naive, var_non_naive in zip(points, variances_naive, variances_non_naive):
print(f"Nombre de points: {n}, Variance Naïve: {var_naive:.5f}, Variance Non-Naïve:
{var_non_naive:.5f}")
```

Résultats :

```
Nombre de points: 1, Variance Naïve: 2.32800, Variance Non-Naïve: 0.67660
Nombre de points: 2, Variance Naïve: 1.04697, Variance Non-Naïve: 0.44681
Nombre de points: 4, Variance Naïve: 0.59824, Variance Non-Naïve: 0.34017
Nombre de points: 8, Variance Naïve: 0.31385, Variance Non-Naïve: 0.24391
Nombre de points: 16, Variance Naïve: 0.15243, Variance Non-Naïve: 0.20310
Nombre de points: 32, Variance Naïve: 0.07717, Variance Non-Naïve: 0.18532
Nombre de points: 64, Variance Naïve: 0.04022, Variance Non-Naïve: 0.19434
Nombre de points: 128, Variance Naïve: 0.01442, Variance Non-Naïve: 0.18604
```

Observations :

La méthode Monte-Carlo naïve montre une amélioration constante et significative avec l'augmentation du nombre de points, tandis que la méthode non naïve présente une variance initialement plus faible que la méthode naïve, mais elle a tendance à stagner même si le nombre de points augmente. Cela suggère que, bien que la méthode non-naïve soit plus efficace pour des petits échantillons, la méthode naïve peut la surpasser en précision à mesure que le nombre de points augmente.

Conclusion

L'objectif de nos travaux pratiques était de découvrir les méthodes aléatoires pour calculer une intégrale. Nous avons choisi de nous concentrer sur les méthodes de Monte-Carlo, en gardant à l'esprit qu'il existe d'autres méthodes pour calculer une intégrale, aléatoires ou non, comme la méthode Simpson et la méthode des rectangles.

Nos travaux reposent principalement sur l'exécution des méthodes de Monte-Carlo et de la méthode des rectangles à qui on les compare, grâce à la programmation sur Python. Cet exercice nous a permis de mieux nous approprier ces méthodes pour ensuite mieux les observer et critiquer sous différents aspects comme la facilité de mise en œuvre, le temps de calcul, la variance et l'erreur.

Il en est ressorti que les méthodes de Monte-Carlo sont effectivement utiles aux calculs d'intégrales de fonctions, surtout pour les fonctions qu'on ne sait pas intégrer à la main. Cependant, les méthodes de Monte-Carlo, naïve comme non naïve, sont facilement surpassées par d'autres méthodes d'intégration. Par exemple, ici, on a remarqué, par les tests et l'analyse, que la méthode des rectangles est généralement la plus intuitive et facile pour calculer une intégrale car elle nécessite uniquement des calculs de base à partir de boucles et opérations arithmétiques simples, et, contrairement aux méthodes de Monte-Carlo, elle n'exige pas de génération de nombres aléatoires ni de compréhension approfondie des distributions de probabilité.