
Méthodes et simulations numériques

I. Introduction

En mathématiques, de nombreuses quantités se calculent par une formule exacte : primitive d'une fonction, solution d'une équation, espérance d'une variable aléatoire... Mais il arrive souvent qu'aucune formule simple n'existe, ou qu'elle soit trop coûteuse à évaluer. On se contente alors d'une **valeur approchée**, obtenue par un algorithme.

On parle de **méthode numérique** lorsqu'on approche une quantité déterministe (une intégrale, une racine d'équation, etc.) par un calcul fini. On parle de **simulation** lorsqu'on reproduit une expérience aléatoire sur l'ordinateur, afin d'estimer une probabilité, une espérance ou une variance.

Nous avons déjà rencontré une méthode numérique au chapitre 8 : la **dichotomie** permet d'approcher une solution de l'équation $f(x) = 0$. Ce chapitre en présente d'autres, puis montre comment simuler les lois usuelles du cours de probabilités.

II. Méthodes numériques

1) Méthode des rectangles

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue. L'intégrale $\int_a^b f(t) dt$ s'interprète comme l'aire algébrique sous la courbe de f . Lorsque f n'admet pas de primitive élémentaire, on ne peut pas calculer cette aire par la formule

$$\int_a^b f(t) dt = F(b) - F(a)$$

On l'approche alors par une somme d'aires de rectangles.

Définition 1 : Subdivision régulière

Soit $n \in \mathbb{N}^*$. On pose $h = \frac{b-a}{n}$ et, pour tout $k \in \llbracket 0, n \rrbracket$,

$$x_k = a + kh$$

La suite finie $(x_0, x_1, \dots, x_n)$ s'appelle une **subdivision régulière** du segment $[a, b]$. Le réel h est le **pas** de la subdivision.

Sur chaque sous-intervalle $[x_k, x_{k+1}]$, de longueur h , on remplace f par une constante. Selon le point où

l'on évalue f , on obtient plusieurs variantes de la [méthode des rectangles](#).

Définition 2 : Méthode des rectangles

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue et $n \in \mathbb{N}^*$.

★ **Rectangles à gauche** : on prend la hauteur $f(x_k)$ sur $[x_k, x_{k+1}]$.

$$R_n^g = h \sum_{k=0}^{n-1} f(x_k) = \frac{b-a}{n} \sum_{k=0}^{n-1} f\left(a + k \frac{b-a}{n}\right)$$

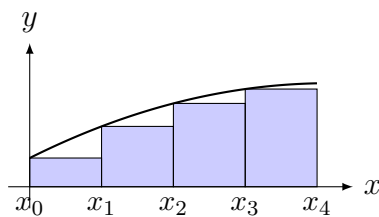
★ **Rectangles à droite** : on prend la hauteur $f(x_{k+1})$.

$$R_n^d = h \sum_{k=1}^n f(x_k) = \frac{b-a}{n} \sum_{k=1}^n f\left(a + k \frac{b-a}{n}\right)$$

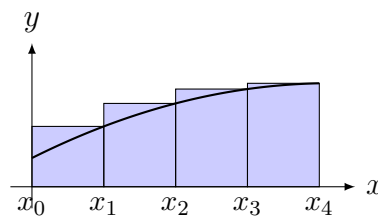
★ **Rectangles au point milieu** : on prend la hauteur au milieu de $[x_k, x_{k+1}]$.

$$R_n^m = h \sum_{k=0}^{n-1} f\left(x_k + \frac{h}{2}\right)$$

Lorsque n tend vers $+\infty$, ces trois sommes tendent vers $\int_a^b f(t) dt$. En pratique, plus n est grand, meilleure est l'approximation, au prix d'un plus grand nombre de calculs.



Rectangles à gauche



Rectangles à droite

Comme pour la dichotomie du chapitre 8, on peut passer une **fonction Python** en argument d'une autre fonction.

```
1 def rectangles_gauche(f, a, b, n):
2     """
3     Entrée : une fonction f, deux réels a et b, un entier n >= 1
4     Sortie : approximation de l'intégrale de f sur [a, b]
5               par la méthode des rectangles à gauche
6     """
7     h = (b - a) / n
8     s = 0
9     for k in range(n):
10         s = s + f(a + k * h)
11     return h * s
```

Les variantes à droite et au point milieu s'obtiennent en changeant le point d'évaluation :

```

1 def rectangles_droite(f, a, b, n):
2     h = (b - a) / n
3     s = 0
4     for k in range(1, n + 1):
5         s = s + f(a + k * h)
6     return h * s
7
8
9 def rectangles_milieu(f, a, b, n):
10    h = (b - a) / n
11    s = 0
12    for k in range(n):
13        s = s + f(a + (k + 0.5) * h)
14    return h * s

```

Exemple : On sait que $\int_0^1 \frac{4}{1+t^2} dt = \pi$. On peut donc approcher π sans connaître de primitive, en appliquant la méthode des rectangles à la fonction $f : t \mapsto \frac{4}{1+t^2}$ sur $[0, 1]$.

```

def f(t):
    return 4 / (1 + t ** 2)

>>> rectangles_gauche(f, 0, 1, 10)
3.239925988907233
>>> rectangles_gauche(f, 0, 1, 100)
3.1515759862321282
>>> rectangles_gauche(f, 0, 1, 1000)
3.1425924869231286

```

On rappelle que $\pi \approx 3,1415926535$. L'erreur diminue quand n augmente : elle est de l'ordre de 10^{-1} pour $n = 10$, de 10^{-2} pour $n = 100$ et de 10^{-3} pour $n = 1000$.

Remarque : Dans la boucle, on a écrit $s = s + f(a + k * h)$. On peut bien sûr utiliser l'écriture raccourcie $s += f(a + k * h)$.

2) Méthode des trapèzes

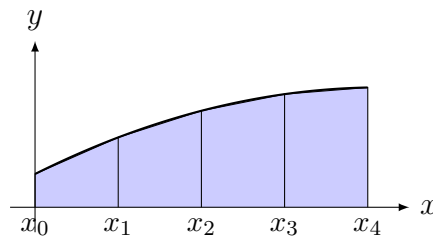
Au lieu de remplacer f par une constante sur chaque sous-intervalle, on peut la remplacer par le **segment** joignant les points $(x_k, f(x_k))$ et $(x_{k+1}, f(x_{k+1}))$. L'aire obtenue est alors celle d'un trapèze.

Définition 3 : Méthode des trapèzes

Soit $f : [a, b] \rightarrow \mathbb{R}$ une fonction continue et $n \in \mathbb{N}^*$. L'approximation de $\int_a^b f(t) dt$ par la **méthode des trapèzes** est

$$T_n = h \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f(x_k) \right)$$

où $h = \frac{b-a}{n}$ et $x_k = a + kh$.



Méthode des trapèzes

Remarque :

- ★ L'aire d'un trapèze de hauteurs $f(x_k)$ et $f(x_{k+1})$ et de largeur h vaut $h \cdot \frac{f(x_k) + f(x_{k+1})}{2}$.
En sommant sur k de 0 à $n-1$, les termes « intérieurs » $f(x_1), \dots, f(x_{n-1})$ apparaissent deux fois, d'où la formule.
- ★ On a $T_n = \frac{R_n^g + R_n^d}{2}$: la méthode des trapèzes est la **moyenne** des rectangles à gauche et à droite.

```

1 def trapezes(f, a, b, n):
2     """
3     Entrée : une fonction f, deux réels a et b, un entier n >= 1
4     Sortie : approximation de l'intégrale de f sur [a, b]
5               par la méthode des trapèzes
6     """
7     h = (b - a) / n
8     s = (f(a) + f(b)) / 2
9     for k in range(1, n):
10         s = s + f(a + k * h)
11     return h * s

```

Exemple : Reprenons l'approximation de $\pi \approx 3,1416$ par $\int_0^1 \frac{4}{1+t^2} dt$.

n	Rectangles à gauche	Trapèzes
10	3,2399	3,1399
100	3,1516	3,1416
1000	3,1426	3,1416

À n fixé, la méthode des trapèzes est nettement plus précise que celle des rectangles à gauche (ou à droite). Les rectangles au point milieu donnent, sur cet exemple, une précision comparable à celle des trapèzes.

3) Méthode de Newton

La méthode des rectangles et celle des trapèzes approchent une **intégrale**. Un autre problème numérique classique est la recherche d'une **racine** : on cherche x tel que $f(x) = 0$.

Nous avons déjà vu la **dichotomie** au chapitre 8. Elle s'applique dès que f est continue et change de signe sur $[a, b]$, et elle est très robuste : l'erreur est divisée par 2 à chaque étape.

Lorsque f est dérivable, la [méthode de Newton](#) (ou de Newton-Raphson) construit une suite qui converge en général beaucoup plus rapidement vers une racine.

Principe de la méthode de Newton

On part d'une valeur initiale x_0 proche d'une racine. Au point $(x_n, f(x_n))$, on trace la **tangente** à la courbe de f . Cette tangente coupe l'axe des abscisses en un point x_{n+1} , que l'on retient comme nouvelle approximation :

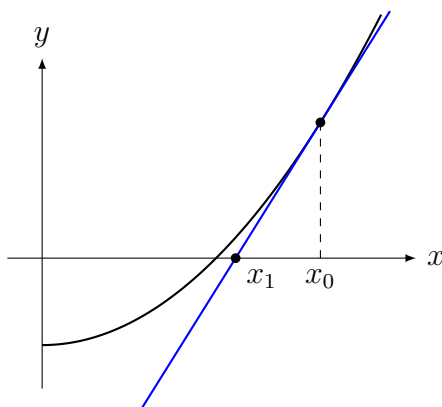
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

On s'arrête lorsque deux termes consécutifs sont suffisamment proches, c'est-à-dire lorsque $|x_{n+1} - x_n| < \varepsilon$.

La formule se justifie ainsi : l'équation de la tangente en x_n est

$$y = f(x_n) + f'(x_n)(x - x_n)$$

Elle coupe l'axe des abscisses lorsque $y = 0$, d'où $x = x_n - \frac{f(x_n)}{f'(x_n)}$, à condition que $f'(x_n) \neq 0$.



```
1 def newton(f, fprime, x, eps):
2     """
3     Entrée : une fonction f, sa dérivée fprime, une valeur initiale x
4             et une précision eps > 0
5     Sortie : une approximation d'une racine de f obtenue par
6             la méthode de Newton
7     """
8     delta = eps + 1
9     while delta > eps:
10         x_suivant = x - f(x) / fprime(x)
11         delta = abs(x_suivant - x)
12         x = x_suivant
13     return x
```

Exemple : Approchons $\sqrt{2}$ comme racine positive de $f(x) = x^2 - 2$. On a $f'(x) = 2x$. En partant de $x_0 = 1$ avec $\varepsilon = 10^{-6}$:

```
def f(x):
    return x ** 2 - 2

def fprime(x):
    return 2 * x
```

```
>>> newton(f, fprime, 1, 10**(-6))
1.4142135623730951
```

Les premiers termes de la suite sont $x_0 = 1$, $x_1 = 1,5$, $x_2 \approx 1,4167$, $x_3 \approx 1,4142$. Cinq itérations suffisent pour atteindre 10^{-6} . Pour la même précision, la dichotomie sur $[1, 2]$ nécessite 21 étapes.

La méthode de Newton n'est pas infaillible.

- ★ Il faut connaître une expression de f' et pouvoir l'implémenter.
- ★ Si $f'(x_n) = 0$, la formule n'a pas de sens (division par zéro).
- ★ Si x_0 est mal choisi, la suite peut diverger ou converger vers une autre racine que celle visée.
- ★ Contrairement à la dichotomie, on n'a pas de garantie a priori que l'algorithme s'arrête.



En pratique, on l'utilise lorsque f est « régulière » et que l'on dispose d'une valeur initiale raisonnable. La dichotomie reste préférable dès que l'on connaît seulement un changement de signe.

Comparaison des méthodes numériques

- ★ **Intégrale** $\int_a^b f$: méthode des rectangles (simple, moins précise) ou méthode des trapèzes (à peine plus coûteuse, souvent bien meilleure).
- ★ **Équation** $f(x) = 0$: dichotomie (lente mais sûre, chapitre 8) ou méthode de Newton (rapide si f' est connue et x_0 bien choisi).

III. Simulations de variables aléatoires

1) Nombres pseudo-aléatoires

Un ordinateur est déterministe : le même programme, lancé dans les mêmes conditions, produit toujours le même résultat. On ne peut donc pas tirer un « vrai » hasard. On se contente de générer une suite de nombres qui *imitent* le hasard, calculés par une formule (discrète et périodique). On parle de nombres [pseudo-aléatoires](#).

En Python, ces nombres sont fournis par le module `random`, déjà mentionné au chapitre 3.

Importation

Pour ce chapitre, on importera les deux fonctions dont on a besoin :

```
from random import random, randint
```

Définition 4 : Fonction *random*

L'appel `random()` renvoie un flottant de l'intervalle $[0, 1[$, suivant (une approximation de) la loi uniforme sur $[0, 1[$.

Cette loi a la propriété suivante : pour tout intervalle $I \subset [0, 1[$,

$$P(U \in I) = \text{longueur de } I$$

```
>>> random()
0.38476731167450007
>>> random()
0.5958884283611281
>>> random()
0.46804988778467727
```

Remarque : La loi uniforme sur un intervalle réel n'est étudiée en mathématiques qu'en seconde année. On retient seulement la propriété sur les longueurs, qui suffit à justifier les simulations de ce chapitre.

Définition 5 : Fonction *randint*

L'appel `randint(a, b)` renvoie un entier choisi selon la loi uniforme sur $\llbracket a, b \rrbracket$: chaque entier entre a et b **inclus** a la même probabilité $\frac{1}{b - a + 1}$ d'apparaître.

```
>>> randint(1, 6)
4
>>> randint(1, 6)
1
>>> randint(1, 6)
6
```



Contrairement à `range(a, b)` qui s'arrête **avant** b , la fonction `randint(a, b)` inclut b . Ainsi `randint(1, 6)` simule bien un dé à six faces.

On peut reconstruire `randint` à partir de `random` uniquement. L'idée est de découper $[0, 1[$ en $b - a + 1$ intervalles de même longueur : si U tombe dans le k^{e} d'entre eux, on renvoie $a + k$.

Lien entre random et randint

Si U suit la loi uniforme sur $[0, 1[$ et si $a \leq b$ sont entiers, alors

$$\lfloor (b - a + 1)U + a \rfloor$$

suit la loi uniforme sur $\llbracket a, b \rrbracket$.

```
1 from math import floor
2 from random import random
3
4 def mon_randint(a, b):
5     """
6     Entrée : deux entiers a et b
7     Sortie : un entier uniforme entre a et b inclus, simulé à partir de random
8     """
9     return floor((b - a + 1) * random() + a)
```

En pratique, on utilisera directement `randint` du module `random`. L'intérêt de cette reconstruction est de comprendre que **toute** simulation de ce chapitre se ramène, in fine, à des appels de `random()`.

2) Lois usuelles

On rappelle les modèles vus en probabilités, puis on en donne une simulation.

Définition 6 : Loi uniforme discrète

On dit que X suit la [loi uniforme](#) sur $\llbracket a, b \rrbracket$ lorsque

$$X(\Omega) = \llbracket a, b \rrbracket \quad \text{et} \quad \forall k \in \llbracket a, b \rrbracket, \quad P(X = k) = \frac{1}{b - a + 1}$$

C'est exactement ce que fait `randint(a, b)`. En particulier, `randint(1, n)` simule un n -dé équilibré.

Définition 7 : Loi de Bernoulli

On dit que X suit la [loi de Bernoulli](#) de paramètre $p \in [0, 1]$ lorsque X ne prend que les valeurs 0 et 1, avec $P(X = 1) = p$ et $P(X = 0) = 1 - p$.

On note $X \hookrightarrow \mathcal{B}(p)$.

Pour simuler X , on tire U uniforme sur $[0, 1[$. L'événement $[U < p]$ a pour probabilité p , puisque c'est un intervalle de longueur p . On renvoie alors 1 si $U < p$, et 0 sinon.

```
1 def bernoulli(p):
2     """
3     Entrée : un réel p dans [0, 1]
4     Sortie : une simulation d'une variable de Bernoulli de paramètre p
5     """
6     if random() < p:
7         return 1
8     else:
9         return 0
```

```
>>> bernoulli(0.3)
0
>>> bernoulli(0.3)
1
>>> bernoulli(0.5)
1
```

Définition 8 : Loi binomiale

On dit que X suit la **loi binomiale** de paramètres $n \in \mathbb{N}$ et $p \in [0, 1]$ lorsque X compte le nombre de succès parmi n épreuves de Bernoulli indépendantes de même paramètre p .

On a alors $X(\Omega) = \llbracket 0, n \rrbracket$ et $P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$. On note $X \hookrightarrow \mathcal{B}(n, p)$.

D'après le cours de probabilités, X a même loi que $X_1 + \dots + X_n$ où les X_i sont des Bernoulli indépendantes de paramètre p . On simule donc une binomiale en additionnant n appels de `bernoulli(p)`.

```
1 def binomiale(n, p):
2     """
3     Entrée : un entier n et un réel p dans [0, 1]
4     Sortie : une simulation d'une variable binomiale de paramètres n et p
5     """
6     s = 0
7     for _ in range(n):
8         s = s + bernoulli(p)
9     return s
```

```
>>> binomiale(10, 0.5)
6
>>> binomiale(10, 0.5)
4
>>> binomiale(10, 0.5)
7
```

Remarque :

- ★ Chaque appel de `bernoulli` (donc chaque appel de `random`) est indépendant des précédents. La somme suit bien une loi binomiale.
- ★ Le cas $n = 1$ redonne la loi de Bernoulli : $\mathcal{B}(1, p) = \mathcal{B}(p)$.

3) Simulation à l'aide de la fonction de répartition

Les trois lois précédentes ont une simulation « naturelle » (uniforme, test $U < p$, somme de Bernoulli). Pour une variable aléatoire X à valeurs dans un ensemble **fini** quelconque, on dispose d'une méthode générale, fondée sur la fonction de répartition.

Définition 9 : Fonction de répartition

La **fonction de répartition** de X est la fonction F_X définie par

$$\forall t \in \mathbb{R}, \quad F_X(t) = P(X \leq t)$$

Supposons que X prenne les valeurs $x_0, x_1, \dots, x_{m-1}$ avec $P(X = x_i) = p_i$, où $p_0 + \dots + p_{m-1} = 1$. On

partitionne l'intervalle $[0, 1[$ en m créneaux successifs de longueurs $p_0, p_1, \dots, p_{m-1}$:

$$[0, p_0[\quad [p_0, p_0 + p_1[\quad \dots \quad [p_0 + \dots + p_{m-2}, 1[$$

On tire U uniforme sur $[0, 1[$, et l'on renvoie x_i si U tombe dans le i^{e} créneau. Ainsi $P(\text{on renvoie } x_i) = p_i$.

En pratique, on parcourt les probabilités en les additionnant jusqu'à dépasser U :

```
1 def simule(valeurs, probas):
2     """
3     Entrée : une liste valeurs (univers image) et une liste probas
4             de même longueur, avec sum(probas) = 1
5     Sortie : une simulation de la variable aléatoire qui prend
6             la valeur valeurs[i] avec la probabilité probas[i]
7     """
8     u = random()
9     s = 0
10    for i in range(len(probas)):
11        s = s + probas[i]
12        if u < s:
13            return valeurs[i]
14    return valeurs[-1]
```

Exemple : Un jeu rapporte 0, 10 ou 50 euros, avec les probabilités $\frac{7}{10}$, $\frac{2}{10}$ et $\frac{1}{10}$. Pour simuler le gain :

```
>>> simule([0, 10, 50], [0.7, 0.2, 0.1])
0
>>> simule([0, 10, 50], [0.7, 0.2, 0.1])
10
```

Si par exemple $U = 0,85$, on a $0,85 \geq 0,7$ mais $0,85 < 0,7 + 0,2 = 0,9$, donc on renvoie 10.

Remarque :

- ★ Le dernier `return valeurs[-1]` protège des erreurs d'arrondi : si la somme des p_i vaut 0,999999 au lieu de 1 et que U est très proche de 1, la boucle pourrait se terminer sans rien renvoyer. Mais en théorie, on pourrait ne pas le mettre.
- ★ La loi de Bernoulli est un cas particulier : `simule([0, 1], [1 - p, p])` est équivalent à `bernoulli(p)` (aux égalités $U = p$ près, de probabilité nulle).
- ★ La **justification** du fait que cette construction simule bien une variable de fonction de répartition F_X (méthode d'inversion) sera donnée en seconde année.



Les listes `valeurs` et `probas` doivent être de même longueur, et les p_i doivent être positifs de somme 1. Sinon, l'algorithme ne simule plus la loi voulue.

4) Estimation d'une probabilité, d'une espérance et d'une variance

Une fois que l'on sait **simuler** une expérience, on peut en **estimer** les caractéristiques en la répétant un grand nombre de fois. L'idée, issue de la **loi des grands nombres** (cours de seconde année), est que les moyennes observées se rapprochent des moyennes théoriques.

Estimations par simulation

On répète N fois, indépendamment, une même expérience.

- ★ **Probabilité.** Si N_A est le nombre de réalisations d'un événement A , alors $\frac{N_A}{N}$ estime $P(A)$.
- ★ **Espérance.** Si $x_1, \dots, x_N$ sont N simulations de X , alors la [moyenne empirique](#)

$$\bar{x}_N = \frac{1}{N} \sum_{i=1}^N x_i$$

estime $E(X)$.

- ★ **Variance.** D'après la formule de König-Huygens, $V(X) = E(X^2) - (E(X))^2$. On estime donc $V(X)$ par

$$\frac{1}{N} \sum_{i=1}^N x_i^2 - \left(\frac{1}{N} \sum_{i=1}^N x_i \right)^2$$

```
1 def estime_proba(evenement, n):
2     """
3     Entrée : une fonction evenement sans argument, qui renvoie True
4             si A est réalisé et False sinon ; un entier n
5     Sortie : une estimation de P(A) obtenue en répétant n fois l'expérience
6     """
7     s = 0
8     for _ in range(n):
9         if evenement():
10            s = s + 1
11     return s / n
12
13
14 def estime_esperance(tirage, n):
15     """
16     Entrée : une fonction tirage sans argument, qui simule X ; un entier n
17     Sortie : une estimation de E(X)
18     """
19     s = 0
20     for _ in range(n):
21         s = s + tirage()
22     return s / n
23
24
25 def estime_variance(tirage, n):
26     """
27     Entrée : une fonction tirage sans argument, qui simule X ; un entier n
28     Sortie : une estimation de V(X)
29     """
30     s = 0
31     s2 = 0
32     for _ in range(n):
33         x = tirage()
34         s = s + x
35         s2 = s2 + x ** 2
36     return s2 / n - (s / n) ** 2
```

Exemple : Estimons $P(X = 6)$ où X est le résultat d'un dé équilibré, dont la valeur théorique est $\frac{1}{6} \approx 0,1667$.

```
def six():
    return randint(1, 6) == 6

>>> estime_proba(six, 100)
0.19
>>> estime_proba(six, 10000)
0.1642
>>> estime_proba(six, 1000000)
0.16683
```

Plus N est grand, plus l'estimation se rapproche de la valeur théorique.

Exemple : Pour $X \hookrightarrow \mathcal{B}(0,3)$, on a $E(X) = 0,3$ et $V(X) = 0,3 \times 0,7 = 0,21$. Comme `bernoulli` attend un paramètre, on définit un tirage sans argument :

```
def bernoulli_03():
    return bernoulli(0.3)

>>> estime_esperance(bernoulli_03, 10000)
0.2964
>>> estime_variance(bernoulli_03, 10000)
0.2085
```

En répétant pour plusieurs valeurs de N :

N	Espérance estimée	Variance estimée
100	0,36	0,23
1 000	0,30	0,21
10 000	0,30	0,21
100 000	0,30	0,21

Les fluctuations sont visibles pour $N = 100$, et deviennent négligeables (au centième) dès que N atteint quelques milliers.

Remarque :

- ★ Les fonctions `evenement` et `tirage` sont passées en argument, comme `f` l'était pour les rectangles ou la dichotomie. Chaque appel `evenement()` ou `tirage()` relance une nouvelle expérience, indépendante des précédentes.
- ★ Une estimation n'est jamais exacte : c'est une **approximation aléatoire**. On l'améliore en augmentant N .



Pour estimer $E(X)$ ou $V(X)$, il faut bien simuler N **nouvelles** valeurs de X , et non réutiliser N fois le même tirage. Autrement dit, l'appel à `random` (ou à `bernoulli`, `binomiale`, `simule`, etc.) doit se trouver **à l'intérieur** de la boucle.