TP Informatique 7 : Introduction au hasard

1 Informatique et hasard

Dans la vie de tous les jours, si l'on souhaite obtenir un élément choisi aléatoirement parmi un ensemble fini E, la solution la plus simple consiste à lancer un dé équilibré ayant un nombre de faces égal au cardinal de l'ensemble en question, après avoir associé chacune des faces du dé à un des éléments de E. Par exemple, si $E = [\![1\,..\,6]\!]$, on peut utiliser un dé « courant » à six faces.

Cela fonctionne car le jet de dé est chaotique : une toute petite variation des conditions initiales (que l'on ne peut reproduire de façon exactement identique à chaque lancer) causera un changement dans le résultat obtenu, et l'on obtiendra les six résultats de façon à peu près équiprobables (des expériences récentes ont montré que le résultat n'était pas rigoureusement équiprobable, même pour un dé équilibré, si le dé est toujours lancé avec la même orientation initiale).

Pour un ordinateur, la tâche est plus ardue. En effet, un ordinateur est par construction un système déterministe, donc il est conçu pour donner toujours les mêmes résultats.

On pourrait créer un dispositif contrôlé par l'ordinateur lançant un dé physique et utilisant une caméra pour visualiser le résultat (ce qui, comme tout projet farfelu, a évidemment été fait : http://gamesbyemail.com/News/DiceOMatic) mais outre le fait que cela ne garantira pas une équiprobabilité rigoureuse des résultats, chose qui peut être importante pour des simulations numériques, un tel dispositif est très peu pratique.

Il a donc fallu trouver des sources d'entropie, l'équivalent microscopique d'un lancer de dé. Certains ordinateurs sont équippés d'un dispositif électronique créant un bruit d'origine quantique, qui peut être utilisé pour obtenir des bits aléatoires. Plus généralement, on se sert de l'environnement de l'ordinateur, imprévisible, pour obtenir de tels bits aléatoires. Par exemple, les mouvements de la souris (si vous déplacez votre souris deux fois, il est peu probable que vous soyez capable de faire le même mouvement au micromètre près et à la microseconde près).

Seulement, ces sources d'entropie ne permettent généralement pas de produire plus d'une poignée de valeurs aléatoires par seconde. Pour une application qui en a besoin d'une grande quantité, comme celles que nous allons étudier, on préfère en général utiliser des générateurs *pseudo-aléatoires*.

Ce sont des algorithmes qui fournissent des séquences de valeurs (entiers, flottants...) de façon déterministe, mais ayant l'apparence du hasard. Il s'agit toujours de la même suite de valeurs, mais elle est tellement longue ($2^{19937}-1$ valeurs pour l'un des générateurs les plus populaires, utilisé par le module `random` de Python, le *Mersenne Twister*) qu'on ne s'apercevra pas, dans la pratique, de sa périodicité. Il suffit alors de choisir un point de départ au hasard dans cette séquence (ce point de départ est appelé *graine*, ou *seed* en anglais).

Par exemple, on peut générer des jets d'un dé à six faces avec le générateur pseudo-aléatoire `randint`, par exemple 20 jets successifs :

```
In []: from random import randint

In []: [ randint(1, 6) for _ in range(20) ]
Out[]: [1, 2, 1, 3, 5, 6, 2, 3, 1, 6, 6, 2, 5, 6, 6, 3, 3, 6, 5, 1]
```

Si l'on en générait suffisamment de jets (10^{6002} environ), on finirait par obtenir à nouveau les mêmes résultats dans le même ordre, mais aucun ordinateur ne pourrait générer une séquence d'une telle longueur.

Dans la suite de ce TP, on cherchera à écrire quelques fonctions :
- générant des entiers aléatoires 8 bits (entre 0 et 255) ;
- générant des flottants aléatoires dans $[0,1[$.

Puis on se penchera sur quelques utilisations pratiques de tels générateurs aléatoires.

2 Générateurs pseudo-aléatoires d'entiers 8 bits

2.1 Générateurs congruentiels linéaires

Puisque les générateurs pseudo-aléatoires génèrent une suite de valeurs $(u_n) \in \mathbb{N}$, beaucoup utilisent une relation $u_{n+1} = f(u_n)$ pour calculer les différents u_n. La difficulté est de choisir correctement la fonction f.

Une fonction très utilisée est la fonction $f(u_{n+1}) = (a \times u_n + c) \mod m$ où a, b et m sont des entiers naturels, et l'opérateur $\mod$ calcule le reste de la division entière (ici par le diviseur m). De façon évidente, on obtient des valeurs dans $[\![0\,..\,m-1]\!]$. Pour obtenir des valeurs entre 0 et 255, il est donc naturel de choisir $m = 256$.

1. Proposer une fonction `Genere(N, u0, a, c, m)` qui retourne une liste contenant les termes u_1 à u_N de la suite définie par la valeur initiale `u_0` et la relation de récurrence $f(u_{n+1}) = (a \times u_n + b) \mod m$.

2.a Générer une liste de 30 valeurs pour $a = 25$, $c = 16$, $m = 256$ et $u_0 = 17$. Que pensez-vous du résultat ?

2.b Faire de même pour les valeurs initiales $u_0 = 54$, $u_0 = 34$ puis $u_0 = 42$, pour les mêmes paramètres a, c et m. Qu'en penser ?

2.2 Tester un générateur pseudo-aléatoire

Tous les générateurs pseudo-aléatoires ne sont donc pas de bonne qualité. Il nous faut des outils permettant de les comparer. L'ennui, c'est qu'il est impossible de déterminer si une séquence de valeurs a les caractéristiques d'une séquence « aléatoire » : pour un générateur parfaitement aléatoire, la probabilité pour qu'il retourne successivement mille ou un million de fois la valeur 1 est certes très faible, mais pas nulle.

Les premiers générateurs pseudo-aléatoires proposés en informatique n'étaient pas très bons. Cela a une conséquence importante : si un ordinateur a vérifié une propriété ou calculé quelque chose avec un corpus qui n'était pas aussi aléatoire qu'il l'aurait dû, le résultat obtenu peut être remis en cause.

Il a donc dû falloir développer des outils statistiques qui permettent de déterminer si un générateur pseudo-aléatoire est "bon" ou pas. La suite de tests la plus complète et la plus utilisée pour comparer différentes solutions est la suite de tests *die hard*. Nous allons étudier quelques critères simplifiés qu'un générateur pseudo-aléatoire devrait vérifier la plupart du temps pour que l'on puisse le considérer comme un générateur de qualité raisonnable.

Fréquences

3. Créer des listes `L1`, `L2` et `L3` contenant les 10000 entiers u_1 à u_{10000} pour $u_0 = 17$, $m = 256$ et les valeurs suivantes de a et c :

- $a = 25$ et $c = 16$ pour `L1` ;
- $a = 23$ et $c = 17$ pour `L2` ;
- $a = 25$ et $c = 17$ pour `L3`.

Nous allons à présent comparer les résultats et voir si un de ces générateurs congruentiels linéaires a donné des résultats plus intéressants qu'un autre.

Tout d'abord, dans une liste aléatoire, on s'attend, la plupart du temps, à avoir à peu près autant de nombre pairs que de nombre impairs.

4.a Écrire une fonction `ComptePair(L)` comptant le nombre d'entiers pairs dans une liste `L` fournie en paramètre.

4.b S'en servir pour compter le nombre d'entiers pairs et le nombre d'entiers impairs dans les liste `L1`, `L2` et `L3`. Peut-on éliminer un générateur sur ce critère ?

De la même façon, on s'attend à ce que tous les entiers apparaissent un nombre similaire de fois dans la liste ($10000/256 \simeq 39$ fois chacun en moyenne).

5.a Écrire une fonction `Compte(L)` qui retourne une liste `res` de longeur 256 telle que `res[i]` corresponde au nombre d'occurrence de l'entier `i` dans L.

5.b Se servir de la fonction `Compte` sur les listes `L1`, `L2` et `L3`. Qu'en conclure sur les générateurs proposés ?

Valeurs successives

Même si toutes les valeurs apparaissent de façon équiprobable, cela ne suffit pas pour qu'un générateur pseudo-périodique soit de qualité. Par exemple, la séquence d'entiers $0, 1, 2, \ldots, 254, 255, 0, 1, \ldots$ fournit les 256 valeurs de façon équiprobable, mais n'est pas très « aléatoire ».

6.a Ecrire une fonction `Suit(L, elem)` qui retourne la liste des `L[i+1]` pour tous les index i <`len(L)-1` tels que `L[i] == elem`.

6.b Déterminer les valeurs suivant un 17 dans `L3`. Les valeurs suivant un 42. Que constate-t-on ?

6.c Pourquoi était-ce parfaitement prévisible ?

6.d Vérifier également qu'un nombre pair est toujours suivi d'un nombre impair, et inversement. Justifier cette constatation à partir de la fonction f.

Ces trois premiers générateurs sont donc parfaitement épouvantables en terme de pseudo-aléatoire. Tant que l'on travaille avec $m = 256$, on ne pourra pas résoudre ce problème. On va donc choisir des m plus grands pour augmenter la période du générateur.

7.a Construire une liste `P4` contenant les 1000 entiers u_1 à u_{1000} pour $a = 16807$, $c = 0$, $m = 2 * *31 - 1$ (ce générateur aléatoire porte le nom de *standard minimal*) et $u_0 = 17$.

Le problème, c'est que les entiers de `P4` ne sont pas compris entre 0 et 255 ! Pour obtenir des nombres entre 0 et 255, on va prendre tous les éléments de `P4` et ne conserver que leur reste par une division entière par 256 (qui est un entier entre 0 et 255, comme souhaité).

7.b Construire une liste `L4` contenant les 10000 restes des divisions entières des u_i contenus dans `P4` par 256, et vérifier qu'elle contient bien des entiers entre 0 et 255.

7.c Vérifier que l'on n'a plus, cette fois, toujours le même entier suivant les 17 dans la liste `L4`.

D'un point de vue binaire, retenir le reste de la division par 256 correspond à ne conserver que les 8 derniers bits des entiers u_i. Ces entiers sont des entiers sur 31 bits (étant donné la valeur de m). Or ces derniers bits sont « moins aléatoires » que les autres. En particulier, il est aisé de prédire le comportement des bits de poids faibles, tel le bit le plus à droite qui détermine la parité du nombre.

Plutôt que de garder les 8 derniers bits, on préférera donc garder les 8 premiers bits (dans l'écriture sur 31 bits, en considérant des 0 à gauche si nécessaire). Obtenir ces 8 premiers bits se fait aisément, en divisant les u_i par 2^{23} !

7.d Construire une liste `L4h` contenant les 10000 quotients entiers des u_i contenus dans `P4` par $2^2 3$.

7.e Vérifier que, cette fois encore, on a des nombres différents suivant les 17.

7.f En utilisant la fonction `Compte`, vérifier que l'on a bien toutes les valeurs entre 0 et 255. Que penser de la répartition ? Est-elle plus ou moins satisfaisante que celle de `L3`, pour un générateur pseudo-aléatoire ?

Pour tracer la répartition, on peut utiliser les instructions suivantes si `lst` est une liste contenant les fréquences :

```
import matplotlib.pyplot as plt

plt.bar(range(len(lst)), lst, width=1.0)
```

Un autre générateur populaire il y a quelques dizaines d'années était le générateur RANDU, pour lequel $a = 65539$, $c = 0$ et $m = 2^{31} - 1$.

8.a Créer une liste `P5` contenant 10000 valeurs u_1 à u_{10000} pour le générateur RANDU, en prenant $u_0 = 17$.

8.b Créer une liste `L5h` contenant 10000 valeurs entre 0 et 255, quotient de la division entière des u_i de `P5` par 2^{23}.

9. Vérifier que `L5h` semble passer avec succès les tests précédents.

Représentation des séquences

Une autre manière d'identifier des faiblesses d'un générateur et d'éventuelle des règles qui régissent les successions de nombres est de représenter, dans un graphe, les points de coordonnées (L_i, L_{i+1}) où L est une liste contenant une séquence de valeurs fournies par le générateur aléatoire, et i un entier entre 0 et `len(L)-2`.

10.a Comment obtenir la liste des abscisses des points précédents ? La liste des ordonnées ?

10.b Tracer successivement, avec la commande `plot`, le nuage de points correspondant aux listes `L3`, `L4h` et `L5h`. On utilisera comme troisième argument à `plot` la chaîne `"b."` pour obtenir des points et non des segments les reliant.

Parfois, les défauts n'apparaissent pas en dimension 2, mais nécessitent de regarder en dimension $n > 2$.

Pour $n = 3$, on tracerait donc un nuage de points de coordonnées (L_i, L_{i+1}, L_{i+2}), où $0 \leqslant i \leqslant$ `len(L)-3`. Pour tracer un nuage de points en trois dimensions, on peut utiliser la séquence de commandes suivante :

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

plt.axes(projection="3d").scatter(X, Y, Z)
```

où `X`, `Y` et `Z` sont des listes de N éléments correspondant aux coordonnées selon les trois axes des points.

11.a Tracer successivement les nuages correspondant aux mille premiers points 3D associés à `L3`, `L4h` et `L5h`.

11.b Quel est le seul générateur qui se sort bien de ce test ?

Le générateur RANDU a été utilisé pendant des années, avant que l'on ne s'aperçoive que ses propriétés statistiques étaient épouvantables pour un générateur pseudo-aléatoire. Aujourd'hui, il est abandonné, mais tous les résultats obtenus en utilisant ce générateur ont été *de facto* remis en cause, ce qui n'est pas sans causer quelques difficultés.

Dernière remarque, et générateur d'entiers pseudo-aléatoires

L'ennui, si l'on étudie de plus près le générateur standard minimal, est qu'en l'état, les différentes valeurs n'ont pas exactement la même probabilité d'apparaître. En effet, dans la liste des u_n, 0 n'apparaît jamais (si on choisit $u_0 = 0$, on peut d'ailleurs vérifier que l'on a $u_n = 0 = \text{cste}$) et que $2^{31} - 1$ non plus.

Par conséquent, les valeurs 0 et 255 apparaîssent un tout petit peu moins souvent que les autres. C'est suffisamment rare pour que l'on ne s'en préoccuppe pas dans la plupart des situations, et il existe des moyens pour rétablir l'équiprobabilité.

On peut aussi utiliser un générateur différent. Par exemple, le générateur pour lequel $a = 1103515245$, $c = 12345$ et $m = 2^{31}$, utilisé par beaucoup de compilateurs C.

Parfois, on ne veut pas des entiers sur 8 bits, mais des entiers sur k bits. Plutôt que de prendre les 8 premiers bits des u_n, on prend donc les k premiers bits.

12. Écrire une fonction `GenereEntiers(N, k, u0)` qui :
- construit une liste de N valeurs u_1 à u_N, pour les paramètres a, c et m du dessus ;
- en déduit une liste de N valeurs entre 0 et $2^k - 1$ (inclus) en divisant chacun des termes de la liste précédente par 2^{31-k} ;
- retourne la seconde liste et u_N.

On a ainsi créé un générateur d'entiers aléatoires entre 0 et $2^k - 1$. Pour que les nombres soient statistiquement suffisamment aléatoires, il faut que k ne s'approche pas de 31, puisque les derniers bits sont de mauvaise qualité.

On retourne u_N car si l'on veut d'autres entiers, c'est ce u_N qui pourra servir de graîne pour le prochain appel, afin d'avoir la suite !

3 Générateurs pseudo-aléatoires de flottants dans $[0, 1[$

On s'intéresse à des nombres flottants IEEE sur 32 bits (même si Python stocke ses flottants sur 64 bits), qui comprennent 1 bit de signe, 8 bits pour l'exposant, et 23 bits pour la mantisse.

1.a Retrouver l'écart $\delta x = x' - x$ séparant deux flottants successifs x et x' tous deux dans l'intervale $[0.5, 1[$.

1.b Justifier que tous les réels de $[0, 1[$ de la forme $i \times \Delta x$ sont représentables exactement par un flottant. Combien y en a-t-il ?

1.c Vérifier que ces réels sont de la forme $i/2^k$ où i est un entier entre 0 et 2^k-1 (inclus), en précisant la valeur de k.

On peut donc créer un générateur pseudo-aléatoire de flottants dans $[0,1[$ à partir d'un générateur d'entiers sur k bits, entre 0 et $m = 2^k - 1$ (inclus).

1.d En déduire une fonction `GenereFlottants(N, u0)` qui, à partir de la fonction `GenereEntiers(N, k, u0)`, génère une liste de N flottants IEEE dans $[0,1[$ (`u0` étant toujours une graine entière), et retourne cette liste ainsi que le u_N retourné par `GenereEntiers` (de façon à pouvoir générer d'autres éléments ensuite).

4 Mouvements browniens

On appelle *mouvement brownien* (ou processus de Wiener) un processus mathématique visant à modéliser le mouvement d'une « particule » se déplaçant au sein d'un fluide, soumise à de fréquentes collisions changeant la direction de son déplacement. De très nombreux scientifiques ont contribué à l'étude de ces processus (R. Brown, L. Bachelier, A. Einstein, P. Langevin, N. Wiener...)

Nous allons étudier quelques variantes simples de ce problème, grâce à nos générateurs pseudo-aléatoires.

4.1 Cas unidimensionnel, déplacement entier

Dans un premier temps, on considère une particule initialement en $x = 0$, et se déplaçant uniquement sur l'axe des x. Elle peut se déplacer de deux façon : d'une distance de 1 vers la droite, avec une probabilité 50 %, et d'une distance de 1 vers la gauche, avec une probabilité 50 % également.

1. Proposer une fonction `Deplacement(N, u0)` qui construit les N positions de la particule aux N premiers instants, le sens de déplacement étant obtenu en demandant une liste d'entiers pseudo-aléatoires sur 1 seul bit (0 désignera un déplacement vers la gauche, 1 un déplacement vers la droite). La fonction retournera la liste de positions, et le u_N retourné par l'appel à `GenereEntiers`.

2. Tracer, sur un même graphe, le déplacement de dix particules pour N = 100 en utilisant la commande `plot` avec chacune des listes retournées par déplacement. On utilisera pour u_0 pour la première simulation la valeur 17, et pour les suivantes le u_N retourné.

3.a Déterminer la position finale pour N = 100 de 500 particules, et regrouper ces positions dans une liste `L`.

3.b Déterminer la valeur moyenne de ces positions. Quelle est l'espérance de la variable aléatoire associée à la position de la particule après 100 déplacements ?

4.a Dans une liste `P` de longueur $2N+1$, déterminer le nombre de particules `P[i]` qui a terminé son déplacement à la position $-N+i$, pour tout $0 \leqslant i \leqslant 2N$, et tracer (avec la commande `bar`) la distribution correspondante. Quel type de distribution obtient-on ?

4.b Estimer l'écart-type σ de cette distribution.

5.a Faire de même pour différentes valeurs de N (N = 100, N = 500, N = 2000, N = 10000). On pourra se limiter à 50 particules dans chaque cas, pour accélérer les calculs.

5.b Tracer sur un graphe $\log(\sigma(N))$ en fonction de $\log(N)$.

5.c Quelle relation peut-on conjecturer entre l'écart-type de la variable aléatoire associée à la position finale et N ?

4.2 Cas unidimensionnel, déplacement réel

On souhaite faire de même pour des déplacements latéraux qui peuvent avoir une valeur quelconque entre $-\sqrt{3}$ et $\sqrt{3}$. Précisons que l'on choisit $\sqrt{3}$ car une distribution uniforme sur $[-\sqrt{3},\sqrt{3}]$ a un écart-type de 1.

6.a Comment obtenir un ensemble de valeurs aléatoires régulièrement réparties dans $[-\sqrt{3},\sqrt{3}[$ à partir de `GenereFlottants` ? On ignorera le minuscule biais dû au caractère ouvert de l'intervalle à droite.

7. En utilisant la fonction `GenereFlottant`, effectuer plusieurs simulations pour plusieurs valeurs de N, avec des déplacements réels cette fois, et tracer à nouveau $\log(\sigma(N))$ en fonction de $\log(N)$. Que constate-t-on ?

4.3 Cas bidimensionnel, déplacement constant

Cette fois, on se place dans le plan, et on suppose qu'à chaque étape, la particule se déplace sur une distance constante $\delta = 1$, mais dans une direction choisie aléatoirement. Pour ce faire, on choisit aléatoirement des angles dans $[-\pi,\pi[$.

On s'intéresse à la position de la particule après N déplacements.

8.a Que peut-on dire de l'espérance associée à la variable aléatoire donnant l'abcisse de la particule après N déplacements ? Son ordonnée ?

8.b Qu'en est-il pour la variable aléatoire indiquant la distance de la particule par rapport au centre du repère ?

8.c Estimer la distance moyenne $d(N)$ au centre du repère pour 50 particules après respectivement N = 100, N = 500, N = 2000 et N = 10000 déplacements.

8.d Tracer $\log(d(N))$ en fonction de $\log(N)$. Que constate-t-on ?

Un tel modèle (ou du moins son équivalent en trois dimensions) décrit assez bien par exemple la diffusion d'un élément dans un gaz ou un liquide. Pour une substance chimique produite en un point donné, on peut donc estimer le temps nécessaire à ce que cette substance atteigne un point situé à une distance Δ de la source.

5 Méthode de Monte-carlo

La méthode de Monte-Carlo (qui tire son nom des casinos de la principauté) consiste, lorsqu'un calcul exact est difficile car nécessitant trop de temps, d'effectuer un certain nombre d'expériences particulières, et d'en tirer des conclusions.

Par exemple, si le calcul précis d'une espérance d'une variable aléatoire est difficile, on effectue un grand nombre de simulations, et on se sert de la moyenne des résultats pour obtenir une approximation de l'espérance en question. Plus le nombre d'expériences est grand, plus on a de chances d'avoir une moyenne proche de l'espérance.

Ce genre d'approche a d'innombrables applications, y compris de nombreuses applications qui ne sont pas directement liées aux probabilité. Par exemple, pour obtenir des images de synthèse de grande qualité, il faudrait envisager tous les trajets possibles de tous les photons dans la pièces, afin de tenir précisément compte des rebonds, réflexions, réfractions, etc. C'est évidemment impossible tant il y a de trajectoires possibles. Toutefois, en lançant un suffisamment grand nombre de photons au hasard (non seulement au départ, mais également à chaque étape d'un rebond sur une surface diffusant la lumière par exemple), on peut s'approcher du résultat souhaité. Les images ainsi obtenues (techniques de *radiosité*) sont alors de bien meilleure qualité que les images de synthèse habituelles.

De façon plus modeste, nous allons ici montrer comment il est possible de déterminer le volume d'une boule de rayon R = 1 dans R^3 à partir de la méthode de Monte-Carlo. Pour le calcul de ce volume, nous n'aurons pas besoin de π !

On s'intéresse au cube, dans $\mathbb{R}^3$ muni d'un repère $\left(\mathrm{O}, \vec{e}_x, \vec{e}_y, \vec{e}_z\right)$, défini par $-1 \leqslant x \leqslant 1$, $-1 \leqslant y \leqslant 1$ et $-1 \leqslant z \leqslant 1$.

1.a Quel est le volume V_c d'un tel cube ?

On note V le volume de la boule de rayon R = 1 centrée en O, origine du repère.

1.b Déterminer la probabilité $\mathscr{P}$ pour un point choisi aléatoirement dans le cube de se trouver dans la boule, en fonction de V et V_c. En déduire V_c en fonction de V et $\mathscr{P}$.

2.a Proposer une fonction `DansBoule(x, y, z)` retournant un booléen indiquant si le point (x, y, z) se trouve dans la sphère de rayon R = 1.

Pour estimer $\mathscr{P}$, nous allons utiliser la méthode de Monte-Carlo, en choisissant N points aléatoirement dans le cube, et en déterminant la proportion de ces points qui se situent dans la sphère.

2.b Écrire une fonction `Proba(N, u0)` qui retourne une estimation de $\mathscr{P}$ à partir de N points aléatoires, dont les coordonnées ont été obtenues en produisant 3N flottants à partir de la fonction `GenereFlottants`. La fonction retournera également le u_{N} retournée par `GenereFlottants`

2.c En déduire une fonction `Volume(N, u0)` retournant une estimation du volume d'une boule de rayon R = 1 par la méthode de Monte-Carlo, en utilisant N points aléatoires. Là encore, la fonction retournera également la valeur u_{N}.

2.d Tester la fonction pour N = 100, N = 1000, N = 10000 et N = 100000, et comparer le résultat à la valeur « théorique » $4\pi/3$.

Notons qu'il n'est pas intéressant d'effectuer k expériences pour un même N et de moyenner les résultats : de par le principe même de la méthode, cela revient très exactement à utiliser la méthode sur $k \times \mathrm{N}$ points choisis aléatoirement.

En revanche, on peut vouloir estimer l'écart-type des résultats de plusieurs appels à `Proba(N, u0)` (en utilisant pour u_0 à chaque appel le u_{N} retourné par l'appel précédent). Cela donne en effet une information quant à la précision du résultat.

3. En réalisant $k = 20$ expériences pour N = 100 et N = 10000, et en collectant les k résultats, calculer l'écart-type pour chaque N. Que pensez-vous de la précision de la méthode ?

La méthode de Monte-Carlo présente surtout un avantage quand le nombre de variables intervnant dans le calcul est très grand.

4.a Par une méthode similaire, estimer le volume d'une boule de rayon R = 1 dans $\mathbb{R}^{10}$.

Le volume d'une boule de rayon R = 1 en dimension n vaut $\pi^{n/2}/(n/2)!$ lorsque n est pair.

4.b Quel problème rencontre-t-on pour $\mathbb{R}^{20}$ ou $\mathbb{R}^{50}$?