Université de Poitiers
Département de Mathématiques

1M02 – Probabilités Générales
Master MFA–MMAS 1, année 2009–10

TRAVAUX PRATIQUES. — GÉNÉRATEURS ALÉATOIRES

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. — John von Neumann (1951)

Les travaux pratiques s'effectuerons en MAPLE ou en SCILAB selon les préférences de l'étudiant. Il est néanmoins recommandé à la spécialité MFA d'utiliser MAPLE et à la spécialité MMAS d'utiliser SCILAB.

Nous conseillons de se placer dans un répertoire de travail spécifique `1m02tp` (par exemple) pour y stocker scripts ou feuilles de travail (`*.mws` [*Maple Work Sheet*] ou `*.txt` [texte ASCII brut ou presque] pour MAPLE, `*.sce` [*SCilab Executable*] ou `*.sci` [*SCilab Input*] pour SCILAB), les graphiques PostScript ou PDF (`*.ps`, `*.eps`, `*.pdf`), et les ébauches de rapport tapées en TEX ou LATEX (`*.tex`). Crayons et papiers seront toujours nécessaires.

La saisie des scripts pourra se faire avec un éditeur de texte quelconque (`emacs` est évidemment recommandé, celui du logiciel devrait aussi convenir). L'avantage de l'écriture de scripts par rapport à l'utilisation directe de la ligne de commande est de pouvoir conserver sous une forme lisible le fruit de sa réflexion sans message d'erreurs ni sortie kilométrique plus ou moins utiles.

L'exécution d'un script ⟨*script*⟩`.sce` avec SCILAB se fait en tapant depuis la ligne de commande

`--> exec` ⟨*script*⟩`.sce` ou `--> exec("/home/`⟨*toto*⟩`/1m02tp/`⟨*script*⟩`.sce")`

Une possibilité comparable est offerte par MAPLE avec l'instruction `read("`⟨*script*⟩`.txt")`.

Les séances de travaux pratiques donneront lieu à un rapport unique et individuel.

Une introduction

Pour réaliser des simulations probabilistes, c'est-à-dire mettre un œuvre un modèle contenant une part d'aléatoire, on souhaite de pouvoir générer des suites $(u_n)_{n\geqslant 1}$ de nombres dans $[0,1]$ qui soient, ou représentent, une suite $(U_n(\omega))_{n\geqslant 1}$, où $U_n : (\Omega, \mathcal{A}, \mathbb{P}) \to [0,1]$, $n \geqslant 1$, est une suite de variables aléatoires, indépendantes uniformément distribuées sur $[0,1]$, et ω un élément de Ω. Mais, aussi bien conceptuellement que pratiquement, cela pose des problèmes.

- La suite $(u_n)_{n\geqslant 1}$ peut-elle prendre n'importe quelle valeur réelle dans $[0,1]$? Les représentations informatiques des nombres — c'est-à-dire des représentations finies — ne permettent pas d'accéder à tous les nombres réels, seulement à un nombre fini d'entre eux.
- Comment faire intervenir un hasard réel dans les choix successifs des $(u_n)_{n\geqslant 1}$? Des procédés physiques existent pour cela et se basent sur la nature quantique de la matière à petite échelle. Mais pour toute expérience scientifique — dont les simulations aléatoires — la reproductibilité est exigée. Si un tel hasard est mis en œuvre pour une simulation, la reproductibilité risque fort d'être impossible.
- En quoi une suite particulière $(u_n)_{n\geqslant 1}$ représente-t-elle plus ou mieux une réalisation de $(U_n)_{n\geqslant 1}$ qu'une autre ? Des suites numériques même assez régulières (et même constantes) seraient pour le mathématicien tout aussi recevables (mais peu amusantes).

Ces problèmes et les solutions qui ont été proposées pour y remédier sont abondamment discutées dans [1] et dépassent largement notre propos. Nous pouvons seulement, et très sommairement, indiquer quelle est la nature de ces solutions.

- Si on ne peut pas accéder à tous les nombres réels de l'intervalle $[0,1]$, au moins peut-on considérer un « grand nombre » d'entre eux régulièrement espacés afin d'approcher autant que possible une répartition uniforme dans cet intervalle. Ceci se fait classiquement en considérant l'ensemble des entiers $\{0, 1/m, \ldots, (m-1)/m\}$ où m est un entier « grand ». Ainsi, au lieu de regarder une suite à valeurs dans $[0,1]$, on s'intéresse aux suites à valeurs dans l'ensemble à m éléments $\{0, 1, \ldots, m-1\}$.
- L'exigence de reproductibilité impose que la suite $(x_n)_{n\geqslant 1}$ à valeurs dans $\{0,1,\ldots,m-1\}$, où $u_n = x_n/m$, soit donnée de manière algorithmique. En général, on se donne un certain entier $k \geqslant 1$, une certaine fonction $\phi : \{0,1,\ldots,m-1\}^k \to \{0,1,\ldots,m-1\}$, une donnée initiale $(x_1,\ldots,x_k)$, et on calcule de manière itérative $x_{n+1} = \phi(x_n,\ldots,x_{n-k+1})$. Il est à noter que la suite obtenue évolue dans un un ensemble fini (précisément $\{0,1,\ldots,m-1\}^k$) et qu'elle reviendra nécessairement à un k-uplet déjà atteint ; alors elle bouclera, c'est-à-dire aura une évolution périodique.
- Les propriétés mathématiques que le probabiliste serait tenté de demander ne porteraient pas sur une suite particulière $(x_n)_{n\geqslant 1}$ mais sur l'ensemble des suites qu'on pourrait obtenir de cette façon. Ceci n'ayant pas d'application concrète — c'est à partir d'une seule suite $(x_n)_{n\geqslant 1}$ que se fait couramment une simulation —, c'est sur le comportement en n que se portent les exigences.

  On exige ainsi dans un premier temps que chaque k-uplet soit atteint au cours d'une période, ce qui signifie que la suite $(x_n)_{n\geqslant 1}$ passe par chaque élément de $\{0,1,\ldots,m-1\}$ avec la même fréquence au cours de chacune de ses périodes. Ceci correspond à l'hypothèse selon laquelle chaque U_n est uniformément distribuée sur $[0,1]$.

  Pour rendre compte de l'indépendance des $(U_n)_{n\geqslant 1}$, certains critères ont été formulés sur ce que doit satisfaire de plus une telle suite de nombres $(x_n)_{n\geqslant 1}$, disons seulement qu'une certaine « imprédictibilité » est requise dans le passage d'un terme au suivant — ce qui bien sûr n'est pas, au sens strict, réalisable puisque la suite reste déterministe.

On appelle *générateur de nombres aléatoires* un algorithme, ou son implémentation, définissant une telle suite, c'est-à-dire la donnée de ϕ. La donnée initiale est appelée *graine*, ou *seed* en anglais. Sa valeur est souvent modifiable par l'utilisateur afin d'obtenir de nouvelles suites ou de reproduire une suite donnée.

1. Un exemple historique : la méthode du carré médian

En 1943, John von Neumann intégra le projet *Manhattan*, découvrit l'ENIAC à Philadélphie et se passionna pour ce nouvel outil — l'ordinateur — dont il envisageait l'importance dans la résolutions de problèmes ou de calculs mathématiques jusqu'àlors insolubles. Puisque certaines solutions numériques devaient être obtenues par des méthodes probabilistes, il s'intéressa à la génération de nombres aléatoires. En 1946, il proposa l'algorithme suivant : soit n entier strictement positif et supposons que x_k soit un nombre entier codé sur $2n$ décimales, (la base 10 était la base « physique » de calcul de l'ENIAC), on pose x_{k+1} l'entier défini par les $2n$ décimales centrales de x_k^2 (qui s'exprime quant à lui avec $4n$ décimales). Cette méthode est connue sous le nom de *middle-square method* (méthode du carré médian).

EXERCICE 1 (*théorique*). — Quel nombre suit 1010101010 dans la méthode du carré médian (essayer de le calculer de tête) ?

EXERCICE 2 (*théorique*). — Montrer que la méthode du carré médian utilisant des nombres à $2n$ chiffres en base b a le défaut suivant : si la suite inclut n'importe quel nombre dont les n

chiffres les plus significatifs sont nuls, les nombres suivants vont devenir de plus en plus plus petits jusqu'à ce que la suite stationne en zéro.

Exercice 3 (*théorique*). — Avec les hypothèses de l'exercice précédent, que peut-on dire de la suite des nombres succédant à x si les n chiffres les moins significatifs de x sont nuls ? Que dire si les $n+1$ chiffres les moins significatifs sont nuls ?

Exercice 4 (*pratique*). — (i) Définir une fonction `msrand()` (*Middle Square Random*) sans argument implémentant cette méthode et retournant la nouvelle valeur de la suite. Ses paramètres globaux pourront être `msseed`, etc., initialisés par une procédure `smsrand` (*Set Middle Square Random*) dont les arguments pourront être la graine, la base et la longueur de mot et être transformés en paramètres globaux. Une fonction `umsrand()` en sera déduite pour générer des nombres dans $[0,1]$ de manière uniforme (on l'espère).

(ii) Tester le générateur `umsrand()` obtenu en base 10 avec `msseed` $= 123456$, $2n = 6$: stocker dans un tableau la suite `u` obtenue en prenant par exemple $N = 100$ valeurs successives et tracer un histogramme de la suite obtenue.

Maple. Charger la bibliothèque `Statistics` avec `with(Statistics)`, puis, si $\langle u\rangle$ est la suite, exécuter `Histogram(`$\langle u\rangle$`)`. Des fonctions intéressantes sur les entiers sont ici $\langle x\rangle$ `mod` $\langle y\rangle$ ou `modp(`$\langle x\rangle$`,`$\langle y\rangle$`)`, `floor(`$\langle x\rangle$`)` ou encore `ceil(`$\langle x\rangle$`)`. Noter que x^y se code indifféremment $\langle x\rangle$`^`$\langle y\rangle$ et $\langle x\rangle$`**`$\langle y\rangle$.

Scilab. Exécuter la commande `histplot(`$\langle n\rangle$`,`$\langle u\rangle$`)` ($\langle n\rangle$ le nombre de classes ou une subdivision donnée). Les analogues Scilab des fonctions précédentes sont `modulo(`$\langle x\rangle$`,`$\langle y\rangle$`)` ou `pmodulo(`$\langle x\rangle$`,`$\langle y\rangle$`)`, `floor(`$\langle x\rangle$`)` ou encore `ceil(`$\langle x\rangle$`)`. Noter que x^y se code indifféremment $\langle x\rangle$`^`$\langle y\rangle$ et $\langle x\rangle$`**`$\langle y\rangle$.

Changer la graine (celle qui a été proposé n'étant peut-être pas si mauvaise), augmenter le nombre de valeurs de la suite, etc.

Exercice 5 (*pratique*). — On étudie complètement la méthode du carré médian pour des nombres à 2 décimales ($b = 10$, $2n = 2$).

(i) On commence le processus à partir de n'importe quelle valeur initiale 00, 01, ..., 98, 99. Combien de ces valeurs mènent au cycle 00, 00, ...

(ii) Combien de cycles finaux y a-t-il ? Quel est le plus long cycle ?

(iii) Quelle(s) valeur(s) initiale va produire le maximum de nombres distincts avant que la suite ne se répète ?

Le principal, voire le seul, intérêt de ce générateur est calculatoire : dans la base de l'ordinateur, les calculs sont rapides, et l'obtention du résultat par extraction invite à ne pas calculer tous les produits et ainsi à ne pas dépasser les limites de calculs éventuelles. Von Neumann n'avait aucune illusion sur cette méthode.

2. **Un aspect statistique**

Précédemment ainsi que dans ce qui suit pour cette séance de travaux dirigés, la suite `x` (ou plus exactement `u`) obtenue est censée représenter un *échantillon* (*sample* en anglais) de la loi uniforme sur $[0,1[$. Une procédure classique pour évaluer la qualité de cet échantillonnage consiste à comparer la fonction de répartition observée avec la fonction de répartition cible, ici celle de la loi uniforme sur $[0,1[$:

$$F : \mathbb{R} \longrightarrow [0,1]$$
$$x \longmapsto F(x) = \begin{cases} 0 & \text{si } x \leqslant 0 \\ x & \text{si } x \in [0,1] \\ 1 & \text{si } x \geqslant 1. \end{cases}$$

La fonction de répartition empirique F_n associée à l'échantillon $(x_1, \dots, x_n)$ est définie par

$$F_n(x) = \frac{1}{n} \operatorname{Card}\{1 \leqslant i \leqslant n : x_i \leqslant x\}.$$

pour tout $x \in \mathbb{R}$. Elle prend ses valeurs dans $\{0, 1/n, \dots, (n-1)/n, 1\}$. La *statistique* de Kolmogorov–Smirnov associée à ce problème est

$$k = \|F - F_n\|_\infty = \sup_{x \in \mathbb{R}} |F(x) - F_n(x)|,$$

et on montre que si F est continue on a

$$k = \max_{i=1}^n \big(F(x_{(i)}) - (i-1)/n\big) \vee \big(i/n - F(x_{(i)})\big),$$

où $x_{(1)} \leqslant \cdots \leqslant x_{(n)}$ est l'échantillon ordonné dans le sens croissant et $a \vee b = \max(a, b)$, et qui est une expression facilement calculable sur ordinateur (le tri s'obtient avec des commandes `sort(`$\langle x \rangle$`)` ou `gsort(`$\langle x \rangle$`)` [attention à l'ordre de tri], le maximum avec `max(`$\langle x \rangle$`)`).

Pour une méthode donnée, cette statistique k va varier en fonction de l'échantillon obtenu. Lorsque celle-ci est généralement petite — ce qui est quantifié par la loi de Kolmogorov de paramètre n —, l'hypothèse d'uniformité est acceptable, sinon il faut remettre en cause la méthode.

Si le logiciel dispose de fonctions de calcul sur les distributions de Kolmogorov, on peut calculer la p-valeur du test d'adéquation de Kolmogorov–Smirnov à la loi cible : celle-ci est simplement $1 - F_{K_n}(k)$ où F_{K_n} est la fonction de répartition de la loi de Kolmogorov de paramètre n la taille de l'échantillon et k la statistique du test. Typiquement, si la p-valeur est inférieure à 5 %, on rejette l'hypothèse d'adéquation. Évidemment, un seul test sur le générateur ne suffit pas pour décider s'il convient ou non.

Avec Maple et Scilab, les fonctions de répartitions de lois de probabilités sont des commandes comportant `cdf` (*Cumulative Distribution Function*) dans leurs noms. Les commandes disponibles pour ces logiciels se limitent à des lois de probabilité relativement communes.

Exercice 6 (*pratique*). — (i) Implémenter via une fonction `KSustat(`$\langle u \rangle$`)` le calcul de la statistique de Kolmogorov–Smirnov pour une suite $\langle u \rangle$ censée se répartir uniformément sur $[0, 1]$. Le mettre en œuvre avec la méthode du carré médian sur de petits échantillons. Comparer les valeurs obtenues avec le ou les quantiles d'ordre $1 - \alpha = 0{,}95$ correspondants dans la table donnée en fin de document.

(ii) (*facultatif*) Dudley (1964) a montré que pour tout $u > 0$,

$$\lim_{n \to \infty} \mathbb{P}\big\{K_n \leqslant u/\sqrt{n}\big\} = 1 + 2 \sum_{k=1}^{\infty} (-1)^k \exp\big(-2k^2u^2\big).$$

ou encore pour $x > 0$, lorsque n tend vers l'infini,

$$1 - \mathbb{P}\big\{K_n \leqslant x\big\} \sim 2 \sum_{k=1}^{\infty} (-1)^{k+1} \exp\big(-2k^2nx^2\big) = 2\exp\big(-2nx^2\big) - 2\exp\big(-8nx^2\big) + \cdots$$

En utilisant cette approximation grossière, définir une fonction `KSupvalue(`$\langle u \rangle$`)` calculant la p-valeur approchée du test de Kolmogorov–Smirnov d'adéquation avec la loi uniforme sur $[0, 1]$.

Exercice 7 (*pratique*). — Des implémentations des fonctions de répartitions des Lois de Kolmogorov se trouvent dans le répertoire

`http://www-math.univ-poitiers.fr/~phan/masterMMAS/documents/1m02/`

sous la forme de fichiers de macros `kolmogorov.sci` pour Scilab et `kolmogorov.txt` pour Maple. Récupérer l'un de ces fichiers, examiner rapidement son contenu et l'utiliser pour réaliser à nouveau le test précédent.

3. **Un exemple fondamental : les suites congruentes linéaires**

La classe de générateurs la plus simple est donnée, d'une part, pour $k = 1$, c'est-à-dire avec $x_{n+1} = \phi(x_n)$, et avec ϕ la fonction la plus simple qu'on puisse imaginer dans ce cadre, à savoir une fonction affine modulo l'entier maximal m :

$$x_{n+1} = (a \times x_n + c) \bmod m,$$

ainsi x_{n+1} est encore un entier compris entre 0 et $m - 1$. Ce type de relation est appelée *congruence linéaire* et les propriétés d'une suite ainsi définie sont bien connues des spécialistes. Le choix des entiers a (le multiplicateur [*multiplier*]), c (l'accroissement [*increment*]), et m (le module [*modulus*]) n'est pas quelconque. Pour ne discuter que de m, ce doit être un entier grand dans le domaine de calcul de la machine ou du logiciel utilisé ; il est de plus essentiel que m soit un nombre premier de la forme $2^p - 1$, c'est-à-dire un nombre premier de Mersenne[1] (on démontre qu'alors p est nécessairement premier). Cette méthode est souvent appelée « méthode des congruences linéaires » et est due à D. H. Lehmer (1948).

Implémentations. — Des logiciels de calcul scientifique ou statistique utilisent ce type de générateurs avec des choix de (a, c, m) éventuellement différents. Par exemple,

$$m = 2^{31} - 1 = 2\,143\,483\,647, \qquad a = 7^5 = 16\,807, \qquad c = 0$$

est un choix qui satisfait — ou presque car, comme $c = 0$, la valeur $x = 0$ poserait clairement problème pour une telle suite — les propriétés qu'on peut attendre d'un tel générateur. Remarquons que $m = 2^{31} - 1$ est le plus grand nombre premier de Mersenne qui reste dans le domaine de calcul en entiers d'un processeur 32 bits. Ce choix est celui de [2].

Logiciel	`commande`	`graine`	m	a	c
TEX/random.tex	`\nextrandom`	`\randomi`	$2^{31} - 1$	16 807	0
SAS	`(2^31-1)*ranuni()`	`seed`	$2^{31} - 1$	397 204 094	0
SCILAB	`(2^31)*rand()`	`rand("seed")`	2^{31}	843 314 861	453 816 693
MAPLE $v \leqslant 9.5$	`rand()`	`_seed`	$10^{12} - 11$	427 419 669 081	0
...	...	...	...	...	...

Citons, pour MAPLE $v > 9.5$, le sous-paquet `RandomTools[LinearCongruence]` où on trouve l'ancien générateur invoqué par les commandes `GenerateInteger`, `GetState`, `NewGenerator`, `SetState`.

Pour SCILAB, `rand()` ne retourne que les nombres dans $[0, 1]$ correspondant à la suite d'entiers sous-jacente et il semble devoir multiplier par 2^{31} pour récupérer cette suite ; l'initialisation de la graine se fait par `rand("seed",`$\langle n \rangle$`)`.

Le cas de SAS est presque identique.

Quel que soit le logiciel utilisé, il est fortement recommandé de se reporter à sa documentation courante : les noms des commandes, leurs paramètres internes et les algorithmes utilisés peuvent changer sans véritable préavis (ou sinon assez discret).

EXERCICE 8 (*pratique*). — (i) Définir une fonction MAPLE, ou SCILAB, `myrand()`, ainsi que la fonction `myurand()` correspondante, basée sur les explications précédentes en utilisant le triplet (a, c, m) de l'ancienne implémentation de MAPLE ou celui de SCILAB et avec pour graine `myseed`.

(ii) Comparer le générateur obtenu avec l'homologue du logiciel utilisé pour vérifier que les informations glanées dans différentes documentations ont un fond de vrai. Dans le cas de

1. Marin MERSENNE (1588–1648), moine français.

SCILAB, pour une graine identique, on doit avoir une coïndence approximative de `myrand()` et de `2^31*rand()` au moins pour les premiers termes ; cependant une divergence apparaît assez rapidement. Bien que ceci ne soit documenté nulle part, avancer une explication de ce phénomène.

(iii) Mettre en concurrence le « nouveau » générateur avec celui qui est actuellement recommandé par le logiciel sur la production de suites uniformément réparties sur $[0, 1]$: comparer les histogrammes et les statistiques de Kolmogorov–Smirnov pour des suites de longueurs 10, 100, 1000, etc.

(iv) (MAPLE) Vérifier que $m = 10^{12} - 11$ est premier (`isprime(m);`). Est-ce un nombre premier de Mersenne ? (Utiliser la fonction `log[2](.)` pour voir si $m + 1$ est de la forme 2^n.) Sinon, quelle justification simple pourrait-on trouver pour un tel choix ?

Remarques. — *a*) Dans la définition de la procédure `myrand()` les quantités `a`, `c`, `m` et `myseed` ont une valeur globale : ce sont les valeurs préalablement connues de `a`, `c`, `m` et `myseed` qui sont utilisées et la nouvelle valeur de `myseed` est prise en compte pour la suite.

b) Lorsqu'on sait qu'un générateur aléatoire est basé sur la méthode des congruences linéaires, il n'est pas difficile de trouver la valeur de c : il suffit d'initialiser la graine à 0 et de faire un appel au générateur ($(a \times 0 + c) \bmod m = c \bmod m = c$). Puis on peut retrouver la valeur de a en initialisant la graine à 1 (en supposant $a + c \leqslant m$). Pour m, on peut toujours faire une longue liste de nombres premiers de Mersenne, puis faire tourner le générateur un certain nombre de fois afin d'identifier la ou les bornes supérieures plausibles, et enfin faire quelques essais pour s'assurer d'avoir trouvé le bon module m.

4. **Autres générateurs de nombres aléatoires**

La commande `rand()` de MAPLE est désormais basée sur le *Mersenne Twister* qui est un algorithme récent (1997, les initiales des noms des auteurs sont M. et T.) qui commence à être suffisamment sérieusement éprouvé pour parvenir à s'imposer à la place des congruences linéaires. On constate que SCILAB 5.1 utilise aussi le Mersenne Twister via `grand()`, et que c'est aussi le cas de SAS ($v \geqslant 9$) via `rand()`. On commence à trouver de la documentation grand public sur ce générateur (Wikipedia anglophone).

Par ailleurs, on ne peut résister à évoquer le générateur aléatoire implémenté par Donald Knuth dans MetaFont, et que l'on retrouve dans MetaPost, accessible via les instruction `randomseed :=` $\langle x \rangle$, `uniformdeviate` $\langle x \rangle$ ou encore `normaldeviate` ; celui-ci est décrit dans [1], p. 27–28, et est du type *lagged Fibonacci sequences* (suites de Fibonacci décalées).

En C, c'est encore plus compliqué. La librairie `stdlib.h` définit la fonction `rand()` dont l'initialisation est faite par `srand()` mais aussi un couple `random()` et `srandom()` et puis les `<d,e,l,n,m,j>rand48()` et `srand48()`, etc. Les mécanismes invoqués par ces fonctions peuvent changer d'une version à l'autre de la librairie, changements qui ne sont pas nécessairement signalés et pas toujours bien documentés. Si bien que certains physiciens théoriciens (grands amateurs de simulations aléatoires) préfèrent parfois écrire leurs propres librairies standard...

EXERCICE 9 ([1], exercice 6, p. 193). — Look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find.

RÉFÉRENCES

[1] KNUTH (Donald E.), *The Art of Computer Programming, volume 2, Seminumerical Algorithms*, third edition, Addison–Wesley (1998).

[2] Press (W.H.), Teukolsky (S.A.), Vetterling (W.T.), Flannery (B.P.), *Numerical Recipes in C*, second edition, Cambridge University Press (2002).

[3] Marsaglia (G.) et Wan Tsang (W.), « Evaluating Kolmogorov's Distribution », préprint.

[4] Perrin–Riou (Bernadette), *algèbre, arithmétique et maple*, Cassini (2000).

[5] Chancelier (J.-P.), Delebecque (F.), Gomez (C.), Goursat (M.), Nikoukhah (R.), Steer (S.), *Introduction à Scilab, deuxième édition*, Springer-Verlag France (2007).

Table de quantiles de la statistique de Kolmogorov–Smirnov

Si K_n est la statistique de Kolmogorov–Smirnov correspondant à une taille d'échantillon égale à n, la table donne $k_{n,1-\alpha} \in [0,1]$ tel que $\mathbb{P}\{K_n \leqslant k_{n,1-\alpha}\} = 1-\alpha$.

α \ n	0	1	2	3	4	5	6	7	8	9
0.01	1.0000	0.9950	0.9293	0.8290	0.7342	0.6685	0.6166	0.5758	0.5418	0.5133
0.05	1.0000	0.9750	0.8419	0.7076	0.6239	0.5633	0.5193	0.4834	0.4543	0.4300
0.10	1.0000	0.9500	0.7764	0.6360	0.5652	0.5094	0.4680	0.4361	0.4096	0.3875
0.15	1.0000	0.9250	0.7261	0.5958	0.5248	0.4744	0.4353	0.4050	0.3806	0.3601
0.20	1.0000	0.9000	0.6838	0.5648	0.4927	0.4470	0.4104	0.3815	0.3583	0.3391

α \ n	10	11	12	13	14	15	16	17	18	19
0.01	0.4889	0.4677	0.4490	0.4325	0.4176	0.4042	0.3920	0.3809	0.3706	0.3612
0.05	0.4092	0.3912	0.3754	0.3614	0.3489	0.3376	0.3273	0.3180	0.3094	0.3014
0.10	0.3687	0.3524	0.3381	0.3255	0.3142	0.3040	0.2947	0.2863	0.2785	0.2714
0.15	0.3425	0.3273	0.3141	0.3023	0.2918	0.2823	0.2737	0.2659	0.2587	0.2520
0.20	0.3226	0.3083	0.2957	0.2847	0.2748	0.2658	0.2577	0.2503	0.2436	0.2373

α \ n	20	21	22	23	24	25	26	27	28	29
0.01	0.3524	0.3443	0.3367	0.3295	0.3229	0.3166	0.3106	0.3050	0.2997	0.2947
0.05	0.2941	0.2872	0.2809	0.2749	0.2693	0.2640	0.2591	0.2544	0.2499	0.2457
0.10	0.2647	0.2586	0.2528	0.2475	0.2424	0.2377	0.2332	0.2290	0.2250	0.2212
0.15	0.2459	0.2402	0.2348	0.2298	0.2251	0.2207	0.2166	0.2127	0.2089	0.2054
0.20	0.2315	0.2261	0.2211	0.2164	0.2120	0.2079	0.2040	0.2003	0.1968	0.1934

α \ n	30	31	32	33	34	35	36	37	38	39
0.01	0.2899	0.2853	0.2809	0.2768	0.2728	0.2690	0.2653	0.2618	0.2584	0.2552
0.05	0.2417	0.2379	0.2342	0.2308	0.2274	0.2242	0.2212	0.2183	0.2154	0.2127
0.10	0.2176	0.2141	0.2108	0.2077	0.2047	0.2018	0.1991	0.1965	0.1939	0.1915
0.15	0.2021	0.1989	0.1958	0.1929	0.1901	0.1875	0.1849	0.1825	0.1801	0.1779
0.20	0.1903	0.1873	0.1844	0.1817	0.1791	0.1766	0.1742	0.1718	0.1696	0.1675

α \ n	40	42	44	46	48	50	52	54	56	58
0.01	0.2521	0.2461	0.2406	0.2354	0.2306	0.2260	0.2217	0.2177	0.2138	0.2102
0.05	0.2101	0.2052	0.2006	0.1963	0.1922	0.1884	0.1848	0.1814	0.1782	0.1752
0.10	0.1891	0.1847	0.1805	0.1766	0.1730	0.1696	0.1664	0.1633	0.1604	0.1577
0.15	0.1757	0.1715	0.1677	0.1641	0.1607	0.1575	0.1545	0.1517	0.1490	0.1465
0.20	0.1654	0.1616	0.1579	0.1545	0.1514	0.1484	0.1456	0.1429	0.1404	0.1380

α \ n	60	65	70	75	80	85	90	95	100	105
0.01	0.2067	0.1988	0.1917	0.1853	0.1795	0.1742	0.1694	0.1649	0.1608	0.1570
0.05	0.1723	0.1657	0.1597	0.1544	0.1496	0.1452	0.1412	0.1375	0.1340	0.1308
0.10	0.1551	0.1491	0.1438	0.1390	0.1347	0.1307	0.1271	0.1238	0.1207	0.1178
0.15	0.1441	0.1385	0.1336	0.1291	0.1251	0.1214	0.1181	0.1150	0.1121	0.1094
0.20	0.1357	0.1305	0.1258	0.1216	0.1178	0.1144	0.1112	0.1083	0.1056	0.1031

α \ n	110	120	130	140	150	160	170	180	190	200
0.01	0.1534	0.1470	0.1413	0.1362	0.1316	0.1275	0.1237	0.1203	0.1171	0.1142
0.05	0.1279	0.1225	0.1178	0.1135	0.1097	0.1063	0.1031	0.1003	0.0976	0.0952
0.10	0.1151	0.1103	0.1060	0.1022	0.0988	0.0957	0.0929	0.0903	0.0879	0.0857
0.15	0.1070	0.1025	0.0985	0.0950	0.0918	0.0889	0.0863	0.0839	0.0817	0.0796
0.20	0.1008	0.0965	0.0928	0.0895	0.0865	0.0838	0.0813	0.0790	0.0769	0.0750

TRAVAUX PRATIQUES. — GÉNÉRATEURS ALÉATOIRES, COMPLÉMENTS

Nous donnons un « corrigé pratique » pour l'exercice 5. L'aspect algorithmique n'a rien d'intéressant ou presque. Nous avons simplement traduit un petit programme MetaPost fait à la va-vite en langage MAPLE et SCILAB, ce qui, *a posteriori*, était assez facile. On notera les éléments de programmation : boucles `for` et `while`, tests, fonction d'affichage élémentaire `printf` ou `disp`, les manipulations de chaînes de caractères... et surtout la proximité de syntaxe entre MAPLE et SCILAB ainsi que leurs différences.

```
# code maple
# Exercice 5
# M\'ethode du carr\'e m\'edian, \'etude des orbites, $2n=2$

# On notera que la variable xcurrent varie de 0 \`a n-1
# alors que les indices du tableau x[] vont de 1 \`a n

Orbits := proc(F, n)
local x, xcurrent, noz, i, j, k, s;

x := array(1..n); s := ""; noz := 0;# number of zeros

for i from 0 to n-1 do
    xcurrent := i; x[i+1] = -1;# initialisation
    for j from 1 to n do
        xcurrent := F(xcurrent);
        if xcurrent = 0 then noz := noz+1; break; end if;
    end do;
end do;
printf("Nombre de suites terminant en 0 : %d\n", noz);# syntaxe C

printf("Cycles terminaux :\n");
for i from 0 to n-1 do
    xcurrent := i; x[i+1] := i;
    for j from 0 to n-1 do
        xcurrent := F(xcurrent);
        if x[xcurrent+1] = i then break; end if;
        x[xcurrent+1] := i;
    od:
    s := sprintf("cycle terminal de %d : %d", i, xcurrent);# syntaxe C
    x[xcurrent+1] := infinity;# constante ``infinie''
    xcurrent := F(xcurrent);
    while x[xcurrent+1] <> infinity do# bien noter ``diff\'erent de''
        s := cat(s, sprintf(", %d", xcurrent));
        x[xcurrent+1] := infinity;
        xcurrent := F(xcurrent);
    end do;
    printf(cat(s, "\n"));
end do;
```

```
printf("Plus longues orbites :\n");
for i from 0 to n-1 do
    xcurrent := i; x[i+1] := i; s := sprintf("%d", xcurrent); k := 1;
    for j from 0 to n-1 do
        xcurrent := F(xcurrent);
        if x[xcurrent+1] = i then break; end if;
        x[xcurrent+1] := i; s := cat(s, sprintf(", %d", xcurrent)); k := k+1;
    end do;
    printf(cat(sprintf("orbite %d, longueur %d : ", i, k), s, "\n"));
end do;
end proc:

F := x -> floor((x*x mod 1000)/10); Orbits(F, 100);
```

On notera dans le code SCILAB que l'auteur a choisi de préciser en commentaires qu'elles étaient les variables locales du programme. Bien que la commande `local` n'existe peut-être pas en SCILAB contrairement à la commande `global`, cette précision rend le code plus clair et, de plus, plus aisément transposable dans un langage où cette mention serait nécessaire.

```
// code scilab
// Exercice 5
// M\'ethode du carr\'e m\'edian, \'etude des orbites, $2n=2$

// On notera que la variable xcurrent varie de 0 \`a n-1
// alors que les indices du tableau x() vont de 1 \`a n

function Orbits(F, n)
//local x xcurrent noz i j k s;

x = zeros(n); s = ""; noz = 0;// number of zeros

for i = 0:n-1
    xcurrent = i; x(i+1) = -1;// initialisation
    for j = 1:n
        xcurrent = F(xcurrent);
        if xcurrent == 0 then noz = noz+1; break; end
    end
end
disp("Nombre de suites terminant en 0 : "+string(noz));
// noter l'affichage d'une cha\^\i ne de caract\`eres
// et la concat\'enation de deux cha\^\i nes en Scilab

disp("Cycles terminaux :");
for i = 0:n-1
    xcurrent = i; x(i+1) = i;
    for j = 0:n-1
        xcurrent = F(xcurrent);
        if x(xcurrent+1) == i then break; end
        x(xcurrent+1) = i;
    end
    s = "cycle terminal de "+string(i)+" : "+string(xcurrent);
    x(xcurrent+1) = %inf;// constante ``infinie''
    xcurrent = F(xcurrent);
    while x(xcurrent+1) ~= %inf// bien noter ``diff\'erent de''
        s = s+", "+string(xcurrent);
```

```
        x(xcurrent+1) = %inf;
        xcurrent = F(xcurrent);
    end
    disp(s);
end

disp("Plus longues orbites :");
for i = 0:n-1
    xcurrent = i; x(i+1) = i; s = string(xcurrent); k = 1;
    for j = 0:n-1
        xcurrent = F(xcurrent);
        if x(xcurrent+1) == i then break; end
        x(xcurrent+1) = i; s = s+", "+string(xcurrent); k = k+1;
    end
    disp("orbite "+string(i)+", longueur "+string(k)+" : "+s);
end
endfunction;

deff("y = F(x)", "y = floor(modulo(x*x,1000)/10)");
// Orbits(F, 100);
```

Le code MetaFont/Post est très semblable. L'avantage est qu'on peut indexer à partir de 0, l'inconvénient est qu'on ne peut pas pousser trop loin la méthode à cause des capacités de calcul limitées de ce langage.

```
% code metapost
% Exercice 5
% M\'ethode du carr\'e m\'edian, \'etude des orbites

% On notera que la variable xcurrent varie de 0 \`a n-1
% ainsi que les indices du tableau x[]

def Orbits(suffix F)(expr n) =
begingroup;
    save x, xcurrent, noz, k, s;
    string s; s = ""; noz := 0;% number of zeros

    for i = 0 upto n-1:
        xcurrent := i; x[i] = -1; % initialisation
        for j = 1 upto n:
            xcurrent := F(xcurrent);
            if xcurrent = 0: noz := noz+1; fi
            exitif xcurrent = 0;
        endfor
    endfor
    message "Nombre de suites terminant en 0 : "& decimal noz;

    message "Cycles terminaux :";
    for i = 0 upto n-1:
        xcurrent := i; x[i] := i;
        for j = 0 upto n-1:
            xcurrent := F(xcurrent); exitif x[xcurrent] = i;
            x[xcurrent] := i;
        endfor
        x[xcurrent] := infinity;
```

```
        s := "cycle terminal de "&decimal i&" : "&decimal xcurrent;
        forever:
            xcurrent := F(xcurrent); exitif x[xcurrent] = infinity;
            s := s&", "&decimal xcurrent;
        endfor
        message s;
    endfor

    message "Plus longues orbites :";
    for i = 0 upto n-1:
        xcurrent := i; x[i] := i; s:= decimal xcurrent; k:=1;
        for j = 0 upto n-1:
            xcurrent := F(xcurrent); exitif x[xcurrent] = i;
            x[xcurrent] := i;
            s := s&", "&decimal xcurrent; k := k+1;
        endfor
        message "orbite "&decimal i&", longueur "&decimal k&" : "&s;
    endfor
endgroup;
enddef;

vardef F(expr x) = floor((x*x mod 1000)/10) enddef;
Orbits(F, 100);
```

TRAVAUX PRATIQUES. — SIMULATIONS DE VARIABLES ALÉATOIRES

1. **Méthodes directes**

Ce premier exercice ne sert qu'à cadrer le thème de ces travaux dirigés : nous savons, ou presque, générer des « variables aléatoires indépendantes de loi uniforme sur $[0,1]$ », passons aux autres lois. Celles-ci seront, ou bien discrètes (diagrammes en bâtons et test du χ^2 qui sera vu plus loin), ou bien absolument continues par rapport à la mesure de Lebesgue, notamment, de fonctions de répartition continues (histogrammes, test de Kolmogorov–Smirnov).

Exercice 1 (*pratique*). — Soit $(U_i)_{i\in\mathbb{N}}$ une suite de variables aléatoires indépendantes, uniformément distribuées sur $[0,1]$. D'un point de vue pratique, cette suite correspond aux appels successifs à un générateur de nombres aléatoires convenablement normalisé.

Pour chacune des lois proposées, décrire à l'aide d'une ou plusieurs variables aléatoires de la suite $(U_i)_{i\in\mathbb{N}}$ une ou plusieurs méthodes pour réaliser une variable aléatoire de loi la loi considérée, la traduire dans le langage utilisé.

(i) Loi de Bernoulli $\mathcal{B}(1,p)$, $p \in [0,1]$, qui est discrète et se représente à l'aide d'un diagramme en bâtons.

```
% pseudo-code \‘a traduire et tester
definition PouF(real p) =
    if myurand() < 1-p then return 0; else return 1; fi
enddefinition;
```

(ii) Loi uniforme sur $\{1,2,3,4,5,6\}$, qui est discrète et se représente à l'aide d'un diagramme en bâtons. On pourra procéder de deux manières différentes : la première en utilisant les retours de `myrand()` modulo 6 ; la seconde en utilisant la partie entière de `myurand()` multipliée par 6.

```
% pseudo-code \‘a traduire et tester
definition dice(void) = % utilise les d\’ecimales de poids faible
    return (myrand() mod 6)+1;
enddefinition;
definition Dice(void) = % utilise les d\’ecimales de poids fort
    return floor(6*myurand())+1; % 0 <= myurand() < 1
enddefinition;
```

(iii) Loi discrète à n valeurs possibles $\{x_1,\dots,x_n\}$ de probabilités respectives $p_1,\dots,p_n$. Ici, le vecteur $(p_1,\dots,p_n)$ est un argument de la procédure de génération. Celle-ci peut retourner simplement l'indice i correspondant. (On n'utilisera pas de procédure toute faite…)

```
% pseudo-code \‘a traduire et tester
definition empirical(list t) = % t = p_1,...,p_n
    local u, i, p;
    u = myurand(); i = 0;
    for p = t;
        i = i+1; u = u-p;
```

```
        if u <= 0 then break; fi
    endfor
    return i;
enddefinition;
```

(iv) Loi uniforme $\mathcal{U}([a,b])$, $a<b$, qui n'est bien entendu pas discrète et ne nécessite plus de tests graphiques ou numériques (Kolmogorov–Smirnov) puisqu'ils ont déjà été abordés pour $a=0$ et $b=1$.

2. **Un aspect statistique**

Pour la majorité des quelques cas qui précèdent, la loi considérée est discrète. Si c'est une mesure de probabilité sur $\mathbb{R}$, alors sa fonction de répartition F est une fonction en escalier. La comparaison de cette fonction de répartition avec celle qui a été observée ne peut se faire par l'approche de Kolmogorov–Smirnov pour laquelle F est supposée continue. Dans le cas discret, nous nous retrouvons donc à devoir trouver une approche complémentaire de celle de de Kolmogorov–Smirnov. Celle-ci existe, c'est le *test du χ^2 d'adéquation à une loi discrète*.

Considérons une loi discrète π portée par un ensemble fini $E=\{e_1,\dots,e_k\}$ et notons $\pi_i=\pi\{e_i\}>0$. Si $x=(x_j)_{j=1}^n$ est une suite de points de E, un échantillon, on note $p_i=n_i/n$ la proportion de termes égaux à e_i. La distribution observée est proche de π si tous les p_i sont proches des π_i correspondants. La statistique associée à ce problème est la *statistique du test du χ^2 d'adéquation* (χ^2 se dit « khi-deux » ou *"chi-square"*)

$$cs=\sum_{i=1}^{k}\frac{(n_i-n\times\pi_i)^2}{n\times\pi_i}=n\times\sum_{i=1}^{k}\frac{(p_i-\pi_i)^2}{\pi_i},$$

et devrait prendre généralement de « faibles » valeurs quand la distribution observée est proche de la distribution cible. Ceci est quantifié de manière asymptotique en n par la loi du χ^2, ou de Pearson, à $\nu=k-1$ degrés de liberté. Si F_{k-1} désigne la fonction de répartition de la loi du χ^2 à $k-1$ degrés de liberté, la p-valeur asymptotique du test d'adéquation est $1-F_{k-1}(cs)$. Lorsque des conditions d'application du test asymptotique sont satisfaites ($n\geqslant 30$, et $n_i\geqslant 5$, pour tout i, par exemple), une p-valeur très petite conduit à rejeter l'hypothèse selon laquelle l'échantillon aurait pu être tiré suivant la loi π. Notons que si l'échantillon comporte des valeurs x_j pour lesquelles $\pi\{x_j\}=0$, on rejette l'hypothèse sans même se poser de question : l'échantillon ne peut en aucun cas avoir été tiré suivant la loi π.

3. **Cadre discret, mise en pratique**

Nous allons créer une procédure `dgoftest` pour *discrete goodness of fits* dont le rôle sera de comparer les données d'un échantillon avec une loi discrète donnée en paramètre.

3.1. Programmes Maple

Dans un premier temps, nous introduisons une nouvelle fonction `size` qui devrait résoudre certains problèmes d'évaluation de tailles de tableaux.

```
# code maple
# Travaux pratiques. --- Simulation de variables al\'eatoires

size := proc(u)
    if type(u,array) then return op(2,op(2,eval(u)));
    else return nops(u);
    end if;
end proc:
```

Elle a été testée avec la version 6 de Maple, car alors `nops` ne donne pas de bons résultats avec les tableaux (`array`) alors qu'il convient pour les listes (`list`). On aurait pu aussi se servir par endroits de `convert(`⟨*variable*⟩`,`⟨*type*⟩`)` pour s'assurer d'avoir à faire avec des listes ou des tableaux suivant les cas.

Puis, si et seulement si on ne veut pas se servir des outils actuels, on réintroduit le générateur aléatoire de Maple d'origine.

```
m := 10^12-11: a := 427419669081: c := 0: myseed := 1:

myrand := proc()
    global a, c, m, myseed;
    myseed := modp(a*myseed+c, m);
    return myseed;
end proc:

myurand := proc()
    global m;
    return evalf(myrand()/m);
end proc:
```

Nous aurons besoin de bibliothèques particulières

```
with(plots):# penser aux deux points
#with(Statistics):# penser aux deux points
#with(stats):
```

pour les graphiques, mais aussi pour les calculs liés aux lois de Pearson $\chi^2(\nu)$. Avec la bibliothèque `stats` de Maple, la fonction de répartition de la loi $\chi^2(k-1)$ s'obtient par

`stats[statevalf, cdf, chisquare[`⟨$k-1$⟩`]](`⟨cs⟩`)`,

tandis qu'avec la bibliothèque plus récente `Statistics`, la syntaxe

`evalf(CDF(ChiSquare(`⟨$k-1$⟩`),` ⟨cs⟩`))`

semble conseillée (le respect de la casse est lui-même recommandé).

Et voici enfin la procédure principale :

```
dgoftest := proc(x, xtheo, ptheo)
    local n, k, pobs, cs, pvaleur, flag, i, j, total, dx;
    printf("dgoftest :\n");
    #
    # Comptabilit\'e
    #
    n := size(x); k := size(xtheo); pobs := array(1..k);
    flag := false; total := 0;
    for i from 1 to k do
        pobs[i] := 0;
        for j from 1 to n do
            if x[j] = xtheo[i] then pobs[i] := pobs[i]+1; end if;
        end do;
        if pobs[i] < 5 then flag := true; end if;
        total := total+pobs[i];
        pobs[i] := pobs[i]/n;
    end do;
    #
```

```
    # Test du khi-deux
    #
    printf("Test du khi-deux d'adequation\n");
    printf("Taille de l'echantillon = %d\n", n);
    printf("Nombre theorique de valeurs k = %d\n", k);
    if total < n then
        printf("Des valeurs observees sont hors des valeurs theoriques.\n");
        printf("L'hypothese d'adequation est trivialement rejetee.\n");
    else
        cs := n*add((pobs[i]-ptheo[i])^2/ptheo[i], i = 1..k);
        printf("Statistique de test cs = %f\n", cs);
        if n < 30 or flag then
        printf("Les conditions d'application du test asymptotique\n");
        printf("ne sont pas satisfaites. Cependant,\n");
        end if;
        pvaleur := 1-stats[statevalf, cdf, chisquare[k-1]](cs);
        # pvaleur := 1-evalf(CDF(ChiSquare(k-1), cs));
        printf("p-valeur asymtotique = %f\n", pvaleur);
    end if;
    #
    # Graphiques
    #
    dx := abs(xtheo[k]-xtheo[1]);
    for i from 2 to k do
        dx := min(dx, abs(xtheo[i]-xtheo[i-1]));
    end do;
    dx := dx/10;
    printf("dgoftest, graphique et fin.\n");
    # voir "plot[structure]"
    PLOT(
        POLYGONS(seq([
            [xtheo[i]-dx,0],
            [xtheo[i]-dx,ptheo[i]],
            [xtheo[i],ptheo[i]],
            [xtheo[i],0]
            ], i = 1..k), COLOR(RGB,.7,.7,1.0)),
        POLYGONS(seq([
            [xtheo[i],0],
            [xtheo[i],pobs[i]],
            [xtheo[i]+dx,pobs[i]],
            [xtheo[i]+dx,0]
            ], i = 1..k), COLOR(RGB,1.0,.7,.7))
        );
end proc:
```

Nous avons choisi de comptabiliser les données « à la main ». Cela permet d'avoir un contrôle sur les structures et de pouvoir les réutiliser ensuite plutôt que de demander à différentes procédures de faire plusieurs fois le même travail.

Le calcul de la statistique du test se fait simultanément avec la vérification de son applicabilité. Si la variable `total` est strictement inférieure à n, des données se trouvent en dehors des valeurs théoriques prévues. Le test d'adéquation doit donc conduire au rejet le plus total.

Pour les graphiques, nous n'avons pas utilisé la commande `Histogram` de MAPLE car elle peut être inadaptée et il est plus rapide et plus fiable de faire des diagrammes en bâtons ou barres (pour lesquels c'est la hauteur des barres qui indique la probabilité) à la main.

Il ne reste plus qu'à tester l'ensemble :

```
Bernoulli := proc(p, N)
    local x, i;
    x := array(1..N);
    for i from 1 to N do
        if myurand() > 1-p then x[i] := 1; else x[i] := 0; end if;
    end do;
    dgoftest(x, [0, 1], [1-p, p]);
end proc:

Bernoulli(0.25, 100);
```

Ce qui aura pu donner pour résultat :

```
dgoftest :
Test du khi-deux d'adequation
Taille de l'echantillon = 100
Nombre theorique de valeurs k = 2
Statistique de test cs = 1.920000
p-valeur asymtotique = .165857
dgoftest, graphique et fin.
```

avec un graphique en plus.

3.2. PROGRAMMES SCILAB

Dans un premier temps, si et seulement si on ne veut pas se servir de `rand()` ni de `grand()`, on réintroduit le générateur aléatoire de SCILAB d'origine avec l'ajout de « my » et la distinction entre suite de nombres dans $[0, 1]$ et suite d'entiers avec la présence ou non de « u ».

```
// code scilab
// Travaux pratiques. --- Simulation de variables al\'eatoires

m = 2^31; a = 843314861; c = 453816693;
myseed = 1;

function x = myrand()
    global a c m myseed;
    myseed = modulo(a*myseed+c, m);
    x = myseed;
endfunction;

function x = myurand()
    global m;
    x = myrand()/m;
endfunction;
```

Ici, il n'est pas besoin de charger des bibliothèques spéciales. Les fonctions de répartition des lois de Pearson $\chi^2(\nu)$ sont directement accessibles avec SCILAB. La syntaxe d'appel est, pour obtenir des probabilités

$$\texttt{[}\langle p\rangle\texttt{, }\langle q\rangle\texttt{] = cdfchi("PQ", }\langle x\rangle\texttt{, }\langle df\rangle\texttt{)},$$

des quantiles

$$[\langle x\rangle] = \texttt{cdfchi("X", } \langle df\rangle\texttt{, } \langle p\rangle\texttt{, } \langle q\rangle\texttt{)},$$

et même le degré de liberté

$$[\langle df\rangle] = \texttt{cdfchi("Df", } \langle p\rangle\texttt{, } \langle q\rangle\texttt{, } \langle x\rangle\texttt{)},$$

`"Df"` signifiant *degree of freedom*.

Et voici enfin la procédure principale :

```
function dgoftest(x, xtheo, ptheo)
    // local i j k n pobs cs pvaleur total;
    mprintf("dgoftest :\n");
    //
    // Comptabilit\'e
    //
    n = size(x,1); k = size(ptheo,1); pobs = zeros(k, 1);
    for i = 1:k
        for j = 1:n
            if x(j) == xtheo(i) then pobs(i) = pobs(i)+1; end
        end
    end
    total = sum(pobs); pobs = pobs/n;
    //
    // Test du khi-deux
    //
    // noter le redoublement des apostrophes dans les cha\^\i nes suivantes
    mprintf("Test du khi-deux d''adequation\n");
    mprintf("Taille de l''echantillon = %d\n", n);
    mprintf("Nombre theorique de valeurs k = %d\n", k);
    if total < n then
        mprintf("Des valeurs observees sont hors des valeurs theoriques.\n");
        mprintf("L''hypothese d''adequation est trivialement rejetee.\n");
    else
        cs = 0; flag = %f;// bool\'een, false
        for i = 1:k
            cs = cs+(pobs(i)-ptheo(i))^2/ptheo(i);
            if pobs(i)*n < 5 then flag = %t;// bool\'een, true
            end
        end
        cs = n*cs;
        mprintf("Statistique du khi-deux d''adequation cs = %f\n", cs);
        if n < 30 | flag then// le ``ou'' logique est la barre verticale
        mprintf("Les conditions d''application du test asymptotique\n");
        mprintf("ne sont pas satisfaites. Cependant,\n");
        end
        pvaleur = 1-cdfchi("PQ", cs, k-1);
        mprintf("p-valeur asymtotique = %f\n", pvaleur);
    end
    //
    // Graphiques
    //
    printf("dgoftest, graphique et fin.\n");
```

```
    clf();
    bar(xtheo, [ptheo, pobs]);
    xtitle("Simulations et loi theorique");
    legend(["distribution theorique"; "distribution observee"]);
    a = gca(); a.grid = [-1,3];
    // halt("taper return pour continuer");
endfunction;
```

Les commentaires faits sur la version MAPLE de ce programme sont aussi valables ici. Notons que SCILAB ne fait pas les diagrammes en bâtons tout seul, il se contente de dresser les barres.

Il ne reste plus qu'à tester l'ensemble :

```
function Bernoulli(p, N)
    // local x i;
    for i = 1:N
        if myurand() > 1-p then x(i) = 1; else x(i) = 0; end
    end
    dgoftest(x, [0; 1], [1-p; p]);
    // noter l'\'ecriture explicite des vecteurs colonnes
endfunction;

Bernoulli(0.25, 100);
```

Ce qui aura pu donner pour résultat :

```
dgoftest :
Test du khi-deux d'adequation
Taille de l'echantillon = 100
Nombre theorique de valeurs k = 2
Statistique du khi-deux d'adequation cs = 0.000000
p-valeur asymtotique = 1.000000
dgoftest, graphique et fin.
```

avec un graphique en plus.

EXERCICE 2 (*pratique*). — (i) Tester la procédure `Bernoulli` en faisant varier les différents paramètres (graine, taille de l'échantillon, générateur).

(ii) Définir des procédures semblables pour la loi uniforme sur $\{1, \ldots, 6\}$ ainsi que pour une loi discrète à support fini quelconque (*voir* exercice 1).

4. **Cadre continu, mise en pratique**

Nous allons créer une procédure `cgoftest` pour *continuous goodness of fits* dont le rôle sera de comparer les données d'un échantillon avec une loi de fonction de répartition continue qui sera donnée en paramètre. Nous revenons donc au test de Kolmogorov–Smirnov.

4.1. PROGRAMMES MAPLE

Nous avons besoin des distributions de Kolmogorov pour faire les calculs.

```
read "kolmogorov.txt":
```

```
KSstat := proc(u, F)
    local ks, v, n, i;
    if type(u,array) then
        n := op(2, op(2,eval(u)));
        v := sort([seq(u[i], i = 1..n)]);
    else # elif type(u,list) then
        n := nops(u);
        v := sort(u);
    end if;
    ks := 0;
    for i from 1 to n do
        ks := evalf(max(ks, F(v[i])-(i-1)/n, i/n-F(v[i])));
    end do;
    return ks;
end proc:
```

Le premier argument de `KSstat` est l'ensemble des données sous la forme d'un tableau ou d'une liste, le second est le nom de la fonction de répartition de la loi « cible ». Pour la procédure en elle-même, il n'y a rien de compliqué :

```
cgoftest := proc(x, F)
    local xs, n, ks, pvaleur;
    printf("cgoftest :\n");
    n := size(x);
    #
    # Test de Kolmogorov--Smirnov
    #
    ks := KSstat(x, F): pvaleur := 1-kolmogorovcdf(ks, n):
    printf("Test de Kolmogorov-Smirnov\n");
    printf("Taille de l'echantillon n = %d\n", n);
    printf("Statistique de test ks = %f\n", ks);
    printf("p-valeur exacte = %f\n", pvaleur);
    #
    # Graphiques
    #
    printf("cgoftest, graphiques et fin.");
    xs := sort([seq(x[i], i = 1..n)]);
    PLOT(
        CURVES([seq([xs[i], F(xs[i])], i = 1..n)], COLOR(RGB,.0,.0,1.0)),
        CURVES([seq([xs[i], i/n], i = 1..n)], COLOR(RGB,1.0,.0,.0))
    );
end proc:
```

Ce qu'on peut tester :

```
N := 100: u := array(1..N):
for i from 1 to N do u[i] := myurand(): od:
F := x -> min(max(x, 0), 1):
cgoftest(u, F);
```

4.2. Programmes Scilab

Nous avons besoin des distributions de Kolmogorov pour faire les calculs.

```
getf "kolmogorov.sci";
```

```
function ks = KSstat(u, F)
    // local v n i;
    n = size(u,1);
    v = -gsort(-u);// sort() est d\'epr\'eci\'e (attention \`a l'ordre)
    ks = 0;
    for i = 1:n; ks = max(ks, F(v(i))-(i-1)/n, i/n-F(v(i))); end
endfunction
```

Le premier argument de `KSstat` est l'ensemble des données sous la forme d'un vecteur colonne, le second est le nom de la fonction de répartition de la loi « cible ». Pour la procédure en elle-même, il n'y a rien de compliqué :

```
function cgoftest(x, F)
    // local xs n ks pvaleur Fxs;
    mprintf("cgoftest :\n");
    n = size(x,1);
    //
    // Test de Kolmogorov--Smirnov
    //
    ks = KSstat(x, F); pvaleur = 1-kolmogorovcdf(ks, n);
    mprintf("Test de Kolmogorov-Smirnov\n");
    mprintf("Taille de l''echantillon n = %d\n", n);
    mprintf("Statistique de test ks = %f\n", ks);
    mprintf("p-valeur exacte = %f\n", pvaleur);
    //
    // Graphiques
    //
    mprintf("cgoftest, graphiques et fin.");
    clf();
    xs = -sort(-x); rang = [1:n]'/n;
    for i = 1:n; Fxs(i) = F(xs(i)); end
    plot(xs, [Fxs, rang]);
    xtitle("Simulations et loi theorique");
    legend(["distribution theorique"; "distribution observee"]);
    // halt("taper return pour continuer");
endfunction;
```

Ce qu'on peut tester :

```
N = 100; u = zeros(N, 1);
for i = 1:N; u(i) = myurand(); end
deff("y = F(x)", "y = min(max(x, 0), 1)");
cgoftest(u, F);
```

Exercice 3 (*pratique*). — Tester la génération de nombre selon une loi uniforme sur un intervalle $[a, b]$ de bornes autres que 0 et 1 (*voir* exercice 1).

5. Utilisation de la fonction de répartition

Exercice 4 (*théorique*). — Soient π une mesure de probabilité sur $\mathbb{R}$ et F sa fonction de répartition, c'est-à-dire $F(x) = \pi(]-\infty, x])$ pour tout $x \in \mathbb{R}$.

(i) Supposons que F est (strictement) croissante et continue sur un intervalle $]a, b[$ et vérifie $\lim_{x\downarrow a} F(x) = 0$ et $\lim_{x\uparrow b} F(x) = 1$. Montrer que si X est une variable aléatoire de loi π, alors la variable aléatoire $U = F \circ X$ est de loi uniforme sur $[0, 1]$.

(ii) Plus généralement, définissons l'inverse généralisé F^{-1} de F par

$$F^{-1}(u) = \inf\{y \in \mathbb{R} : u \leqslant F(y)\} \qquad \text{pour tout } u \in [0, 1],$$

(qui est une fonction croissante, continue à gauche et limitée à droite). Soient U une variable aléatoire de loi uniforme sur $[0, 1]$ et $X = F^{-1} \circ U$. Montrer que les événements $\{X \leqslant x\}$ et $\{U \leqslant F(x)\}$ sont égaux pour tout $x \in \mathbb{R}$; en déduire que la variable aléatoire X a pour loi π.

Exercice 5 (*pratique*). — On poursuit l'exercice 1 en utilisant la méthode d'inversion de la fonction de répartition.

(i) Loi géométrique $\mathcal{G}(p)$, $p \in]0, 1]$, qui est discrète et portée par $\mathbb{N}^*$. Pour le test du χ^2, on bornera les échantillons par une valeur maximale M en remplaçant les valeurs observées supérieures à M par M. La comparaison se fera alors avec la loi de $T \wedge M$ où T est de loi $\mathcal{G}(p)$.

(ii) Loi exponentielle $\mathcal{E}(\lambda)$, $\lambda \in \mathbb{R}_+^*$, qui est absolument continue et portée par $\mathbb{R}_+$. Le test de Kolmogorov–Smirnov est alors adapté pour vérifier l'adéquation de la loi observée avec la loi cible dont la fonction de répartition, ainsi que son inverse, se calcule facilement à la main.

(iii) Loi de Cauchy $\mathcal{C}(a)$, $a > 0$, qui est absolument continue, portée par $\mathbb{R}$, et admet pour densité

$$x \in \mathbb{R} \longmapsto \frac{a}{\pi(a^2 + x^2)}\,.$$

Comme pour les lois exponentielles, le test d'adéquation est celui de Kolmogorov–Smirnov et les différentes fonctions se déterminent aisément.

(iv) Loi Normale (gaussienne centrée réduite) $\mathcal{N}(0, 1)$, qui est absolument continue et portée par $\mathbb{R}$. Sa fonction de répartition Φ et son inverse s'obtiennent numériquement en considérant

`statevalf[cdf, normald[`$\langle m\rangle$`, `$\langle\sigma\rangle$`]](`$\langle x\rangle$`)`

avec la bibliothèque `stats`, ou

`evalf(CDF(Normal(`$\langle m\rangle$`, `$\langle\sigma\rangle$`), `$\langle x\rangle$`))`

avec `Statistics` pour Maple, et

`cdfnor("PQ", `$\langle x\rangle$`, `$\langle m\rangle$`, `$\langle\sigma\rangle$`)`

pour Scilab (ici $m = 0$ et $\sigma = 1$).

(v) Loi normale (gaussienne) $\mathcal{N}(\mu, \sigma^2)$, $\mu \in \mathbb{R}$, $\sigma \in \mathbb{R}_+^*$.

6. Utilisation de propriétés indirectes

Exercice 6 (*pratique*). — On continue les procédures de simulation et les tests d'adéquation, graphiques, et numériques.

(i) Loi binomiale $\mathcal{B}(n, p)$, $p \in [0, 1]$, $n \in \mathbb{N}$. Penser au schéma de Bernoulli fini.

(ii) Loi géométrique $\mathcal{G}(p)$, $p \in [0, 1]$. Penser au Schéma de Bernoulli infini.

(iii) Loi de Poisson $\mathcal{P}(\lambda)$, $\lambda \geqslant 0$. Penser aux files d'attente.

Exercice 7 (*théorique*). — Soient X et Y deux variables aléatoires indépendantes de lois $\mathcal{N}(0,1)$. Le couple (X,Y) étant considéré comme les coordonnées d'un point aléatoire du plan, on passe en coordonnées polaires : $R = \sqrt{X^2+Y^2}$, $\Theta = \arctan(Y/X)$.

(i) Montrer que R et Θ sont indépendantes et déterminer leurs lois respectives.

(ii) En déduire une méthode pratique pour simuler une variable aléatoire de loi $\mathcal{N}(0,1)$, puis, plus généralement, de loi $\mathcal{N}(\mu,\sigma^2)$.

Remarque. — La méthode présentée dans l'exercice précédent est couramment appelée « méthode polaire ». Elle a été introduite dans l'article de G. E. P. Box et Mervin E. Muller [6].

Exercice 8 (*pratique*). — (i) Implémenter un générateur pour la loi $\mathcal{N}(0,1)$ en utilisant la méthode suggérée par l'exercice précédent. Deux nombres étant générés, on conservera celui qui reste pour l'appel suivant plutôt qu'en générer alors deux nouveaux.

(ii) Implémenter un générateur pour les lois de Pearson $\chi^2(n)$, qui sont les lois de sommes de n variables aléatoires réelles identiquement distribuées de loi $\mathcal{N}(0,1)$. La méthode retenue est-elle la meilleure qu'on puisse envisager lorsque $n = 2$?

(iii) Implémenter un générateur pour les lois de Student $\mathcal{T}(n)$, qui sont les lois de quotients $Z/\sqrt{S^2/n}$ avec Z et S^2 deux variables aléatoires réelles indépendantes, la première de loi $\mathcal{N}(0,1)$, la seconde de loi de Pearson $\chi^2(n)$.

(iv) etc. On pourra se demander s'il n'est pas plus judicieux pour les derniers cas de se servir de l'inverse de la fonction de répartition.

7. Les méthodes de rejet

Il est très facile de générer une loi uniforme sur le carré $[0,1]^2$ ou l'hypercube $[0,1]^n$ à l'aide de 2 ou n variables aléatoires $U_1,\dots,U_n$ indépendantes de loi uniforme sur $[0,1]$, puisque c'est précisément la loi du vecteur aléatoire $(U_1,\dots,U_n)$. La généralisation à la loi uniforme sur un pavé $[a_1,b_1]\times\cdots\times[a_n,b_n]$ est immédiate : c'est la loi du vecteur aléatoire $(a_1+(b_1-a_1)\times U_1,\dots,a_n+(b_n-a_n)\times U_n)$.

Il est assez naturel de vouloir générer des lois uniformes sur des structures géométriques simples.

Exercice 9. — (i) Soit $U_1,\dots,U_n,\dots$ une suite de variables aléatoires indépendantes de loi uniforme sur $[0,1]$. À l'aide de cette suite, proposer des définitions de variables aléatoires devant admettre pour lois :

a) la loi uniforme sur le cercle $\mathbb{S}^1 = \{(x,y)\in\mathbb{R}^2 : x^2+y^2=1\}$;

b) la loi uniforme sur le disque $\mathbb{D}^2 = \{(x,y)\in\mathbb{R}^2 : x^2+y^2<1\}$;

c) la loi uniforme sur la boule $\mathbb{D}^3 = \{(x,y,z)\in\mathbb{R}^2 : x^2+y^2+z^3<1\}$;

d) la loi uniforme sur la sphère $\mathbb{S}^2 = \{(x,y)\in\mathbb{R}^2 : x^2+y^2+z^2=1\}$.

(ii) Pour la loi uniforme sur le disque, mettre en œuvre la proposition retenue à l'aide de Maple ou Scilab. On représentera dans le plan le nuage de points obtenu (1 000 points, par exemple). Obtenez-vous le résultat escompté ? (si oui, tant mieux.) Qu'en est-il en dimension 3 ?

(iii) Soit $M : (\Omega,\mathcal{A},\mathbb{P}) \to \mathbb{D}^2$ une variable aléatoire de loi uniforme sur le disque. En coordonnées polaires, ce point est représenté par deux variables aléatoires R et Θ. Indiquer quelle est la loi de Θ, constater que Θ et R sont indépendantes, et déterminer la fonction de répartition de R. Est-ce cohérent avec la proposition faite pour (i-*b*) ? (si oui, tant mieux.)

(iv) Se servir, si ce n'est déjà fait, de la question précédente pour définir une variable aléatoire de loi uniforme sur le disque. Déduire de cette dernière une variable de loi uniforme sur le cercle.

(v) Semble-t-il facile, voire possible, d'adopter une démarche semblable pour $\mathbb{S}^2$ et $\mathbb{D}^3$? S'inspirer du titre de la section ou du théorème suivant pour trouver une méthode qui fonctionne.

THÉORÈME (MÉTHODE DE REJET). — *Soient E et F deux ensembles tels que $F \subset E$ vérifiant l'une des propriétés suivantes :*

(i) *E et F sont deux ensembles finis non vides ;*

(ii) *E et F sont deux parties mesurables de $\mathbb{R}^n$ de mesures de Lebesgue finies strictement positives.*

Soient $(U_n)_{n\geqslant 1}$ une suite de variables aléatoires indépendantes uniformément réparties sur E et $T = \inf\{n \geqslant 1 : U_n \in F\}$. Alors la variable aléatoire T est presque sûrement finie et

$$\begin{aligned} Z = U_T : \Omega &\longrightarrow F \\ \omega &\longmapsto Z(\omega) = U_{T(\omega)}(\omega) \end{aligned}$$

est une variable aléatoire uniformément répartie sur F.

De plus, si on définit par récurrence

$$T_1 = T, \qquad \text{et} \qquad T_{n+1} = \inf\{n > T_n : U_n \in F\}, \quad n \geqslant 1,$$

les variables aléatoires $Z_n = U_{T_n}$, $n \geqslant 1$, sont indépendantes et uniformément réparties sur F.

Démonstration. — Soit λ la mesure uniforme sur E, et notons $p = \lambda(F) \in]0, 1]$. La variable aléatoire T est le premier rang de succès dans un schéma de Bernoulli infini de paramètre $p > 0$. Nous savons donc que T est presque sûrement finie et que sa loi est la loi géométrique de paramètre p. Quitte à poser $Z(\omega) = \delta$, où δ est un point arbitraire, lorsque $T(\omega) = \infty$, l'application $Z : \Omega \to F$ (ou $F \cup \{\delta\}$) est bien définie et est mesurable (cela se lit *a posteriori* dans le calcul qui suit). Pour $B \subset F$ mesurable, on a par la formule des probabilités totales,

$$\begin{aligned} \mathbb{P}\{Z \in B\} &= \sum_{n=1}^{\infty} \mathbb{P}\{T = n, U_n \in B\} = \sum_{n=1}^{\infty} \mathbb{P}\{U_1 \notin F, \dots, U_{n-1} \notin F, U_n \in B\} \\ &= \sum_{n=1}^{\infty} \mathbb{P}\{U_1 \notin F\} \times \cdots \times \mathbb{P}\{U_{n-1} \notin F\} \times \mathbb{P}\{U_n \in B\} \\ &= \sum_{n=1}^{\infty} \lambda(E\backslash F) \times \cdots \times \lambda(E\backslash F) \times \lambda(B) \\ &= \sum_{n=1}^{\infty} \lambda(E\backslash F) \times \cdots \times \lambda(E\backslash F) \times \lambda(F) \times \lambda(B)/\lambda(F) \\ &= \sum_{n=1}^{\infty} (1-p) \times \cdots \times (1-p) \times p \times \lambda(B)/\lambda(F) \\ &= \left(\sum_{n=1}^{\infty} (1-p)^{n-1} p \right) \times \lambda(B)/\lambda(F) \\ &= \lambda(B)/\lambda(F) \end{aligned}$$

où on note que $\mathbb{P}\{T = \infty\} = 0$ et que les $(U_n)_{n\geqslant 1}$ sont indépendantes de loi λ. Ainsi, Z est bien de loi uniforme sur F. Pour montrer l'indépendance de $(Z_n)_{n\geqslant 1}$, il suffit de calculer

$\mathbb{P}\{Z_1 \in B_1, \dots, Z_n \in B_n\}$ d'une manière semblable à celle qui précède et constater après des écritures de sommations et d'indices multiples que cette probabilité est égale à $\lambda(B_1)/\lambda(F) \times \cdots \times \lambda(B_n)/\lambda(F)$, ce qui permet d'identifier la loi de chaque Z_i, $1 \leqslant i \leqslant n$, et établit leur indépendance, pour tout $n \geqslant 1$, et donc pour la suite toute entière. □

Remarque. — La démonstration nous donne un résultat plus général : si λ est une mesure de probabilité quelconque sur E telle que $\lambda(F) > 0$, la loi de X est $\lambda(\,.\,\cap F)/\lambda(F)$, donc la loi λ conditionnellement à F.

Théorème. — *Soit $p : \mathbb{R}^d \to \mathbb{R}_+$ une fonction de densité d'une mesure de probabilité μ sur $\mathbb{R}^d$. Posons $F = \{(x, y) \in \mathbb{R}^d \times \mathbb{R} : 0 \leqslant y \leqslant p(x)\}$ qui est de mesure de Lebesgue 1 dans $\mathbb{R}^{d+1}$. Si $Z = (X, Y) : (\Omega, \mathcal{A}, \mathbb{P}) \to F$ est de loi uniforme sur F, $X : (\Omega, \mathcal{A}, \mathbb{P}) \to \mathbb{R}^d$ est de loi μ.*

Démonstration. — Par définition, la fonction $p : \mathbb{R}^d \to \mathbb{R}_+$ est mesurable, positive et vérifie

$$1 = \int_{\mathbb{R}^d} p(x)\,\mathrm{d}x = \int_{\mathbb{R}^d} \int_0^{p(x)} \mathrm{d}y\,\mathrm{d}x = \int_{\mathbb{R}^d \times \mathbb{R}} \mathbb{1}_{\{0 \leqslant y \leqslant p(x)\}}\,\mathrm{d}x\,\mathrm{d}y = \int_{\mathbb{R}^d \times \mathbb{R}} \mathbb{1}_F(x, y)\,\mathrm{d}x\,\mathrm{d}y = \mathrm{vol}_{\mathbb{R}^{d+1}}(F)$$

par le théorème de Fubini–Tonelli (l'intégrale itérée est une intégrale multiple).

Soit $Z = (X, Y)$ de loi uniforme sur F. Pour tout borélien B de $\mathbb{R}^d$, on a

$$\begin{aligned}
\mathbb{P}\{X \in B\} &= \mathbb{P}\{Z \in B \times \mathbb{R}\} \\
&= \int_{\mathbb{R}^d \times \mathbb{R}} \mathbb{1}_{B \times \mathbb{R}}(x, y) \times \mathbb{1}_F(x, y)\,\mathrm{d}x\,\mathrm{d}y \\
&= \int_{\mathbb{R}^d \times \mathbb{R}} \mathbb{1}_B(x) \times \mathbb{1}_{\{0 \leqslant y \leqslant p(x)\}}\,\mathrm{d}x\,\mathrm{d}y \\
&= \int_{\mathbb{R}^d} \mathbb{1}_B(x) \int_0^{p(x)} \mathrm{d}y\,\mathrm{d}x = \int_{\mathbb{R}^d} \mathbb{1}_B(x)\,p(x)\,\mathrm{d}x = \mu(B),
\end{aligned}$$

d'où la conclusion. □

La combinaison des deux théorèmes précédents donne une méthode de simulation de variables aléatoires de lois complexes : supposons que le support de μ soit compact et que la fonction de densité p soit bornée. Dans ce cas nous pouvons supposer F inclus dans un pavé E de $\mathbb{R}^{d+1}$ de volume fini. Nous pouvons donc utiliser la méthode de rejet pour obtenir Z de loi uniforme sur F, puis projeter pour obtenir X de loi μ.

Exercice 10. — L'inventer soi-même.

Références

[6] Box (G.E.P.), Muller (M.E.), « A Note on the Generation of Random Normal Deviates », *The Annals of Mathematical Statistics*, vol. 29 (1958), p. 610–613.

Lois de Pearson

Si X est une variable aléatoire suivant la loi du χ^2, de Pearson, à ν degrés de liberté, la table donne, pour α fixé, la valeur $k_{1-\alpha}$ telle que

$$\mathbb{P}\{X \geqslant k_{1-\alpha}\} = \alpha.$$

Ainsi, $k_{1-\alpha}$ est le quantile d'ordre $1-\alpha$ de la loi du χ^2 à ν degrés de liberté. `1m02tp.1`

ν \ α	0.990	0.975	0.950	0.900	0.100	0.050	0.025	0.010	0.001
1	0.0002	0.0010	0.0039	0.0158	2.7055	3.8415	5.0239	6.6349	10.8276
2	0.0201	0.0506	0.1026	0.2107	4.6052	5.9915	7.3778	9.2103	13.8155
3	0.1148	0.2158	0.3518	0.5844	6.2514	7.8147	9.3484	11.3449	16.2662
4	0.2971	0.4844	0.7107	1.0636	7.7794	9.4877	11.1433	13.2767	18.4668
5	0.5543	0.8312	1.1455	1.6103	9.2364	11.0705	12.8325	15.0863	20.5150
6	0.8721	1.2373	1.6354	2.2041	10.6446	12.5916	14.4494	16.8119	22.4577
7	1.2390	1.6899	2.1673	2.8331	12.0170	14.0671	16.0128	18.4753	24.3219
8	1.6465	2.1797	2.7326	3.4895	13.3616	15.5073	17.5345	20.0902	26.1245
9	2.0879	2.7004	3.3251	4.1682	14.6837	16.9190	19.0228	21.6660	27.8772
10	2.5582	3.2470	3.9403	4.8652	15.9872	18.3070	20.4832	23.2093	29.5883
11	3.0535	3.8157	4.5748	5.5778	17.2750	19.6751	21.9200	24.7250	31.2641
12	3.5706	4.4038	5.2260	6.3038	18.5493	21.0261	23.3367	26.2170	32.9095
13	4.1069	5.0088	5.8919	7.0415	19.8119	22.3620	24.7356	27.6882	34.5282
14	4.6604	5.6287	6.5706	7.7895	21.0641	23.6848	26.1189	29.1412	36.1233
15	5.2293	6.2621	7.2609	8.5468	22.3071	24.9958	27.4884	30.5779	37.6973
16	5.8122	6.9077	7.9616	9.3122	23.5418	26.2962	28.8454	31.9999	39.2524
17	6.4078	7.5642	8.6718	10.0852	24.7690	27.5871	30.1910	33.4087	40.7902
18	7.0149	8.2307	9.3905	10.8649	25.9894	28.8693	31.5264	34.8053	42.3124
19	7.6327	8.9065	10.1170	11.6509	27.2036	30.1435	32.8523	36.1909	43.8202
20	8.2604	9.5908	10.8508	12.4426	28.4120	31.4104	34.1696	37.5662	45.3147
21	8.8972	10.2829	11.5913	13.2396	29.6151	32.6706	35.4789	38.9322	46.7970
22	9.5425	10.9823	12.3380	14.0415	30.8133	33.9244	36.7807	40.2894	48.2679
23	10.1957	11.6886	13.0905	14.8480	32.0069	35.1725	38.0756	41.6384	49.7282
24	10.8564	12.4012	13.8484	15.6587	33.1962	36.4150	39.3641	42.9798	51.1786
25	11.5240	13.1197	14.6114	16.4734	34.3816	37.6525	40.6465	44.3141	52.6197
26	12.1981	13.8439	15.3792	17.2919	35.5632	38.8851	41.9232	45.6417	54.0520
27	12.8785	14.5734	16.1514	18.1139	36.7412	40.1133	43.1945	46.9629	55.4760
28	13.5647	15.3079	16.9279	18.9392	37.9159	41.3371	44.4608	48.2782	56.8923
29	14.2565	16.0471	17.7084	19.7677	39.0875	42.5570	45.7223	49.5879	58.3012
30	14.9535	16.7908	18.4927	20.5992	40.2560	43.7730	46.9792	50.8922	59.7031

Lorsque le degré de liberté ν est tel que $\nu > 30$, la variable aléatoire

$$Z = \sqrt{2X} - \sqrt{2\nu - 1}$$

suit approximativement la loi normale centrée réduite.