Université de Poitiers
Département de Mathématiques

Probabilités Générales
M1 MFA–MMAS, année 2013–2014

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 seront effectués en Scilab. Il a été une époque où Maple avait été privilégié dans l'optique de la préparation à l'agrégation. Maple s'est avéré inadapté à nos activités (calcul scientifique et non calcul formel).

Nous conseillons de se placer dans un répertoire de travail spécifique `probgene` (par exemple) pour y stocker les scripts (suites d'instructions) Scilab qu'on suffixera `*.sce` (*SCilab Executable*) ou `*.sci` (*SCilab Input*), les graphiques PostScript ou PDF (`*.ps`, `*.eps`, `*.pdf`), et les ébauches de rapport tapées en TEX ou LATEX (`*.tex`). Avoir un clavier et un écran d'ordinateur devant soi ne saurait dispenser d'utiliser un crayon et du papier...

La saisie des scripts peut se faire avec un éditeur de texte quelconque (celui du logiciel devrait convenir, mais `emacs` est aussi très bien). 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 messages d'erreurs ni sorties longues plus ou moins utiles. De plus, on peut, doit, intercaler dans un script des lignes de commentaires décrivant ce qu'on fait, le mécanisme d'une commande, une remarque judicieuse, un résultat numérique pertinent.

L'exécution d'un script ⟨*script*⟩`.sce` avec Scilab se fait (sous Linux) en tapant depuis la ligne de commande

`--> exec ⟨script⟩.sce` ou `--> exec /home/⟨toto⟩/probgene/⟨script⟩.sce`

ou encore

`--> exec "/home/⟨toto⟩/probgene/⟨script⟩.sce"`

si le chemin comporte des espaces ou certains caractères spéciaux. Les chemins sous Windows s'écrivent un peu différemment.

Les séances de travaux pratiques donneront lieu à un rapport unique et individuel (*voir* les documents de rentrée pour ce qui est attendu d'un compte-rendu de travaux pratiques).

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 modestement, 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 ultimement 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 implantation, 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 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).

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

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

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 qu'on doive 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 1 (*pratique*). — Nous allons définir une fonction `randomi()` avec SCILAB dont chaque appel retourne la nouvelle valeur de $(x_n)_{n \geqslant 0}$ calculée par la méthode des congruences linéaires. Pour ce faire, il suffit ici de connaître le terme précédent, de calculer le nouveau terme et de conserver la valeur de ce dernier dans la variable qui avait contenu le terme précédent. Cette variable sera nommée $\langle$*randomseed*$\rangle$. Comme les valeurs de $\langle a \rangle$, $\langle c \rangle$ et $\langle m \rangle$, $\langle$*randomseed*$\rangle$ doit être connue dans la mémoire de SCILAB de manière globale et ses modifications doivent être prises en compte aussi globalement.

```
clear; // nettoyage de la m\'emoire de Scilab
mode(0); // mode presque silencieux

global a c m randomseed;
a = ???; c = ???; m = ???;
randomseed = ???; // remplacer les ??? par des valeurs convenables

function x = randomi()
    global randomseed; // a c m devraient \^etre connus
```

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

```
    randomseed = pmodulo(a*randomseed+c, m);
    x = randomseed;
endfunction
```

la fonction `pmodulo` correspond au vrai « modulo » mathématique, son fonctionnement diffère de la fonction `modulo` lorsque son premier argument est un nombre négatif (*voir* l'aide de SCILAB).

(i) Définir à la suite la fonction `random()` retournant `randomi()/m`, c'est-à-dire correspondant à la suite $(u_n)_{n\geqslant 0} = (x_n/m)_{n\geqslant 0}$.

(ii) Quitte à changer les valeurs de $\langle a\rangle$, $\langle c\rangle$ et $\langle m\rangle$, comparer le générateur obtenu avec celui de SCILAB. (Pour une graine identique, on doit avoir une coïndence approximative de `randomi()` 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) Nous allons étudier naïvement si la commande `random()` produit des suites de nombres qui tendent approximativement à se répartir uniformément dans $[0, 1]$:

```
N = ???; // remplacer ??? par une valeur convenable
x = zeros(N, 1); // initialisation de l'\'echantillon
for i = 1:N; x(i) = random(); end
```

Représenter un histogramme des valeurs de $\langle x\rangle$. Ceci se fait avec la commande

`histplot(`$\langle k\rangle$`,` $\langle x\rangle$`)`

où $\langle k\rangle$ est le nombre de classes (*voir* `--> help histplot`).

Remarque. — Pour n observations, la règle de Sturges recommande de prendre

$$k = \mathrm{ceil}\big(1 + 3.322 \times \log_{10}(n)\big)$$

classes pour l'histogramme, où ceil est la fonction partie entière supérieure. Il est conseillé de définir une bonne fois pour toute une fonction `sturges(`$\langle n\rangle$`)` calculant ce nombre de classes.

```
function k = sturges(n)
    k = ceil(1+3.322*log10(n));
endfunction
```

(iv) (MAPLE) Vérifier que $m = 10^{12} - 11$ est premier (`isprime(`$\langle m\rangle$`);`). 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 au choix des programmeurs MAPLE ?

EXERCICE 2 (*pratique*). — (i) On souhaite simuler le résultat d'un lancer de dé équilibré, c'est-à-dire la loi uniforme sur $\{1, 2, 3, 4, 5, 6\}$, qui est discrète et se représente à l'aide d'un diagramme en bâtons. On procédera de deux manières différentes : la première en utilisant les retours de `randomi()` modulo 6 ; la seconde en utilisant la partie entière de `random()` multipliée par 6.

```
function x = dice()
    x = pmodulo(randomi(),6)+1;
endfunction
```

```
function x = Dice()
    x = floor(6*random())+1;
endfunction
```

Tester depuis la console le bon fonctionnement de ces deux nouvelles commandes.

(ii) Plutôt que de générer une suite `x` de `N` valeurs produites successivement par l'une ou l'autre des fonctions `dice()` et `Dice()`, nous allons comptabiliser les nombres d'apparitions de $1, \dots, 6$ au cours de ces appels successifs, puis ramener ces nombres aux fréquences correspondantes.

```
pobs = zeros(6, 1); // vecteur colonne des fr\'equences observ\'ees
for i = 1:N;
    outcome = dice(); // ou Dice(), ‘‘outcome’’ signifie r\'esultat
    pobs(outcome) = pobs(outcome)+1;
end
pobs = pobs/N;
```

Le résultat est à comparer avec la loi uniforme sur $\{1, \dots, 6\}$.

```
xtheo = [1:6]'; // vecteur colonne des valeurs possibles
ptheo = ones(6,1)/6; // le vecteur colonne [1/6; 1/6; 1/6; 1/6; 1/6; 1/6]
```

La comparaison graphique se fait naturellement à l'aide d'un diagramme en batôns. Avec Scilab, il s'obtient avec la commande `bar(`$\langle$*absisses*$\rangle$`,` $\langle$*ordonnées*$\rangle$`)`.

```
scf(1); clf();
bar(xtheo, [ptheo, pobs]);
title("Lancer d''un de avec la commande dice()");
legend(["distribution theorique"; "distribution observee"]);
```

Remarque. — On notera au passage plusieurs commandes Scilab : `scf(`$\langle$*numero*$\rangle$`)` qui sélectionne la fenêtre graphique courante (*set current figure*) et permet ainsi au cours de l'exécution d'un script de créer plusieurs figures (`clf()` sans argument nettoie par défaut la figure courante), `title(`$\langle$*chaîne de caractères*$\rangle$`)` qui permet d'ajouter un titre à la figure, et finalement `legend(`$\langle$*vecteur de chaînes de caractères*$\rangle$`)` dont le rôle est évident.

On notera aussi que la commande `bar()` peut admettre comme second argument une matrice. Il faut prendre garde à la cohérence de ses arguments. Nous avons choisi de donner les abscisses sous la forme d'un vecteur colonne $\langle xtheo \rangle$, la matrice des ordonnées est obtenue en juxtaposant les deux vecteurs colonne $\langle ptheo \rangle$ et $\langle pobs \rangle$, juxtaposition obtenue par la virgule dans `[`$\langle$*ptheo*$\rangle$`,` $\langle$*pobs*$\rangle$`]` — un point-virgule aurait conduit à la mise bout à bout des deux vecteurs et abouti à des arguments incohérents.

Il est certainement possible de donner les abscisses sous la forme d'un vecteur ligne. Dans ce cas la donnée des ordonnées doit respecter une règle semblable de cohérence.

Nous conseillons bien évidemment de se reporter à l'aide pour avoir une description plus précise de ces commandes.

2. Un aspect statistique

Précédemment ainsi que dans ce qui suit pour cette séance de travaux pratiques, la suite $\langle x \rangle$ obtenue est censée représenter un *échantillon* (*sample* en anglais) d'une loi μ sur $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ (pour l'instant 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 $[0, 1[$:

$$F : \mathbb{R} \longrightarrow [0, 1]$$

$$x \longmapsto F(x) = \min\big(\max(x, 0), 1\big) = \begin{cases} 0 & \text{si } x \leqslant 0 \\ x & \text{si } x \in [0, 1] \\ 1 & \text{si } x \geqslant 1. \end{cases}$$

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 la commandes `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'adéquation (ici, avec la loi uniforme sur $[0, 1]$) 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

$$p\text{-valeur} = 1 - F_{K_n}(k)$$

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 % (ce qui exprime que k est trop grand), on rejette l'hypothèse d'adéquation.

Évidemment, un seul test sur le générateur ne suffit pas pour décider s'il convient ou non. D'ailleurs, si le générateur fonctionne correctement, la p-valeur doit être distribuée uniformément sur $[0, 1]$. Ceci suggère de générer un grand nombre d'échantillons, d'enregistrer les p-valeurs correspondantes et de faire un test de Kolmogorov–Smirnov sur l'adéquation de la distribution des p-valeurs avec la loi uniforme sur $[0, 1]$, *ad libitum*...

Avec Scilab (et Maple), les fonctions de répartition 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 3 (*pratique*). — (i) Après avoir généré un échantillon $\langle x \rangle$ de taille $\langle N \rangle$ à l'aide de la commande `random()`, on compare graphiquement la fonction de répartition observée avec la fonction de répartition cible, ici celle de la loi uniforme sur $[0, 1]$.

```
y = -gsort(-x); // pour un tri dans le sens croissant
rang = [1:N]'/N; // 1/N, 2/N, ..., N/N=1

scf(2); clf();
plot2d2(y, rang); // graphe en escalier
plot2d([-0.1; 0; 1; 1.1], [0; 0; 1; 1], style=5); // en rouge
```

Noter que le tracé de la fonction de répartition de la loi uniforme sur $[0, 1]$ ne nécessite que celui d'une ligne brisée très simple.

(ii) Définir une fonction une fonction `KSstat(`$\langle x \rangle$`, `$\langle F \rangle$`)` calculant la statistique de Kolmogorov–Smirnov associée à une suite de nombres $\langle x \rangle$ et une fonction de répartition $\langle F \rangle$. Le paramètre $\langle F \rangle$ de `KSstat` est une fonction définie par exemple avec

```
deff("p = F(x)", "p = min(max(x, 0), 1)");
```

dans le cas de la loi uniforme sur $[0,1]$. On peut bien sûr remplacer la lettre `F` par un nom qui serait plus adapté.

(iii) Utiliser cette fonction sur des petits échantillons obtenus par la méthode des congruences linéaires et censés se répartir uniformément sur $[0,1]$. Comparer les valeurs obtenues avec le ou les quantiles d'ordre $1-\alpha = 0{,}95$ correspondants dans la table des quantiles des lois de Kolmogorov.

(iv) Dudley (1964) a montré que pour tout $u > 0$,

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

De cette formule asymptotique, on peut définir une fonction `klmcdf(`$\langle x\rangle$`, `$\langle n\rangle$`)` approchant raisonnablement $F_{K_n}(x)$ en notant bien que la série de Dudley est alternée et qu'ainsi on contrôle assez bien l'erreur. Ceci peut donner avec Scilab (avec un ajustement numérique recommandé par Stephens M.A., 1970) :

```
function p = klmcdf(x, n)
    if x <= 0 then p = 0;
    elseif x >= 1 then p = 1;
    else
        // local z p term k pm; global accuracy;
        accuracy = 1e-12;
        z = -2*(sqrt(n)+0.12+0.11/sqrt(n))^2*x*x;
        p = 1.0; k = 1; pm = -1; term = 1;
        while abs(term) > accuracy & k < 100;
            term = 2*pm*exp(z*k*k);
            p = p+term; pm = -pm; k = k+1;
        end
        p = max(min(p, 1), 0);
    end
endfunction
```

Saisir ce code, le comprendre. Utiliser cette fonction pour calculer une approximation de la p-valeur du test de Kolmogorov–Smirnov.

Remarques. — *a*) Des implantations des fonctions de répartition des lois de Kolmogorov ainsi que l'approximation de Dudley se trouvent dans le répertoire

`http://www-math.univ-poitiers.fr/~phan/downloads/goodies/`

sous la forme d'un fichier d'instructions `kolmogorov.sci` pour Scilab (et `kolmogorov.txt` pour Maple). Ultérieurement, on pourra récupérer ce fichier, examiner rapidement son contenu et l'utiliser à l'avenir. Cependant, une fonction telle que `klmcdf` est largement satisfaisante ici puisque les tailles d'échantillons sont grandes (l'approximation de Dudley est considérée par certains comme légitime dés que $n \geqslant 5$).

b) Il n'existe pas de commande `local` en Scilab. Néanmoins, il est intéressant de préciser dans ses programmes si les variables rencontrées sont locales ou non, même si cela doit se présenter sous la forme de commentaires.

3. Autres générateurs de nombres aléatoires

Scilab propose la commande `grand()` qui permet de simuler un certains nombres de lois classiques (`--> help grand`). Elle utilise pour ce faire des techniques parfois avancées avec sous-jacents différents types de générateurs aléatoires. Citons parmi eux le *Mersenne Twister* qui est un algorithme récent (1997, les initiales des noms des auteurs sont M. et T.) et qui commence à être suffisamment sérieusement éprouvé pour parvenir à s'imposer à la place des congruences linéaires. D'ailleurs, la commande `rand()` de Maple est désormais basée sur le *Mersenne Twister* 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).

Par ailleurs, mentionnons le générateur aléatoire implanté 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 assez 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... (*Voir* `man rand` sous Linux par exemple.)

4. Un exemple historique : la méthode du carré médian (facultatif)

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 résolution de problèmes ou de calculs mathématiques jusqu'alors 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 4 (*théorique*). — Quel nombre suit 1010101010 dans la méthode du carré médian (essayer de le calculer de tête) ?

Exercice 5 (*théorique*). — (i) 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.

(ii) Avec les hypothèses de la question précédente, 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 6 (*pratique*). — (i) Définir une fonction `msrandomi()` (*Middle Square Random Integer*) sans argument implantant cette méthode et retournant la nouvelle valeur de la suite. Ses paramètres globaux pourront être `msrandomseed`, etc., initialisés par une procédure `setmsrandom` (*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 `msrandom()` en sera déduite pour générer des nombres dans $[0, 1]$ de manière uniforme (on l'espère).

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

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 7 (*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.

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] 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

Université de Poitiers
Département de Mathématiques

Probabilités Générales
M1 MFA–MMAS, année 2013–2014

TRAVAUX PRATIQUES. — GÉNÉRATEURS ALÉATOIRES, ÉLÉMENTS D'EXPLICATIONS

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

EXERCICE 1. — Voici le programme SCILAB.

```
clear; // nettoyage de la m\'emoire de Scilab
mode(0); // mode presque silencieux

// TP1, exercice 1

global a c m randomseed;
a = 7^5; c = 0; m = 2^31-1;
randomseed = 1;

function x = randomi()
    global randomseed; // a c m
    randomseed = pmodulo(a*randomseed+c, m);
    x = randomseed;
endfunction

function u = random()
    u = randomi()/m;
endfunction

N = 1000;
x = zeros(N, 1); // initialisation de l'\'echantillon
for i = 1:N; x(i) = random(); end

function k = sturges(n)
    k = ceil(1+3.322*log10(n));
endfunction

scf(0); clf();
histplot(sturges(6), x);
```

EXERCICE 2. — Voici la suite du programme SCILAB.

```
// TP1, exercice 2

function x = dice()
    x = pmodulo(randomi(),6)+1;
endfunction

function x = Dice()
    x = floor(6*random())+1;
endfunction
```

```
pobs = zeros(6, 1); // vecteur colonne des fr\'equences observ\'ees
for i = 1:N;
    outcome = dice(); // ou Dice()
    pobs(outcome) = pobs(outcome)+1;
end
pobs = pobs/N;

xtheo = [1:6]'; // vecteur colonne des valeurs possibles
ptheo = ones(6,1)/6;

scf(1); clf();
bar(xtheo, [ptheo, pobs]);
title("Lancer d''un de avec la commande dice()");
legend(["distribution theorique"; "distribution observee"]);
```

2. **Un aspect statistique**

Exercice 3. — Voici la fin du programme Scilab.

```
// TP1, exercice 3

x = zeros(N, 1); // initialisation de l'\'echantillon
for i = 1:N; x(i) = random(); end

y = -gsort(-x); // pour un tri dans le sens croissant
rang = [1:N]'/N; // 1/N, 2/N, ..., N/N=1

scf(2); clf();
plot2d2(y, rang); // graphe en escalier
plot2d([-0.1; 0; 1; 1.1], [0; 0; 1; 1], style=5); // en rouge

// Commandes pour le test de Kolmogorov--Smirnov

function k = KSstat(x, F)
    // local y n i;
    n = size(x,1);
    y = -gsort(-x); // sort() n'existe plus (attention \`a l'ordre)
    k = 0;
    for i = 1:n; k = max(k, F(y(i))-(i-1)/n, i/n-F(y(i))); end
endfunction

function p = klmcdf(x, n)
    if x <= 0 then p = 0;
    elseif x >= 1 then p = 1;
    else
        // local z p term k pm; global accuracy;
        accuracy = 1e-12;
        z = -2*(sqrt(n)+0.12+0.11/sqrt(n))^2*x*x;
        p = 1.0; k = 1; pm = -1; term = 1;
        while abs(term) > accuracy & k < 100;
            term = 2*pm*exp(z*k*k);
            p = p+term; pm = -pm; k = k+1;
        end
        p = max(min(p, 1), 0);
    end
endfunction
```

```
// Mise en \oe uvre du test

deff("p = F(x)", "p = min(max(x, 0), 1)");
k = KSstat(x, F); pvaleur = 1-klmcdf(k, N);
```

3. **Autres générateurs de nombres aléatoires**

Rien à dire.

4. **Un exemple historique : la méthode du carré médian**

Exercice 4. — On a

$$\begin{aligned} 1010101010^2 &= 1010101010 \times (1000000000 + 10000000 + 100000 + 1000 + 10) \\ &= 1010101010000000000 \\ &+ \quad 10101010100000000 \\ &+ \quad\quad 101010101000000 \\ &+ \quad\quad\quad 1010101010000 \\ &+ \quad\quad\quad\quad 10101010100 \\ &= 1020304050403020100 \end{aligned}$$

d'où le résultat 3040504030.

Exercice 5. — (i) Puisque $x < b^n$, on a $x^2 < b^{2n}$, et le carré médian est $\lfloor x^2/b^n \rfloor \leqslant x^2 < b^n$. Si $x > 0$, alors $x^2/b^n < xb^n/b^n = x$.

(ii) Si $x = ab^n$, le nombre suivant est de la même forme, il est égal à $(a^2 \bmod b^n)b^n$. Si a est un multiple de tous les facteurs premiers de b, la suite va bientôt dégénérer en 0 ; sinon, la suite va dénérer en cycles de nombres de la même forme que x.

Si les $n+1$ chiffres les moins significatifs sont nuls, alors, au rang suivant, les $n+2$ chiffres les moins significatifs sont nuls, etc. La suite converge donc rapidement vers 0.

Exercice 6. — Avec Scilab,

```
// TP 1, exercice 6
// Implantation de la m\'ethode du carr\'e m\'edian

clear(); mode(0);

function k = sturges(n)
    k = ceil(1+3.322*log10(n));
endfunction

global msfactor msrandomseed;

function setmsrandom(seed, base, wordlength)
    global msrandomseed msfactor;
    msfactor = base^ceil(.5*wordlength);
    msrandomseed = pmodulo(seed, msfactor^2); // normalisation
endfunction

function x = msrandomi()
    global msrandomseed; // msfactor devrait \^etre connu
    msrandomseed = floor(pmodulo(msrandomseed^2, msfactor^3)/msfactor);
    x = msrandomseed;
endfunction
```

```
function x = msrandom()
    // msfactor devrait \^etre connu
    x = msrandomi()/msfactor^2;
endfunction

setmsrandom(123456, 10, 6); N = 100; u = zeros(N, 1);
for i = 1:N; u(i) = msrandom(); end
scf(0); clf(); histplot(sturges(N), u);
```

Exercice 7. — (i) et (ii) 00, 00, ... (62 valeurs initiales) ; 10, 10, ... (19 valeurs initiales) ; 60, 60, ...(15 valeurs initiales), 50, 50, ... (1 valeur initiale) ; 24, 57, 24, 57 (3 valeurs initiales).
(iii) 42 ou 69 ; ces deux valeurs initiales mènent toutes deux vers un ensemble de 15 valeurs distinctes : (42 ou 69), 76, 77, 92, 46, 11, 12, 14, 19, 36, 29, 84, 05, 02, 00.

Nous donnons un « corrigé pratique » cet exercice. 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 Scilab, ce qui, *a posteriori*, était assez facile. On notera les éléments de programmation : boucles `for` et `while`, tests, fonctions d'affichage élémentaires `disp`, `mprintf`, les manipulations de chaînes de caractères...

On notera dans le code Scilab que nous avons choisi de préciser en commentaires quelles étaient les variables locales du programme. Bien que la commande `local` n'existe 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.

L'affichage est obtenu à l'aide de la commande `disp`. On va voir que ça ne donne pas un code très élégant. La commande Scilab `mprintf` est l'exacte copie de `printf` (provenant en fait du C). Elle peut avantageusement remplacer `disp` dans ce programme.

```
// code scilab
// TP1, exercice 7
// 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("p = F(x)", "p = floor(pmodulo(x*x,1000)/10)"); Orbits(F, 100);
```

Annexe : équivalents Maple

Ce code a été testé avec Maple 9.5 ainsi que Maple 12. En commentant et décommentant les bonnes lignes, cela fonctionne aussi avec Maple 6.

```
# code maple
# TP 1, exercice 1

#with(Statistics): # maple v >= 9.5
with(stats):
Histogram := proc(u::array, others)
    local n;
    n := op(2,op(2,eval(u)));
    printf("Histogramme de %d donnees", n);
    stats[statplots, histogram]([seq(u[i], i = 1..n)],
    area=1, numbars=sturges(n));
end proc:

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

randomi := proc()
    global a, c, m, randomseed;
    randomseed := modp(a*randomseed+c, m);
    return randomseed;
end proc:
```

```
random := proc()
    global m;
    return evalf(randomi()/m);
end proc:

N := 1000: x := array(1..N):
for i from 1 to N do x[i] := random(): end do:

sturges := proc(n)
    return ceil(1+3.322*log10(n));
end proc:

Histogram(x, bincount = sturges(N), discrete = false);

# TP1, exercice 2

dice := proc()
    return randomi() mod 6+1;
end proc:

Dice := proc()
    return floor(6*random())+1;
end proc:

pobs := [0, 0, 0, 0, 0, 0]:
for i from 1 to N do
    outcome := dice();
    pobs[outcome] := pobs[outcome]+1;
end do:
pobs := pobs/N:

xtheo := [seq(i, i = 1..6)]:
ptheo := [seq(1/6, i = 1..6)]:

# Diagramme en b\^atons assez laborieux

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

bar := proc(xtheo, ptheo, pobs)
    local k, dx, i;
    k := size(xtheo);
    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; # par exemple
    # 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:

bar(xtheo, ptheo, pobs);

# TP1, exercice 3

N := 1000: x := array(1..N):
for i from 1 to N do x[i] := random(): end do:

y := sort([seq(x[i], i = 1..N)]):
PLOT(
    CURVES([[0,0], [1,1]], COLOR(RGB,.0,.0,1.0)),
    CURVES([seq([y[i], i/N], i = 1..N)], COLOR(RGB,1.0,.0,.0))
);

# Commandes pour le test de Kolmogorov--Smirnov

KSstat := proc(u, F)
    local k, 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;
    k := 0;
    for i from 1 to n do
        k := evalf(max(k, F(v[i])-(i-1)/n, i/n-F(v[i])));
    end do;
    return k;
end proc:

accuracy := 1e-12:

klmcdf := proc(x, n)
    global accuracy;
    local z, p, term, k, pm;
    if x <= 0 then return 0;
    elif x >= 1 then return 1;
    else
        z := evalf(-2*(sqrt(n)+0.12+0.11/sqrt(n))^2*x*x);
        p := 1.0; k := 1; pm := -1; term := 1;
        while abs(term) > accuracy and k < 100 do
            term := evalf(2*pm*exp(z*k*k));
            p := evalf(p+term); pm := -pm; k := k+1;
```

```
        end do;
        return evalf(max(min(p, 1), 0));
    end if;
end proc:

# Mise en \oe uvre du test

F := x -> x:
k := KSstat(x, F): pvaleur := 1-klmcdf(k, N):
printf("Statistique de Kolmogorov-Smirnov k = %f\np-valeur = %f\n", k, pvaleur);

# TP1, exercice 6
# Implantation de la m\'ethode du carr\'e m\'edian

setmsrandom := proc(seed, base, wordlength)
    global msrandomseed, msfactor;
    msfactor := base^ceil(.5*wordlength);
    msrandomseed := seed mod msfactor^2;# normalisation
end proc:

msrandomi := proc()
    global msrandomseed;
    # x&^y mod z : astuce de Perrin-Riou, p. 18
    msrandomseed := floor(msrandomseed&^2 mod msfactor^3/msfactor);
    return msrandomseed;
end proc:

msrandom := proc()
    global msfactor;
    return evalf(msrandomi()/msfactor^2);
end proc:

setmsrandom(123456, 10, 6):
N := 100: u := array(1..N):
for i from 1 to N do u[i] := msrandom(): end do:
Histogram(u, bincount = sturges(N), discrete = false);

# Exercice 7
# 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;
    end do:
    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);
```

Université de Poitiers
Département de Mathématiques

Probabilités Générales
M1 MFA–MMAS, année 2013–2014

TRAVAUX PRATIQUES. — SIMULATIONS DE VARIABLES ALÉATOIRES

Introduction

Précédemment, nous avons expérimenté quelques méthodes destinées à simuler la réalisation d'une suite $(U_n)_{n\geqslant 1}$ de variables aléatoires indépendantes, identiquement distribuées, de loi uniforme sur $[0,1[$ (l'ouverture ou la fermeture des bornes ne change rien théoriquement, mais cela peut compter en pratique). Tout logiciel de calcul scientifique dispose d'un tel générateur (plus ou moins bon). Pour SCILAB, il y a la commande `rand()`, maintenant dépréciée (*voir* l'aide), et la commande `grand()` qui repose sur des algorithmes plus récents. Cette commande permet de générer des tableaux de dimension $n \times m$ suivant des lois classiques via `grand(`⟨n⟩`,` ⟨m⟩`, "`⟨*loi*⟩`",` ⟨*paramètres éventuels*⟩`)`. Nous ne nous servirons que d'une petite partie de ses potentialités en limitant chaque appel à cette commande à un retour d'un nombre unique (tableau 1×1) tiré suivant la loi uniforme sur $[0,1[$. Pour nous simplifier la tâche, nous définissons la commande `random()` :

```
function u = random()
    u = grand(1, 1, "def");
endfunction
```

Cette commande retournera des nombres dans $[0,1[$ avec une apparence d'indépendance et de répartition uniforme sur cet intervalle.

Remarque. — La commande `random()` définie dans la feuille précédente satisferait largement nos besoins. Il y a derrière la commande `grand()` des algorithmes de génération de nombres au hasard en vogue tel que le Mersenne Twister — M.T. sont les initiales des noms de leurs deux inventeurs (*voir* wikipedia).

1. Pour commencer

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 random() < 1-p then return 0; else return 1; fi
enddefinition;
```

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

```
% pseudo-code \‘a traduire et tester
definition discrete(list ptheo) = % ptheo = p_1,...,p_n
    local u, i, p;
    u = random(); i = 0;
    for p = ptheo;
        i = i+1; u = u-p;
        if u <= 0 then break; fi
    endfor
    return i;
enddefinition;
```

On pourra tester cette nouvelle commande avec la loi uniforme sur $\{1, \ldots, 6\}$

```
ptheo = ones(6, 1)/6; // vecteur colonne
```

ainsi que sur une loi binomiale $\mathcal{B}(n, p)$ de paramètres bien choisis. Le vecteur colonne correspondant à cette loi est facilement calculable avec SCILAB :

```
ptheo = binomial(p, n)'; // vecteur colonne
```

Remarque. — Avec SCILAB, la liste donnée pour la boucle `for` doit se présenter sous la forme d'une ligne, ainsi, si liste $\langle ptheo \rangle$ est donnée sous la forme d'un vecteur colonne à la commande, il faudra penser à la transposer pour la boucle `for`. Ne pas oublier que SCILAB n'a pas de commande `local`.

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 μ portée par un ensemble fini $E = \{e_1, \ldots, e_k\}$ et notons $\mu_i = \mu\{e_i\} > 0$. 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 μ si tous les p_i sont proches des μ_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”*)

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

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

$$p\text{-valeur} = 1 - F_{k-1}(cs).$$

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 μ. Notons que si l'échantillon comporte des valeurs x_j pour lesquelles $\mu\{x_j\} = 0$, on rejette l'hypothèse sans même se poser de question : l'échantillon ne peut en aucun cas avoir été tiré suivant la loi μ.

Exercice 2. — Calculer les p-valeurs correspondant aux simulations précédentes. (*Voir plus bas pour l'usage de* `cdfchi`.)

3. Quelques programmes utiles

On trouvera à l'adresse

`http://www-math.univ-poitiers.fr/~phan/downloads/goodies/`

quelques programmes qui seront utiles pour la suite. Tout d'abord, il y a le programme `kolmogorov.sci` ou des fonctions permettant de calculer de manière exacte ou approchée les fonctions de répartition des lois de Kolmogorov ainsi que leurs inverses.

- `kolmogorovcdf(`$\langle x\rangle$`, `$\langle n\rangle$`)` calcule de manière exacte la fonction de répartition en x de la loi de Kolmogorov de paramètre n, c'est à dire $\mathbb{P}\{K_n \leqslant x\}$. Il n'est pas raisonnable de l'utiliser pour $n \geqslant 100$.
- `kolmogorovicdf(`$\langle p\rangle$`, `$\langle n\rangle$`)` calcule l'inverse de la fonction de répartition de la loi de Kolmogorov de paramètre n, c'est-à-dire détermine x tel que $\mathbb{P}\{K_n \leqslant x\} = p$.
- `klmcdf(`$\langle x\rangle$`, `$\langle n\rangle$`)` approche `kolmogorovcdf(`$\langle x\rangle$`, `$\langle n\rangle$`)` par la formule de Dudley. Cette approximation convient bien aux simulations pour lesquelles on a $n \geqslant 100$.
- `klmicdf(`$\langle p\rangle$`, `$\langle n\rangle$`)` inverse la fonction précédente par une simple méthode de dichotomie.

Scilab possède une fonction pour les calculs autour des lois du χ^2, il s'agit de `cdfchi`. Ainsi,

- `cdfchi("PQ", `$\langle x\rangle$`, `$\langle \nu\rangle$`)` retourne la valeur de la fonction de répartition en x de la loi du χ^2 à ν degrés de liberté ;
- `cdfchi("X", `$\langle \nu\rangle$`, `$\langle p\rangle$`, 1-`$\langle p\rangle$`)` inverse la fonction précédente.

Le programme `goodness-of-fit.sci` (*goodness of fit test* est l'expression anglo-saxonne consacrée pour test d'adéquation) définit quelques fonctions qui automatisent partiellement la réalisation des tests de Kolmogorov–Smirnov et du χ^2 d'adéquation à une loi discrète.

- `KSstat(`$\langle x\rangle$`,`$\langle F\rangle$`)` calcule la statistique Kolmogorov–Smirnov $K = \|F_n - F\|_\infty$ associée au test correspondant. L'argument $\langle x\rangle$ doit être un vecteur colonne correspondant aux observations, l'argument $\langle F\rangle$ est quant à lui le nom d'une fonction définie avec Scilab tenant lieu de fonction de répartition (continue) cible de ce test.
- `cgoftest(`$\langle x\rangle$`,`$\langle F\rangle$`)` trace les fonctions de répartition observée et cible, et, le plus important, calcule la statistique de Kolmogorov–Smirnov, la p-valeur exacte lorsque l'échantillon est de taille inférieure ou égale à 100, et dans tous les cas la p-valeur approchée. Si $\langle x\rangle$ est un tableau de taille $n \times m$, alors les m colonnes sont considérées comme représentant m échantillons. Dans ce cas, les m p-valeurs sont enregistrées et on teste si elles se répartissent uniformément dans $[0, 1]$ (test approfondi de la qualité de simulation). Le nom `cgoftest` vient de *continuous goodness of fit test*.

- `dgoftest(`$\langle x\rangle$`,`$\langle$*valeurs possibles*$\rangle$`,` $\langle$*loi théorique*$\rangle$`)` trace les diagramme en batôns des lois, et, le plus important, calcule la statistique du χ^2 d'adéquation, la p-valeur correspondante. Les trois arguments sont des vecteurs colonnes, $\langle x\rangle$ est l'échantillon observé, le vecteur des valeurs possibles correspond à ce qui a été précédemment noté $E = \{e_1, \dots, e_k\}$, et la loi théorique est le vecteur (colonne toujours) $\mu = (\mu_1, \dots, \mu_k)$. Les conditions asymptotiques de validité du test ($n \geqslant 30$ et $n_i \geqslant 5$) sont testées en cours de calcul et des messages d'avertissement peuvent apparaître. Il est absolument nécéssaire d'en tenir compte. Si $\langle x\rangle$ est un tableau de taille $n \times m$, alors les m colonnes sont considérées comme représentant m échantillons. Dans ce cas, les m p-valeurs sont enregistrées et on teste si elles se répartissent uniformément dans $[0, 1]$ (test approfondi de la qualité de simulation). Le nom `dgoftest` vient de *discrete goodness of fit test*.

Nous aurons à nous servir directement seulement des commandes `cgoftest` et `dgoftest`. Il faut pour cela charger dans la mémoire de Scilab toutes les fonctions nécessaires via

```
clear; mode(0);
exec kolmogorov.sci;
exec goodness-of-fit.sci;
```

en début de programme, ou, sous Linux,

```
clear; mode(0);
exec /home/mon_login/mon_repertoire/.../kolmogorov.sci;
exec /home/mon_login/mon_repertoire/.../goodness-of-fit.sci;
```

si les fichiers ne se trouvent pas dans le répertoire de travail, le chemin devant être entouré de guillemets s'il comporte des espaces ou des caractères spéciaux.

4. **Utilisation de la fonction de répartition**

Exercice 3 (*théorique*). — Soient μ une mesure de probabilité sur $\mathbb{R}$ et F sa fonction de répartition, c'est-à-dire $F(x) = \mu(]-\infty, x])$ pour tout $x \in \mathbb{R}$.

Définissons l'inverse généralisé F_{g}^{-1} de F par

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

(qui est une fonction croissante, continue à gauche et limitée à droite, et à valeurs dans $\bar{\mathbb{R}}$). Soient U une variable aléatoire de loi uniforme sur $[0, 1]$ et $X = F_{\mathrm{g}}^{-1} \circ U$. 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 μ.

Correction. — Soit $u \in]0, 1[$. Puisque la fonction F est croissante, $I(u) = \{y \in \mathbb{R} : F(y) \geqslant u\}$ est un intervalle, non vide car F tend vers $1 > u$ en $+\infty$, et borné à gauche puisque F tend vers $0 < u$ en $-\infty$. Montrons qu'il est fermé : soient $x = \inf I(u)$, et $(x_n)_{n\geqslant 1}$ une suite de points de $I(u)$ décroissant vers x. On a $F(x_n) \geqslant u$ pour tout $n \geqslant 1$, et puisque F est continue à droite, on a aussi $F(x) \geqslant u$. Donc $x \in I(u) = [x, +\infty[$.

Notons que $I(0) = \mathbb{R}$ et $I(1)$ est un intervalle fermé éventuellement vide.

Soit $X = F_{\mathrm{g}}^{-1} \circ U$ qui est bien une variable aléatoire puisque l'inverse généralisé $F_{\mathrm{g}}^{-1} : [0, 1] \to \bar{\mathbb{R}}$ est mesurable (puisque monotone par exemple). Pour $x \in \mathbb{R}$, nous avons alors

$$\begin{aligned}\{X \leqslant x\} &= \{F_{\mathrm{g}}^{-1} \circ U \leqslant x\} = \big\{\inf\{y \in \mathbb{R} : F(y) \geqslant U\} \leqslant x\big\} \\ &= \{x \in I(U)\} = \big\{x \in \{y \in \mathbb{R} : F(y) \geqslant U\}\big\} = \{F(x) \geqslant U\},\end{aligned}$$

l'antépénultième égalité résultat du fait que $I(U)$ est un intervalle (aléatoire) fermé. Ainsi $\mathbb{P}\{X \leqslant x\} = \mathbb{P}\{U \leqslant F(x)\} = F(x)$. D'où le résultat : la loi de X est μ.

Exercice 4 (*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 à l'aide des commandes

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

et

`cdfnor("X",` $\langle m \rangle$`,` $\langle \sigma \rangle$`,` $\langle p \rangle$`, 1-`$\langle p \rangle$`)`

avec ici $m = 0$ et $\sigma = 1$).

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

5. **Utilisation de propriétés indirectes**

Exercice 5 (*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 6 (*théorique*). — Soient $\Theta : (\Omega, \mathcal{A}, \mathbb{P}) \to [0, 2\pi[$ de loi uniforme sur $[0, 2\pi[$ et $R : (\Omega, \mathcal{A}, \mathbb{P}) \to \mathbb{R}_+$ de loi admettant pour densité

$$r \in \mathbb{R}, \qquad p_R(r) = \mathbb{1}_{\mathbb{R}_+}(r) \times r\,\mathrm{e}^{-r^2/2}.$$

(i) Montrer que $X = R\cos\Theta$ et $Y = R\sin\Theta$ sont deux variables aléatoires de loi $\mathcal{N}(0,1)$ indépendantes. Établir la réciproque.

(ii) Soient U_1 et U_2 deux variables aléatoires indépendantes de loi uniforme sur $[0,1]$. En posant

$$R = \sqrt{-2\ln(1-U_1)} \qquad \text{et} \qquad \Theta = 2\pi U_2.$$

constater que (R, Θ) vérifient les hypothèses précédentes. En déduire une méthode de simulation de la loi normale $\mathcal{N}(0,1)$ et ainsi de toute loi normale $\mathcal{N}(m, \sigma^2)$.

Correction. — (i) La loi du couple (R, Θ) sur $\mathbb{R}_+ \times [0, 2\pi[$ s'exprime de la manière suivante

$$P_{(R,\Theta)}(\mathrm{d}r\,\mathrm{d}\theta) = r\,\mathrm{e}^{-r^2/2} \otimes \frac{\mathrm{d}\theta}{2\pi} = \mathrm{e}^{-r^2/2} \times r\,\mathrm{d}r\,\mathrm{d}\theta \times \frac{1}{2\pi}$$

On reconnaît l'élément de surface $r\,\mathrm{d}r\,\mathrm{d}\theta$ en coordonnées polaires dans le plan $\mathbb{R}^2$. Ainsi, en passant en coordonnées cartésiennes $x = r\cos\theta$, $y = r\sin\theta$, on a $r^2 = x^2 + y^2$ et l'image de la mesure précédente par cette transformation est la mesure

$$\mathrm{e}^{-(x^2+y^2)^2/2}\,\mathrm{d}x\,\mathrm{d}y \times \frac{1}{2\pi} = \mathrm{e}^{-x^2/2}\frac{\mathrm{d}x}{\sqrt{2\pi}} \otimes \mathrm{e}^{-y^2/2}\frac{\mathrm{d}y}{\sqrt{2\pi}}$$

qui est donc la loi du couple (X, Y) obtenu comme image de (R, Θ) par l'application faisant passer des coordonnées polaires au coordonnées cartésiennes. Par simple identification, nous constatons que X et Y sont toutes deux de loi $\mathcal{N}(0, 1)$ et que, leur loi conjointe étant leur loi produit, elles sont indépendantes. La réciproque est immédiate : on passe des coordonnées cartésiennes aux coordonnées polaires.

(ii) Étant données deux telles variables aléatoires U_1 et U_2, il est clair que R et Θ sont indépendantes et Θ de loi uniforme sur $[0, 2\pi[$ (la fermeture ou l'ouverture des bornes de l'intervalle ne change rien). La fonction de répartition de la loi de R se calcule facilement : pour $r \geqslant 0$, $F_R(r) = \mathbb{P}\{R \leqslant r\} = \mathbb{P}\{\sqrt{-2\ln(1-U_1)} \leqslant r\} = \mathbb{P}\{-2\ln(1-U_1) \leqslant r^2\} = \mathbb{P}\{\ln(1-U_1) \geqslant r^2/2\} = \mathbb{P}\{1-U_1 \geqslant \exp(-r^2/2\} = \mathbb{P}\{U_1 \leqslant 1-\exp(-r^2/2\} = 1-\exp(-r^2/2\}$ puisque U_1 est de loi uniforme sur $[0, 1]$. Cette fonction est régulière sur $\mathbb{R}_+$ ce qui montre que la loi de R admet une densité qu'on peut obtenir en dérivant sa fonction de répartition : $p_R(r) = r\exp(-r^2/2)$ pour $r \geqslant 0$. On a la méthode.

Voici une mise en œuvre en Scilab (facilement transposable dans un autre langage) :

```
// Lois normales, m\'ethode polaire (Box-Muller)

global lastnormaldeviate; lastnormaldeviate = %inf;

function x = normaldeviate()
    global lastnormaldeviate;
    if lastnormaldeviate == %inf then
        u = grand(2, 1, "def");
        x = sqrt(-2*log(1-u(1)))*cos(2*%pi*u(2));
        lastnormaldeviate = sqrt(-2*log(1-u(1)))*sin(2*%pi*u(2));
    else
        x = lastnormaldeviate;
        lastnormaldeviate = %inf;
    end
endfunction
```

Notons que puisque la méthode génère deux valeurs à chaque fois alors qu'*a priori* une seule valeur est nécessaire pour chaque appel, on conserve la seconde valeur générée pour un appel ultérieur plutôt que d'en générer deux nouvelles à cette occasion. Pour générer des nombres suivant une loi $\mathcal{N}(m, \sigma^2)$, il suffit alors de calculer $\sigma \times$ `normaldeviate()` $+ m$.

Notons que considérer $\ln(1-U_1)$ plutôt que $\ln U_1$ ne change rien puisque $1-U_1$ est aussi de loi uniforme sur $[0, 1]$ et indépendante de U_2, mais c'est académiquement plus satisfaisant. D'un point de vue numérique, il peut y avoir une subtilité. Ici, l'option `"def"` du générateur aléatoire `grand()` garantit que la valeur 1 ne sera jamais retournée. Il vaut donc mieux considérer $\ln(1-U_1)$.

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 [4].

Exercice 7 (*pratique*). — (i) Implanter 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) Implanter 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) Implanter 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.

6. **Les méthodes de rejet**

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

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

Exercice 8. — (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 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. (On peut aussi y parvenir directement…)

Théorème (méthode de rejet). — *Soient $(E, \mathcal{E}, \mu)$ un espace probabilisé et $F \in \mathcal{E}$ une partie mesurable de mesure strictement positive $\mu(F) > 0$. Soient $X_n : (\Omega, \mathcal{A}, \mathbb{P}) \to (E, \mathcal{E})$, $n \geqslant 1$, une suite de variables aléatoires indépendantes, identiquement distribuées de loi μ et $T = \inf\{n \geqslant 1 : X_n \in F\}$. Alors la variable aléatoire T est presque sûrement finie et*

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

est une variable aléatoire de loi

$$\mu(\,.\,|\,F) : \mathcal{E} \longrightarrow [0,1]$$
$$B \longmapsto \mu(B\,|\,F) = \frac{\mu(B \cap F)}{\mu(F)}$$

autrement dit la loi μ conditionnellement à F. De plus, si on définit par récurrence

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

les variables aléatoires $Z_n = X_{T_n}$, $n \geqslant 1$, sont indépendantes et identiquement distribuées de loi $\mu(\,.\,|\,F)$.

Démonstration. — Notons $p = \mu(F) \in\,]0,1]$. 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,

$$\begin{aligned}
\mathbb{P}\{Z \in B\} &= \sum_{n=1}^{\infty} \mathbb{P}\{T = n, X_n \in B\} = \sum_{n=1}^{\infty} \mathbb{P}\{X_1 \notin F, \dots, X_{n-1} \notin F, X_n \in B\} \\
&= \sum_{n=1}^{\infty} \mathbb{P}\{X_1 \notin F\} \times \cdots \times \mathbb{P}\{X_{n-1} \notin F\} \times \mathbb{P}\{X_n \in B\} \\
&= \sum_{n=1}^{\infty} \mu(E\backslash F) \times \cdots \times \mu(E\backslash F) \times \mu(B) \\
&= \sum_{n=1}^{\infty} \mu(E\backslash F) \times \cdots \times \mu(E\backslash F) \times \mu(F) \times \mu(B)/\mu(F) \\
&= \sum_{n=1}^{\infty} (1-p) \times \cdots \times (1-p) \times p \times \mu(B)/\mu(F) \\
&= \left(\sum_{n=1}^{\infty} (1-p)^{n-1} p \right) \times \mu(B)/\mu(F) \\
&= \mu(B)/\mu(F)
\end{aligned}$$

où on note que $\mathbb{P}\{T = \infty\} = 0$ et que les $(X_n)_{n\geqslant 1}$ sont indépendantes de loi μ. Ainsi, Z est bien de loi $\mu/\mu(F)$ 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 $\mu(B_1)/\mu(F) \times \cdots \times \mu(B_n)/\mu(F)$, 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 méthode de rejet est souvent utilisée dans les cadres suivants : $F \subset E$, et

- ou bien E et F sont deux ensembles finis non vides ;
- ou bien E et F sont deux parties mesurables de $\mathbb{R}^n$ de mesures de Lebesgue strictement positives.

Dans ces deux cas, si la mesure μ est la mesure de probabilité uniforme sur E, alors $\mu(\,.\,|\,F)$ est la mesure de probabilité uniforme sur F.

Théorème. — *Soient $(E, \mathcal{E}, \lambda)$ un espace mesuré σ-fini, μ une mesure de probabilité sur $(E, \mathcal{E})$ absolument continue par rapport à λ de fonction de densité $p : (E, \mathcal{E}) \to \mathbb{R}_+$ ($\mu(\mathrm{d}x) = p(x)\,\lambda(\mathrm{d}x)$).*

Posons $F = \{(x, y) \in E \times \mathbb{R} : 0 \leqslant y \leqslant p(x)\}$ *qui est de mesure* 1 *dans* $E \times \mathbb{R}$ *muni de la mesure produit* $\lambda \otimes \mathrm{d}y$ *et notons* ν_F *la restriction de cette mesure produit à* F.

Si $Z = (X, Y) : (\Omega, \mathcal{A}, \mathbb{P}) \to F$ *est de loi* ν_F, *alors* $X : (\Omega, \mathcal{A}, \mathbb{P}) \to \mathbb{R}^d$ *est de loi* μ.

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

$$\begin{aligned} 1 = \int_E p(x)\,\lambda(\mathrm{d}x) &= \int_E \left(\int_0^{p(x)} \mathrm{d}y \right) \lambda(\mathrm{d}x) \\ &= \int_{E\times\mathbb{R}} \mathbb{1}_{\{0\leqslant y\leqslant p(x)\}}\,\lambda(\mathrm{d}x)\otimes\mathrm{d}y = \int_{E\times\mathbb{R}} \mathbb{1}_F(x,y)\,\lambda(\mathrm{d}x)\otimes\mathrm{d}y = (\lambda\otimes\mathrm{d}y)(F) \end{aligned}$$

car, par le théorème de Fubini–Tonelli, l'intégrale itérée est une intégrale multiple (c'est pourquoi on a besoin d'une hypothèse de σ-finitude).

Soit $Z = (X, Y)$ de loi ν_F. Pour tout $B \in \mathcal{E}$, on a

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

d'où la conclusion. □

La combinaison des deux théorèmes précédents donne une méthode de simulation de variables aléatoires de lois complexes. Supposons qu'on sache tirer des points suivant la loi λ (qui est ici une mesure de probabilité) et que la densité p de la mesure μ par rapport à λ est bornée par une valeur M (un cas simple est lorsque λ est la mesure uniforme sur un intervalle borné de $\mathbb{R}$ ou un produit d'intervalles bornés de $\mathbb{R}^d$). Avec une suite $(X_n, Y_n)_{n\geqslant 1}$ de variables aléatoires indépendantes, toutes de loi $\lambda\otimes\mathcal{U}([0, M])$, on obtient par rejet la variable Z cherchée et donc sa première coordonnée X qui est de loi μ sur E.

Exercice 9. — La fonction B (Beta) d'Euler est définie par

$$B(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1}\,\mathrm{d}x, \qquad a > 0,\ b > 0.$$

Pour a et b deux réels strictement positifs, la loi Beta de paramètres a et b est la mesure de probabilité absolument continue sur $\mathbb{R}$ admettant pour fonction de densité :

$$x \in \mathbb{R} \longmapsto \frac{\mathbb{1}_{[0,1]}(x)}{B(a,b)}\,x^{a-1}(1-x)^{b-1}.$$

Implanter un générateur des lois Beta de paramètres a et b supérieurs ou égaux à 1 sans utiliser, ni les valeurs de $B(a, b)$ (`beta(`$\langle a\rangle$`, `$\langle b\rangle$`)`), ni la fonction de répartition `cdfbet` fournie par Scilab. Comparer pour plusieurs choix pertinents des paramètres les échantillons des lois correspondantes. (La simulation pour $0 < a < 1$ ou $0 < b < 1$ n'est pas demandée).

Références

[4] 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.

Université de Poitiers
Département de Mathématiques

Probabilités Générales
M1 MFA–MMAS, année 2013–2014

TRAVAUX PRATIQUES. — SIMULATIONS DE VARIABLES ALÉATOIRES, ÉLÉMENTS D'EXPLICATIONS

1. **Pour commencer**

EXERCICE 1. — Voici le code SCILAB.

```
// TP2, exercice 1

clear; mode(0);

function u = random()
    u = grand(1, 1, "def");
endfunction

// Question (i)

function x = PouF(p)
    if random() < 1-p then x = 0; else x = 1; end
endfunction

p = 0.25; N = 1000; pobs = 0;
for i = 1:N;
    pobs = pobs+PouF(p);
end;
pobs = pobs/N;

scf(1); clf();
bar([0; 1], [1-p, 1-pobs; p, pobs]);
title("Pile ou Face avec p = "+string(p));
legend(["distribution theorique"; "distribution observee"]);

// Question (ii)

// traduction litt\'erale

function i = discrete(ptheo)
    // local u i p
    u = random(); i = 0;
    for p = ptheo'; // l'argument doit \^etre un vecteur ligne
        i = i+1; u = u-p;
        if u <= 0 then break; end
    end
endfunction

// avec une boucle while
```

```
function i = discrete(ptheo)
    // local u i
    u = random(); i = 0;
    while u >= 0; // noter qu'on doit prendre une in\'egalit\'e large
        i = i+1; u = u-ptheo(i);
    end
endfunction

// autre version

function i = discrete(ptheo)
    // local u;
    u = random();
    for i = 1:size(ptheo,1)
        u = u-ptheo(i);
        if u < 0 then break; end
    end
endfunction

xtheo = [1:6]';
ptheo = ones(6,1)/6;
pobs = zeros(6,1);
for i = 1:N;
    outcome = discrete(ptheo);
    pobs(outcome) = pobs(outcome)+1;
end;
pobs = pobs/N;

scf(2); clf();
bar(xtheo, [ptheo, pobs]);
title("Lancer d''un de avec la commande discrete()");
legend(["distribution theorique"; "distribution observee"]);

n = 10; p = 0.5;
xtheo = [0:n]';
ptheo = binomial(p, n)';
pobs = zeros(n+1,1);
for i = 1:N;
    outcome = discrete(ptheo); // c'est dans 1,...,n+1
    pobs(outcome) = pobs(outcome)+1;
end;
pobs = pobs/N;

scf(3); clf();
bar(xtheo, [ptheo, pobs]);
title("Simulation d''une loi binomiale avec la commande discrete()");
legend(["distribution theorique"; "distribution observee"]);
```

2. Un aspect statistique

Exercice 2. — Pour les deux dernières simulations, on pourra rajouter le code qui suit.

```
// TP2, exercice 2

k = size(xtheo, 1);
cs = sum((pobs-ptheo).^2./ptheo)*N;
pvaleur = 1-cdfchi("PQ", cs, k-1);
```

3. Quelques programmes utiles

Dans un éventuel nouveau script :

```
clear; mode(0);
exec kolmogorov.sci;
exec goodness-of-fit.sci;

function u = random()
    u = grand(1, 1, "def");
endfunction
```

4. Utilisation de la fonction de répartition

Exercice 3. — (i) Soit $p \in]0,1[$. Soit $T : (\Omega, \mathcal{A}, \mathbb{P}) \to \mathbb{N}^*$ une variable aléatoire de loi géométrique de paramètre p. Nous savons que pour $t > 0$,

$$\mathbb{P}\{T > t\} = \mathbb{P}\{T > \lfloor t \rfloor\} = (1-p)^{\lfloor t \rfloor},$$

où $\lfloor x \rfloor$ est la partie entière inférieure de x. On en déduit que la fonction de répartition de la loi géométrique de paramètre p est

$$F_T(t) = \mathbb{P}\{T \leqslant t\} = 1 - \mathbb{P}\{T > t\} = \begin{cases} 0 & \text{si } t \leqslant 0, \\ (1-p)^{\lfloor t \rfloor} & \text{si } t > 0. \end{cases}$$

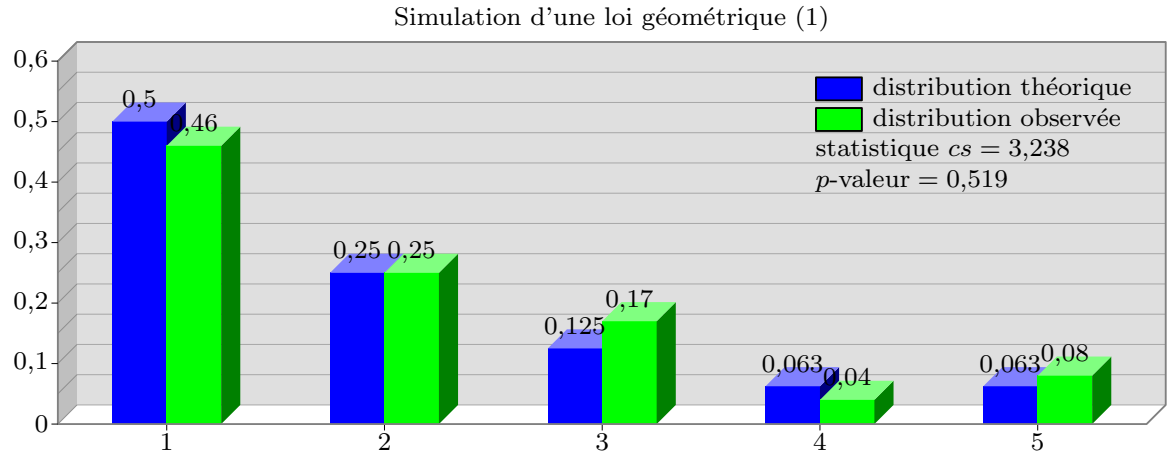
En traçant le graphe de $t \mapsto (1-p)^t$ et en la comparant avec la fonction de répartition précédente, on obtient que l'inverse généralisé de F_T est

$$F_T^{-1}(u) = \begin{cases} -\infty & \text{si } u = 0, \\ \left\lceil \dfrac{\ln(1-u)}{\ln(1-p)} \right\rceil & \text{si } 0 < u < 1, \\ +\infty & \text{si } u = 1. \end{cases}$$

En tenant compte de valeurs extravagantes pour le paramètre p ($p \leqslant 0$ ou $p \geqslant 1$), on obtient

```
// TP2, exercice 4

// Question (i) Loi g\'eom\'etrique G(p).
// La fonction de r\'epartition est F(x)=1-(1-p)^floor(x) pour x >= 0 qui
// s'inverse assez tranquillement.  (Remarque : grand(n,m,"geom",p))

function x = geometricdeviate(p)
    if p <= 0 then x = %inf;
    elseif p >= 1 then x = 1;
    else x = ceil(log(1-random())/log(1-p));
    end
endfunction
```

```
p = 0.25; N = 1000; M = 20;
x = zeros(N, 1);
for i = 1:N; x(i) = min(geometricdeviate(p), M); end

xtheo = [1:M]';
ptheo = zeros(M, 1);
ptheo(1) = p;
for i = 2:(M-1); ptheo(i) = ptheo(i-1)*(1-p); end
ptheo(M)=(1-p)^(M-1);

scf(0); clf();
dgoftest(x, xtheo, ptheo);
title("Simulation de la loi geometrique de parametre p = "+string(p));
```



On notera qu'on a ramené toutes les valeurs possibles ou observables à un seuil maximal M. Celui-ci a été calculé de sorte que `ptheo[i]` $\times N \geqslant 5$, de sorte que nous puissions espérer au moins 5 valeurs observées pour chaque classe `xtheo[i]`, $i = 1, \ldots, M$, ce qui est un minimum pour appliquer le test du χ^2.

(ii) Pour la loi exponentielle de paramètre $\lambda > 0$, on a

$$F_T(t) = \begin{cases} 0 & \text{si } t \leqslant 0, \\ 1 - \mathrm{e}^{-\lambda t} & \text{si } t > 0. \end{cases}$$

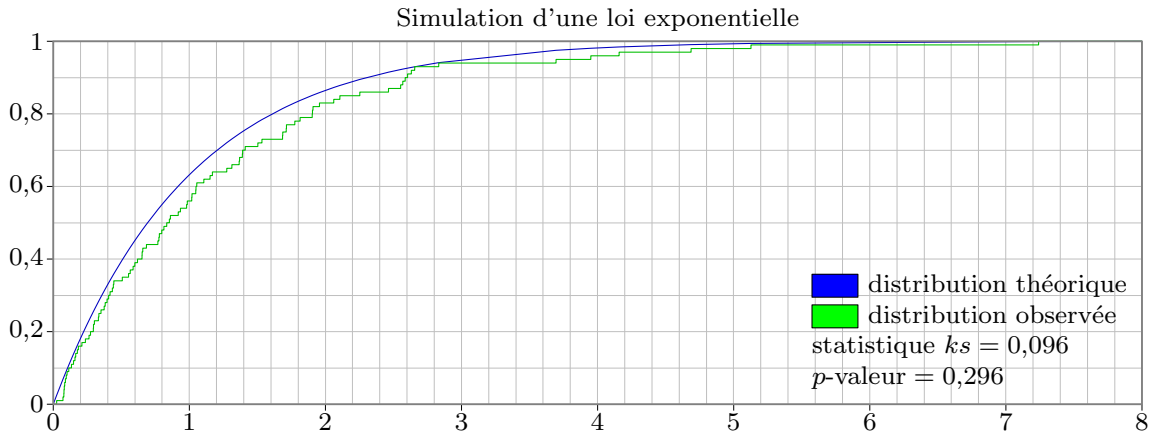
Son inverse généralisé est donné par

$$F_T^{-1}(u) = \begin{cases} -\infty & \text{si } u = 0, \\ -\ln(1-u)/\lambda & \text{si } 0 < u < 1, \\ +\infty & \text{si } u = 1. \end{cases}$$

On programme alors

```
// Question (ii) Loi E(lambda)
// (Remarque : grand(n,m,"exp",lambda))

function x = exponentialdeviate(lambda)
    x = -log(1-random())/lambda;
endfunction

lambda = 1;
for i = 1:N; x(i) = exponentialdeviate(lambda); end

deff("p = F(x)", "p = 1-exp(-lambda*x)");
```

```
scf(1); clf();
cgoftest(x, F);
title("Simulation de la loi exponentielle de parametre lambda = "...
    +string(lambda));
```



(iii) La densité d'une loi de Cauchy de paramètre $a > 0$ est donnée dans le document. Sa fonction de répartition est alors

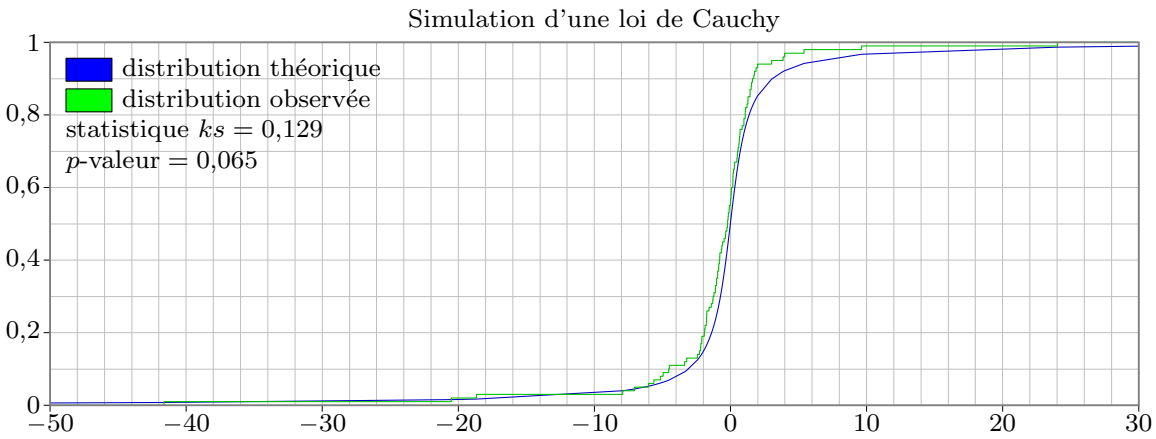
$$\begin{aligned} F(x) &= \int_{-\infty}^{x} \frac{a/\pi}{a^2+y^2}\,\mathrm{d}y = \frac{1}{\pi}\int_{-\infty}^{x} \frac{1}{1+(y/a)^2}\,\mathrm{d}(y/a) \\ &= \frac{1}{\pi}\big(\arctan(x/a) - \arctan(-\infty)\big) = \frac{1}{\pi}\arctan(x/a) + \frac{1}{2}\,. \end{aligned}$$

Son inverse est tout simplement

$$F^{-1}(u) = \begin{cases} -\infty & \text{si } u = 0, \\ a \times \tan\big(\pi(p-1/2)\big) & \text{si } 0 < u < 1, \\ +\infty & \text{si } u = 1. \end{cases}$$

On programme alors

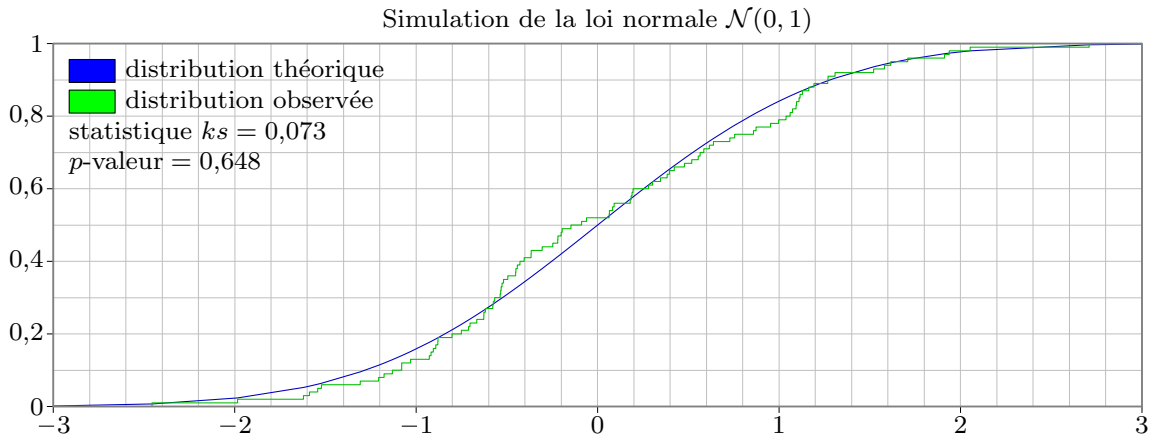
```
// Question (iii) Loi C(a)

function x = cauchydeviate(a)
    x = a*tan(%pi*(random()-0.5));
endfunction

a = 1;
for i = 1:N; x(i) = cauchydeviate(a); end

deff("p = F(x)", "p = 1/%pi*atan(x/a)+0.5"); // noter le atan

scf(2); clf();
cgoftest(x, F);
title("Simulation de la loi de Cauchy de parametre a = "+string(a));
```



La figure ci-dessus étant régénérée à chaque reformatage de ce document, nous ne pouvons pas la commenter précisément. Ce qu'il faut en premier lieu remarquer est l'étendue des valeurs observées. Il n'est pas rare que sur un petit (100 ou 1000) nombre de tirage on obtienne des valeurs supérieures à 100 par exemple en valeur absolue. Ceci provient du fait que la décroissance aux infinis de la fonction de densité est assez lente, et, par conséquent, qu'il est très probable d'observer de grandes valeurs. Les lois de Cauchy sont assez (contre-)exemplaires en théorie des probabilités : pas de théorème central limite (sinon spécifique), discontinuité des trajectoires du processus de Cauchy — le processus à accroissements indépendants dont le semi-groupe de convolution est le semi-groupe à 1 paramètre $(\mathcal{C}auchy(t))_{t\geqslant 0}$ —, etc.

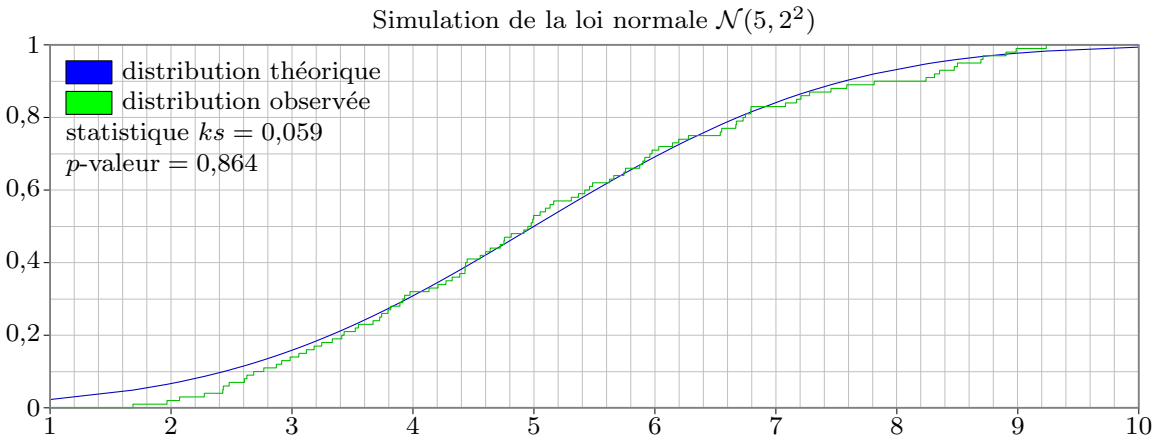
(iv) Pour la loi Normale (sans grand intérêt mis à part d'utiliser la fonction `cdfnor`) :

```
// Question (iv) Loi N(0, 1)
// (Remarque : grand(n,m,"nor",mu,sigma))

function  x = normaldeviate()
    u = random();
    x = cdfnor("X", 0, 1, u, 1-u);
endfunction

for i = 1:N; x(i) = normaldeviate(); end

deff("p = F(x)", "p = cdfnor(''PQ'', x, 0, 1)");
// noter le redoublement des guillemets

scf(3); clf();
cgoftest(x, F);
title("Simulation de la loi normale N(0,1)");
```



(v) Pour une loi normale de moyenne m et d'écart-type σ :

```
// Question (v) Loi N(m, sigma^2)
// grand(n,m,"nor",mu,sigma)

function  x = gaussiandeviate(m, sigma)
    u = random();
    x = sigma*cdfnor("X", 0, 1, u, 1-u)+m;
    // x = cdfnor("X", m, sigma, u, 1-u);
endfunction

m = 5; sigma = 2;
for i = 1:N; x(i) = gaussiandeviate(m, sigma); end

deff("p = F(x)", "p = cdfnor(''PQ'', x, m, sigma)");
// noter le redoublement des guillemets

scf(4); clf();
cgoftest(x, F);
title("Simulation de la loi normale de moyenne "+string(m)...
    +" et de d''ecart-type "+string(sigma));
```

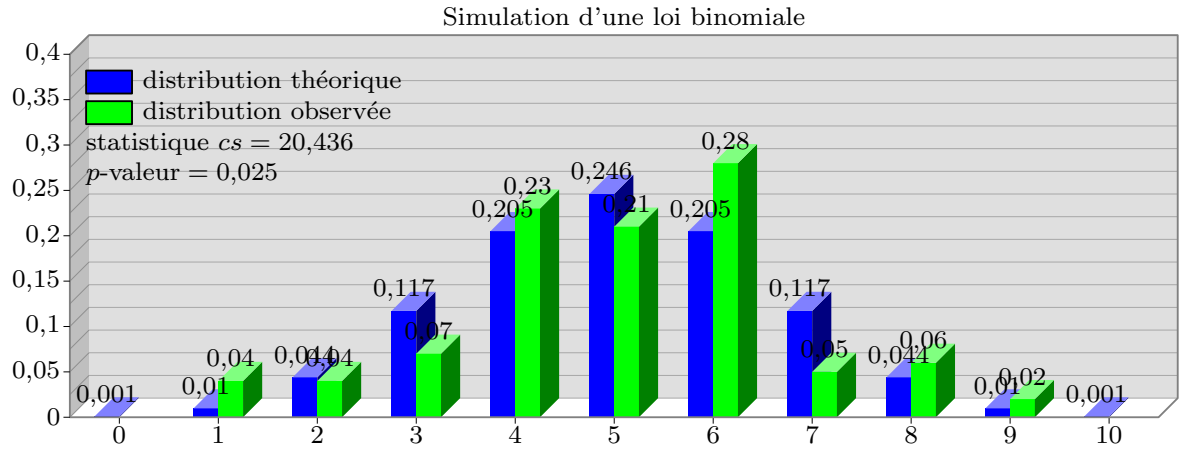


5. **Utilisation de propriétés indirectes (2nd semestre)**

Exercice 4. — (i) La loi binomiale de paramètres $n \in \mathbb{N}$ et $p \in [0,1]$ est par exemple la loi du nombre de succès dans un schéma de Bernoulli de mêmes paramètres, ou encore la somme de n variables aléatoires indépendantes et identiquement distribuées de loi de Bernoulli de paramètre p. Compte tenu des propriétés pratiques du générateur aléatoire, on peut programmer :

```
clear; mode(0);
exec kolmogorov.sci;
exec goodness-of-fit.sci;

function u = random()
    u = grand(1, 1, "def");
endfunction

// TP2, exercice 5

// Question (i)  Lois binomiales par sch\'ema de Bernoulli fini
```

```
function x = binomialdeviate(n, p)
    x = 0;
    for i = 1:n;
        if random() > 1-p then x = x+1; end;
    end;
endfunction

n = 10; p = 0.5; N = 1000;
x = zeros(N,1);
for i = 1:N; x(i) = binomialdeviate(n, p); end

xtheo = [0:n]';
// ptheo = zeros(n+1, 1);
// ptheo(1) = (1-p)^n;
// for i = 1:n; ptheo(i+1) = ptheo(i)*(p/(1-p))*((n-i+1)/i); end
ptheo = binomial(p, n)';

scf(0); clf();
dgoftest(x, xtheo, ptheo);
title("Simulation d''une loi binomiale");
```

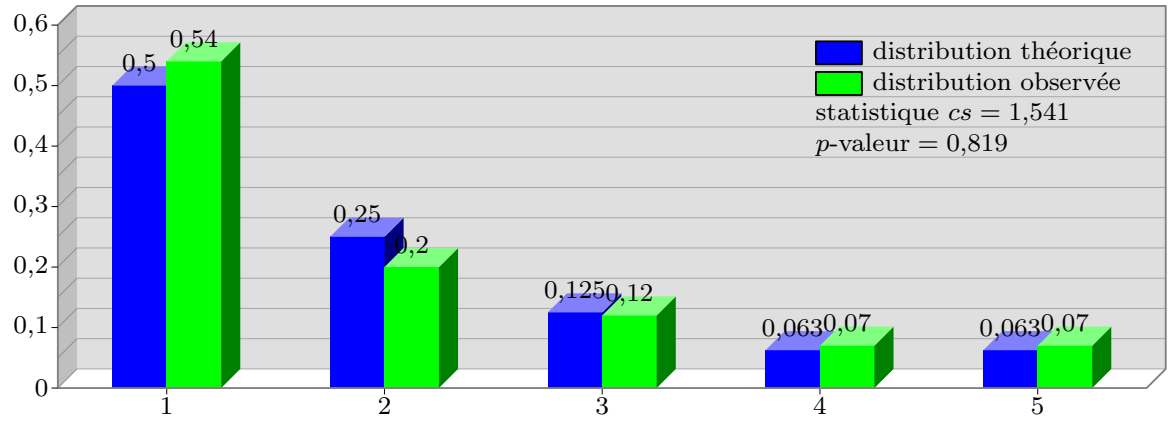


(ii) La loi géométrique de paramètre $p \in [0,1]$ est par exemple la loi du premier succès dans un schéma de Bernoulli infini. Compte tenu des propriétés pratiques du générateur aléatoire, on peut programmer :

```
// Question (ii) Lois g\'eom\'etriques par sch\'ema de Bernoulli infini

function x = geometricdeviate(p)
    if p <= 0 then x = %inf;
    elseif p >= 1 then x = 1;
    else x = 1;
        while random() < 1-p; x = x+1; end
    end
endfunction

p = 0.5;
M = ceil(log(5/N)/log(1-p)); // 5 <= N*(1-p)^(M-1)
for i = 1:N; x(i) = min(geometricdeviate(p), M); end

xtheo = [1:M]';
// ptheo = zeros(M, 1);
// ptheo(1) = p;
// for i = 2:(M-1); ptheo(i) = ptheo(i-1)*(1-p); end
```

```
// ptheo(M)=(1-p)^(M-1);
ptheo =[p*(1-p)^[0:M-2], (1-p)^(M-1)]';

scf(1); clf();
dgoftest(x, [1:M]', ptheo);
title("Simulation d''une loi geometrique");
```

Simulation d'une loi géométrique (2)

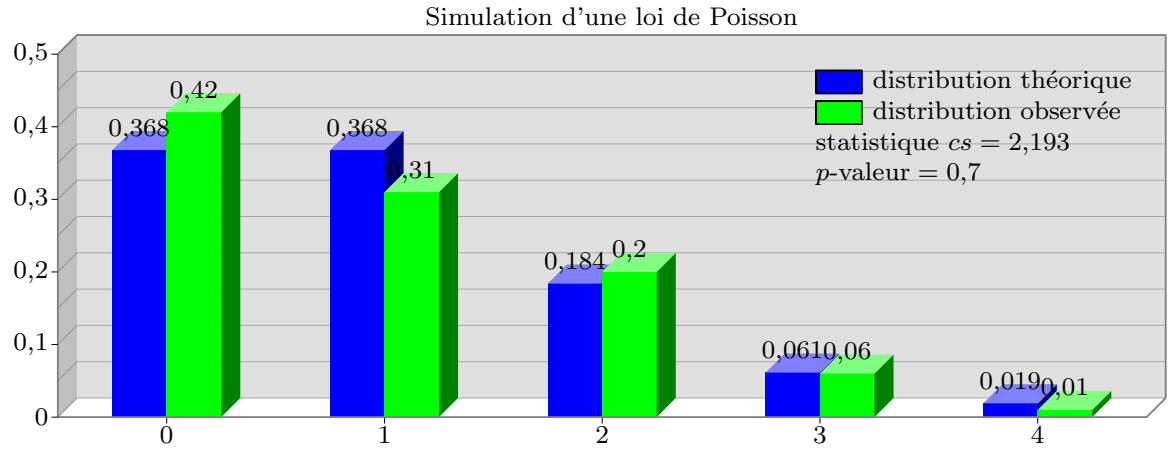


On notera qu'on aura recopié la partie d'analyse du cas précédent qui correspond. La méthode d'inversion est préférable car elle nécessite un et un seul appel au générateur aléatoire.

(iii) Le cas poissonnien est un peu plus difficile à expliquer. On peut faire le calcul...

```
// Question (iii) Lois Poisson par files d'attente

function x = poissondeviate(lambda)
    if lambda <= 0 then x = 0;
    else x = -1;
        while lambda > 0;
            lambda = lambda+log(1-random());
            x = x+1;
        end
    end
endfunction

// en passant \'a l'exponentielle on \'economise des calculs

function x = poissondeviate(lambda)
    if lambda <= 0 then x = 0;
    else x = -1; t = exp(lambda);
        while t > 1;
            t = t*random();
            x = x+1;
        end
    end
endfunction

lambda = 1;
if %f then // solution de facilit\'e
    M = 10; // difficile \'a \'evaluer
    ptheo = zeros(M+1, 1); ptheo(1) = exp(-lambda); ptheo(M+1) = 1-ptheo(1);
    for i = 1:(M-1);
        ptheo(i+1) = ptheo(i)*lambda/i;
        ptheo(M+1) = ptheo(M+1)-ptheo(i+1);
```

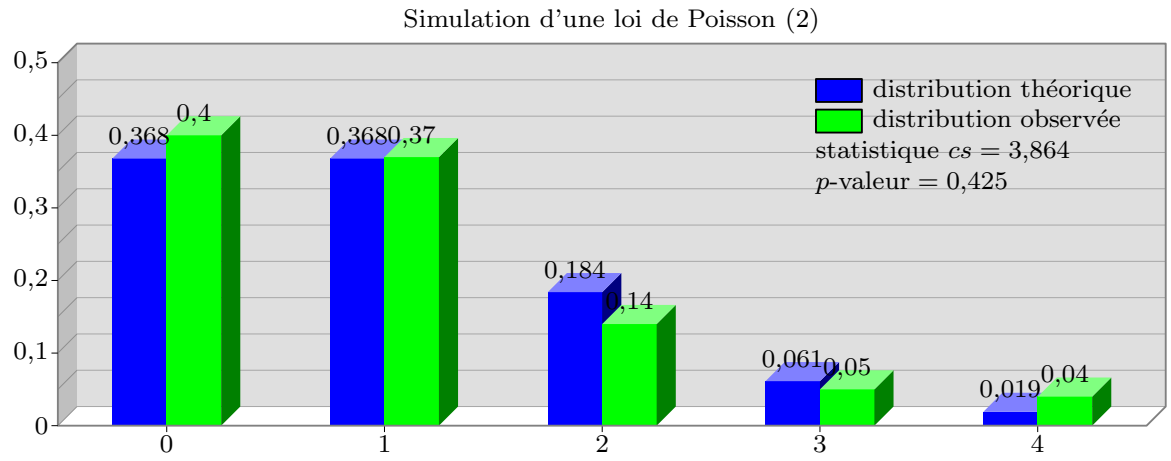
```
    end
end

M = 0; p = exp(-lambda); ptheo = p;
while p*N >= 5 | M <= lambda;
    M = M+1;
    p = p*lambda/M;
    ptheo = [ptheo; p];
end
ptheo(M+1) = 1; for i = 1:M; ptheo(M+1) = ptheo(M+1)-ptheo(i); end
xtheo = [0:M]';

for i = 1:N; x(i) = min(poissondeviate(lambda), M); end

scf(2); clf();
dgoftest(x, xtheo, ptheo);
title("Simulation d''une loi de Poisson (1)");
```



La détermination de l'entier M servant à borner les valeurs théoriques et observées aura été faite lors du calcul des probabilités théoriques. Il dépend de λ (ici $\lambda = 1$) et de la taille de l'échantillon N.

Une autre méthode est peut-être préférable. Le code suivant consiste finalement à inverser la fonction de répartition. Chaque requête ne nécessite qu'un appel au générateur aléatoire. De plus, il n'y a qu'un unique appel à une fonction complexe, l'exponentielle, contre un nombre variable d'appels au logarithme dans le cas précédent.

```
function x = newpoissondeviate(lambda)
    if lambda <= 0 then x = 0;
    else // local p, t;
        t = exp(-lambda); p = random()-t; x = 0;
        while p > 0;
            x = x+1;
            t = t*lambda/x;
            p = p-t;
        end
    end
endfunction

for i = 1:N; x(i) = min(newpoissondeviate(lambda), M); end

scf(3); clf();
dgoftest(x, [0:M]', ptheo);
title("Simulation d''une loi de Poisson (2)");
```

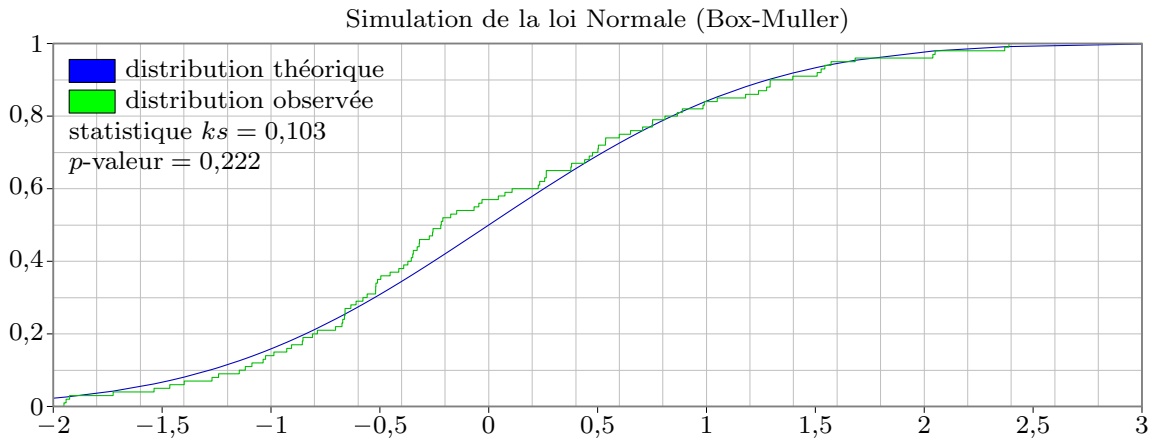


Exercice 5. — (i) Pour la méthode de Box–Müller, lorsque U_1 et U_2 sont deux variables aléatoires indépendantes de loi uniforme sur $[0,1]$,

$$\begin{cases} X_1 = \sqrt{-2\ln(1-U_1)} \times \cos(2\pi U_2) \\ X_2 = \sqrt{-2\ln(1-U_1)} \times \sin(2\pi U_2) \end{cases}$$

définissent deux variables aléatoires indépendantes de loi $\mathcal{N}(0,1)$. On peut programmer :

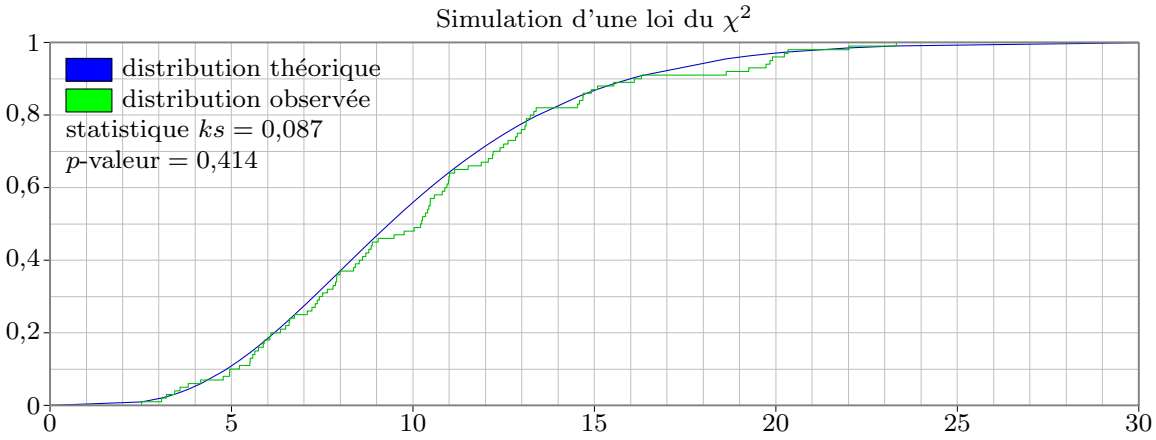
```
// TP2, exercice 7
// Lois normales, m\'ethode polaire (Box-Muller)

global lastnormaldeviate; lastnormaldeviate = %inf;

function x = normaldeviate()
    global lastnormaldeviate;
    if lastnormaldeviate == %inf then
        u = grand(2, 1, "def");
        x = sqrt(-2*log(1-u(1)))*cos(2*%pi*u(2));
        lastnormaldeviate = sqrt(-2*log(1-u(1)))*sin(2*%pi*u(2));
    else
        x = lastnormaldeviate;
        lastnormaldeviate = %inf;
    end
endfunction

x = []; for i = 1:N; x = [x; normaldeviate()]; end
deff("p = F(x)", "p = cdfnor(''PQ'', x, 0, 1)");

scf(0); clf();
cgoftest(x, F);
title("Simulation de la loi normale N(0,1) (Box-Muller)");
```



(ii) Pour une loi du χ^2 de paramètre entier n, nous utilisons le fait que si $X_1, \dots, X_n$ sont des variables aléatoires indépendantes de loi $\mathcal{N}(0,1)$, alors $X = X_1^2 + \cdots + X_n^2$ est une variable aléatoire positive de loi la loi du χ^2 à n degrés de liberté. On programme immédiatement :

```
// Lois du khi-2 de param\‘etres entiers

function x = chisquaredeviate(n)
    x = 0;
    for i = 1:n; x = x+normaldeviate()**2; end
endfunction

n = 10;
x = []; for i = 1:N; x = [x; chisquaredeviate(n)]; end
deff("p = F(x)", "p = cdfchi(’’PQ’’, x, 10)");

scf(1); clf();
cgoftest(x, F);
title("Simulation d’’une loi du khi-deux");
```

Notons que la fonction `chisquaredeviate` accepte n'importe quel argument mais fonctionnera comme si celui-ci était entier.



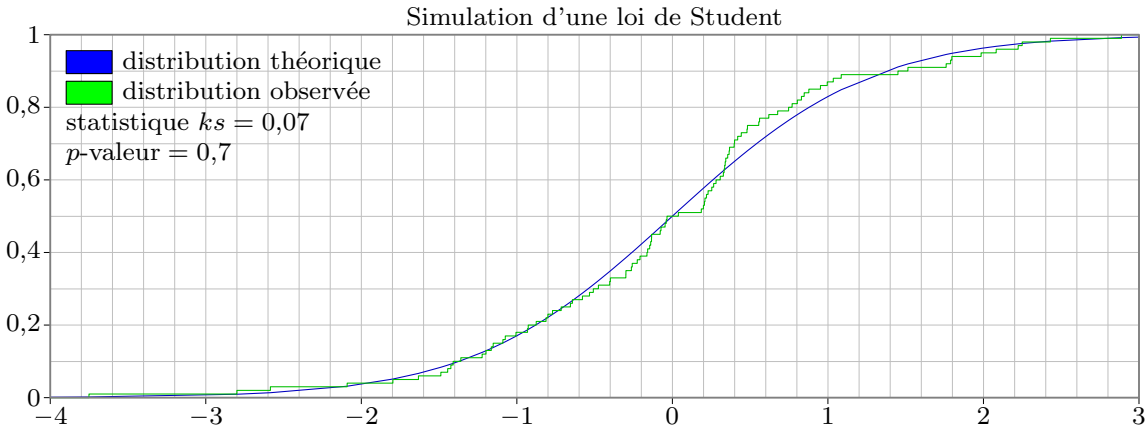
Pour $n = 2$, c'est sans intérêt ou presque puisqu'on a à faire alors à la loi exponentielle de paramètre $1/2$. Il est préférable d'inverser la fonction de répartition puisque ça ne coûte qu'un appel alors que ce qui précède en nécessite 2.

Pour un paramètre ν non entier, les lois $(\chi^2(\nu))_{\nu \geqslant 0}$ formant un semi-groupe de convolution, le problème se réduit alors à $\nu \in]0,1[$, mais nous ne connaissons pas de méthode exacte (autre que l'inversion de la fonction de répartition) pour ce faire.

(iii) Pour une loi de Student de paramètre entier (ou quelconque), nous utilisons le fait que si Z et X sont deux variables aléatoires réelles indépendantes, la première de loi $\mathcal{N}(0,1$, la

seconde de loi $\chi^2(n)$, alors $T = Z/\sqrt{X/n}$ a pour loi la loi de Student $\mathcal{T}(n)$ à n degrés de liberté.

```
// Lois de Student de param\‘etres entiers

function x = studentdeviate(n)
    x = normaldeviate()/sqrt(chisquaredeviate(n)/n);
endfunction

n = 10;
x = []; for i = 1:N; x = [x; studentdeviate(n)]; end
deff("p = F(x)", "p = cdft(’’PQ’’, x, n)");

scf(2); clf();
cgoftest(x, F);
title("Simulation d’’une loi de Student");
```



Observons que si on sait générer des nombres au hasard selon n’importe quelle loi du χ^2, on sait alors le faire pour les lois de Student.

Exercice 6. — Ceci n’est pas vraiment un exercice puisque pour être pertinentes, la majorité des réponses doivent être fausses (ce qui est difficile à faire honnêtement).

(i) Définissons des variables de lois supposées uniformes.

a) On pose

$$X_a = \exp(2\mathrm{i}\pi U_1) : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{S}^1.$$

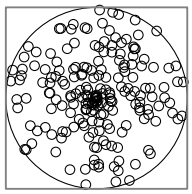
Il est évident et vrai que la loi de X_a est la loi uniforme sur $\mathbb{S}^1$. Au hasard, avec $V_i = 2U_i - 1$ de loi uniforme sur $[-1, 1]$,

$$X'_a = \frac{(V_1, V_2)}{\sqrt{V_1^2 + V_2^2}} : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{S}^1$$

aurait pu être un candidat mais ce n’est pas de loi uniforme.

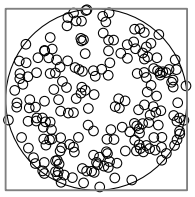
b) Au hasard,

$$X'_b = U_1 \times \exp(2\mathrm{i}\pi U_2) : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{D}^2,$$



mais ce n'est pas de loi uniforme car $\mathbb{P}\{X_b \in \mathbb{D}^2_r\} = r$ n'est pas proportionnel à la surface πr^2 du disque considéré. On constate qu'une bonne réponse est

$$X_b = \sqrt{U_1} \times \exp(2\mathrm{i}\pi U_2) : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{D}^2.$$



On vérifie que si $B \subset \mathbb{D}^2$ est un secteur de rayon $0 \leqslant r \leqslant 1$ et d'angle $0 \leqslant \theta \leqslant 2\pi$,

$$\mathbb{P}\{X_b \in B\} = \mathbb{P}\{U_1 \leqslant r^2, U_2 \leqslant \theta/2\pi\} = \mathbb{P}\{U_1 \leqslant r^2\} \times \mathbb{P}\{U_2 \leqslant \theta/2\pi\} = r^2 \times \theta/2\pi$$

qui est bien l'aire relative du secteur dans le disque.

c) Les réponses vont être fausses : par exemple

$$X_c = (U_1)^{1/3} \times \begin{pmatrix} \cos(2\pi U_2)\cos(2\pi U_3) \\ \sin(2\pi U_2)\cos(2\pi U_3) \\ \sin(2\pi U_3) \end{pmatrix} : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{D}^3$$

n'est pas de loi uniforme dans la boule unité.

d) De même,

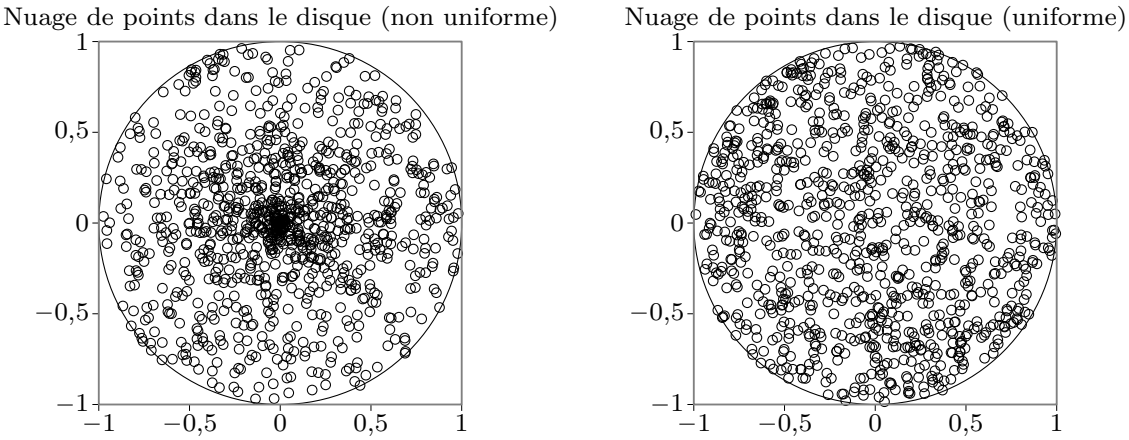
$$X_d = \frac{X_c}{\|X_c\|} = \begin{pmatrix} \cos(2\pi U_2)\cos(2\pi U_3) \\ \sin(2\pi U_2)\cos(2\pi U_3) \\ \sin(2\pi U_3) \end{pmatrix} : (\Omega, \mathcal{A}, \mathbb{P}) \longrightarrow \mathbb{S}^2$$

n'est pas de loi uniforme sur la sphère unité.

(ii) Programmons un peu dans le cas du disque en retenant une méthode qui marche.

```
clear; mode(0);

function u = random()
    u = grand(1, 1, "def");
endfunction

// Recherche de la loi uniforme dans le disque unit\'e

// disque en dimension 2
N = 1000;
// une bonne solution !
r = sqrt(grand(N, 1, "def"));
// une mauvaise
// r = grand(N, 1, "def");
theta = 2*%pi*grand(N, 1, "def");
x = r.*cos(theta); y = r.*sin(theta);
```

```
scf(0); clf();
a = gca(); // axes courants
a.isoview = "on"; // on veut un cercle et pas une ellipse
plot2d(y, x, style = -2);
t = linspace(0, 2*%pi, 50); plot2d(sin(t), cos(t));
xtitle("Nuage de point dans le disque");
```

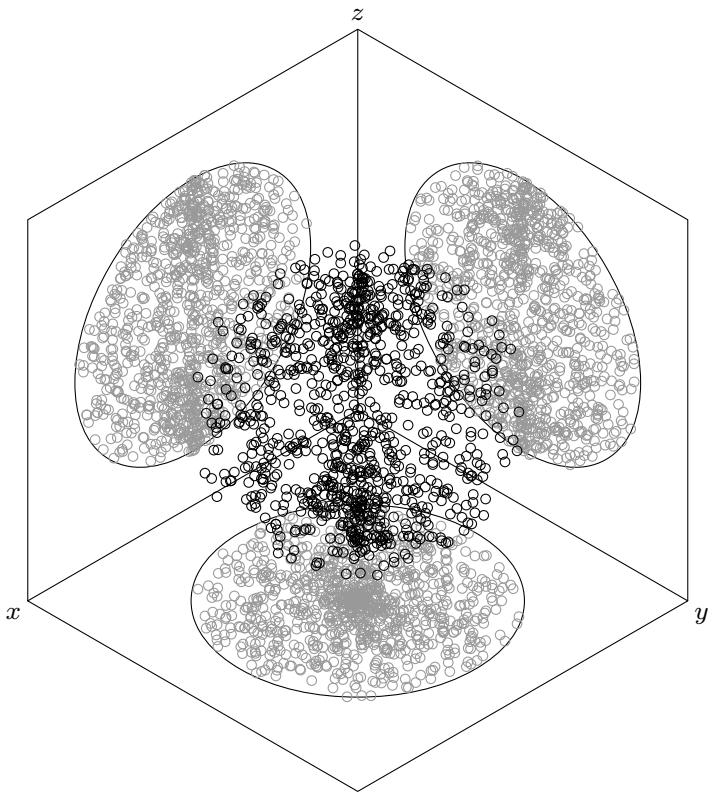


Pour la dimension 3, on sait que le rayon au cube est de loi uniforme sur $[0, 1]$, on a donc quelque chose comme $R = U^{1/3}$, que l'angle Θ (longitude) est de loi uniforme sur $[0, 2\pi[$, mais la détermination de la loi de l'angle Φ (lattitude) qui est à valeurs dans $[-\pi/2, \pi/2]$ nécessiterait un peu de calcul... prenons pour loi de Φ la loi uniforme, générons ces trois variables de manière indépendante, le résultat ne sera pas celui espéré.

```
// Recherche de la loi uniforme dans la boule unit\'e,
// une mauvaise solution

r = grand(N, 1, "def").^(1/3);
theta = 2*%pi*grand(N, 1, "def");
phi = %pi*(grand(N, 1, "def")-0.5);
x = r.*cos(theta).*cos(phi);
y = r.*sin(theta).*cos(phi);
z = r.*sin(phi);

// constater en particulier l'accumulation aux p\^oles

scf(1); clf();

t = linspace(0, 2*%pi, 50);

subplot(2, 2, 1);
a = gca(); a.isoview = "on";
xtitle("projection x-y");
plot2d(sin(t), cos(t)); plot2d(x, y, style = -2);

subplot(2, 2, 2);
a = gca(); a.isoview = "on";
xtitle("projection y-z");
plot2d(sin(t), cos(t)); plot2d(y, z, style = -2);

subplot(2, 2, 3);
a = gca(); a.isoview = "on";
xtitle("projection x-z");
plot2d(sin(t), cos(t)); plot2d(x, z, style = -2);
```

```
subplot(2, 2, 4);
//a = gca(); a.isoview = "on";
xtitle("Nuage de points dans la boule");
param3d1(x, y, list(z, -1), flag = [3,3], ebox=[-1,1,-1,1,-1,1]);
```

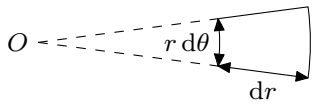
Ce qui donne un certain graphique plus ou moins lisible. Sur le graphique suivant, on constate que la densité de points varie selon des cônes d'axe $z'Oz$. La loi retenue pour la variable aléatoire Φ est peut-être à mettre en cause (nous reviendrons plus tard sur cette question).



Nuage de points dans la boule (non uniforme)

(iii) Soient R et Θ les coordonnées polaires d'un point aléatoire uniformément réparti sur le disque. De $\mathbb{P}\{M \in \mathbb{D}_r^2\} = \mathbb{P}\{R \leqslant r\} = r^2$, on obtient la loi de R (R^2 est de loi uniforme sur $[0,1]$). Si B est le secteur défini en b) avec $r = 1$, $\mathbb{P}\{M \in B\} = \mathbb{P}\{\Theta \leqslant \theta\} = \theta/2\pi$, donc Θ est de loi uniforme sur $[0, 2\pi]$. Pour l'indépendance de R et Θ, il suffit de la vérifier sur des secteurs de couronnes. Une autre façon de le voir est d'écrire l'élément de surface de surface normalisé du disque en coordonnées polaires :

$$\frac{r\,\mathrm{d}r\,\mathrm{d}\theta}{\pi} = 2r\,\mathrm{d}r \otimes \frac{\mathrm{d}\theta}{2\pi}\,.$$



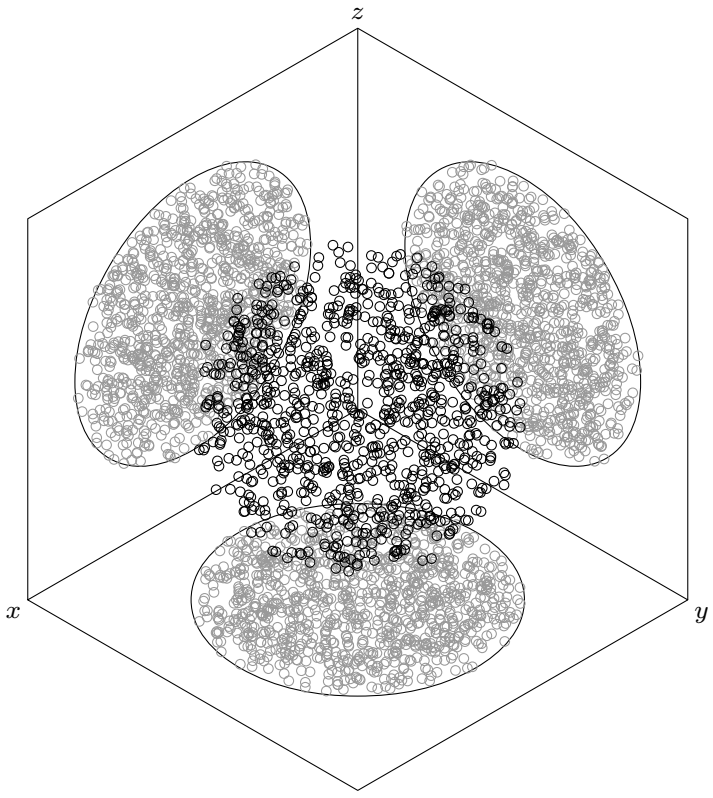
Pour que le couple (R, Θ) soit de loi uniforme sur le disque, il faut et il suffit que R et Θ soient indépendantes, que la loi de R vérifie $P_R(\mathrm{d}r) = 2r\,\mathrm{d}r$, $r \in [0,1]$, et que la loi de Θ soit la loi uniforme sur $[0, 2\pi[$.

(iv) La variable X_b définie précédemment est bien de loi uniforme sur le disque. On a $X_a'' = X_b/\|X_b\| = \exp(2\mathrm{i}\pi U_2)$ est de loi uniforme sur le cercle.

(v) Le cas de la boule est plus difficile. On génére X_c par la méthode de rejet : $M_k = (2U_{3k+1} - 1, 2U_{3k+1} - 2, 2U_{3k+3} - 1)$, $k \geqslant 0$, forme une suite de variables aléatoires indépendantes uniformément réparties sur le cube $[-1,1]^3$. On pose $T = \inf\{k \geqslant 0 : \|M_k|\} \leqslant 1\}$ et $X_c = M_T$. Le théorème donné dans l'énoncé affirme que T est fini presque sûrement et que X_T est de loi uniforme sur la boule. En posant $X_d = X_c/\|X_d\|$, on obtient une variable uniformément répartie sur la sphère.

```
// Recherche de la loi uniforme dans la boule unit\'e,
// une bonne solution : m\'ethode de rejet

x = ones(N, 1); y = ones(N, 1); z = ones(N, 1);
for i = 1:N;
    while x(i)^2+y(i)^2+z(i)^2 > 1;
        x(i) = 2*random()-1; y(i) = 2*random()-1; z(i) = 2*random()-1;
    end
end

scf(2); clf();

subplot(2, 2, 1);
a = gca(); a.isoview = "on";
xtitle("projection x-y");
plot2d(sin(t), cos(t)); plot2d(x, y, style = -2);

subplot(2, 2, 2);
a = gca(); a.isoview = "on";
xtitle("projection y-z");
plot2d(sin(t), cos(t)); plot2d(y, z, style = -2);

subplot(2, 2, 3);
a = gca(); a.isoview = "on";
xtitle("projection x-z");
plot2d(sin(t), cos(t)); plot2d(x, z, style = -2);

subplot(2, 2, 4);
// a = gca(); a.isoview = "on";
xtitle("Nuage de points dans la boule");
param3d1(x, y, list(z, -1), flag = [3,3], ebox=[-1,1,-1,1,-1,1]);
```

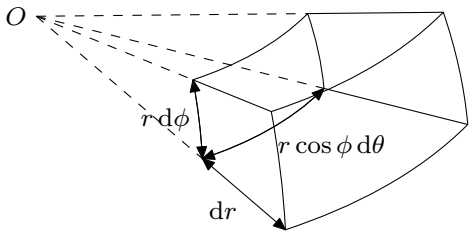
Ce qui donne un certain graphique plus ou moins lisible. Nous obtenons quant à nous :



Nuage de points dans la boule (méthode de rejet)

Remarque. — Tout praticien honnête du calcul intégral en coordonnées sphériques voit ce qu'il faut faire : avec notre choix et nos notations pour les coordonnées sphériques, l'élément de volume euclidien s'écrit

$$r^2 \cos\phi \, \mathrm{d}r \, \mathrm{d}\theta \, \mathrm{d}\phi.$$



En normalisant par le volume $4\pi/3$ de la boule unité et en arrangeant un peu les termes, ceci donne

$$3r^2 \, \mathrm{d}r \otimes \frac{\mathrm{d}\theta}{2\pi} \otimes \frac{\cos\phi \, \mathrm{d}\phi}{2} \, .$$

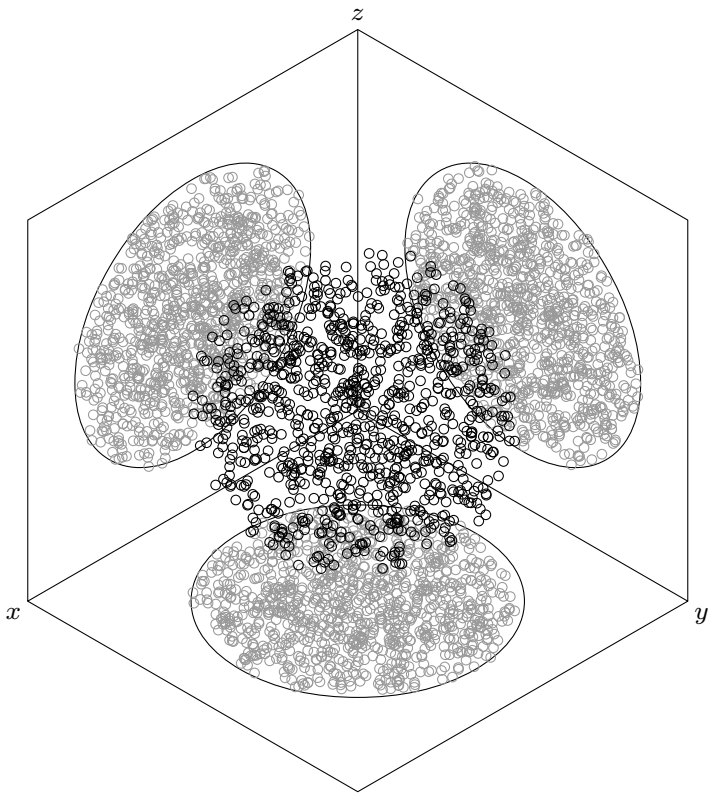
Ainsi, le triplet (R, Θ, Φ) est de loi uniforme sur la boule si et seulement si R, Θ et Φ sont indépendantes, de lois données par

$$\begin{array}{ll} P_R(\mathrm{d}r) = 3r^2 \, \mathrm{d}r, & r \in [0,1], \\ P_\Theta(\mathrm{d}\theta) = \mathrm{d}\theta/2\pi, & \theta \in [0, 2\pi[, \\ P_\Phi(\mathrm{d}\phi) = \cos\phi \, \mathrm{d}\phi/2, & \phi \in \,]-\pi/2, \pi/2[, \end{array}$$

ce qu'on peut réaliser de la manière suivante

$$R = (U_1)^{1/3}, \qquad \Theta = 2\pi U_2, \qquad \Phi = \mathrm{Arc}\sin(2U_3 - 1).$$

En effet, pour $\phi \in \,]-\pi/2, \pi/2[$, $F_\Phi(\phi) = \mathbb{P}\{\Phi \leqslant \phi\} = \int_{-\pi/2}^{\phi} \cos\psi \, \mathrm{d}\psi/2 = (\sin\phi + 1)/2$. La variable Φ est alors définie par inversion de la fonction de répartition en prenant garde au domaine de définition de l'Arc sinus.



Nuage de points dans la boule (uniforme et sans rejet)

Avec Scilab, ces points auraient pu être engendrés ainsi :

```
r = grand(N, 1, "def").^(1/3);
theta = 2*%pi*grand(N, 1, "def");
s = 2*grand(N, 1, "def")-1;
c = sqrt(1-s.^2);
x = r.*cos(theta).*c;
y = r.*sin(theta).*c;
z = r.*s;
```

en notant qu'on sait calculer le sinus d'un Arc sinus ainsi que son cosinus en faisant néanmoins attention au domaine de définition de l'Arc sinus (on ne le répète jamais assez).

Exercice 7. — Pour les lois Beta, on ne retient qu'une fonction de densité non normalisée pour se dispenser de trop de calculs :

$$x \in [0,1] \longmapsto x^{a-1}(1-x)^{b-1}.$$

Cette fonction est positive et bornée pour a et $b \geqslant 1$. Sa fonction dérivée est

$$x \in [0,1] \longmapsto \big(a-1-(a+b-2)x\big) \times x^{a-2}(1-x)^{b-2}$$

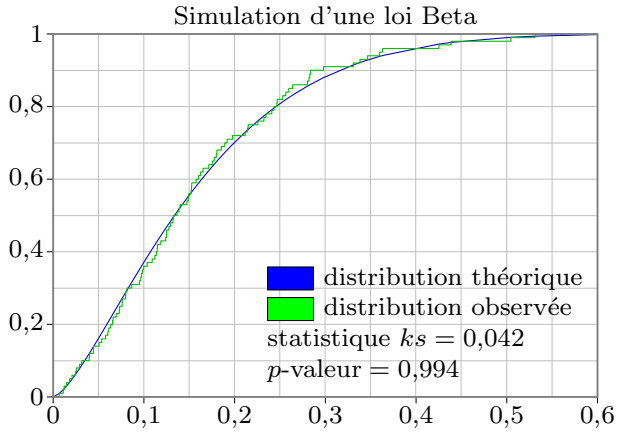
et elle s'annule en $x = (a-1)/(a+b-2) \in [0,1]$, point où le maximum (*upper-bound*)

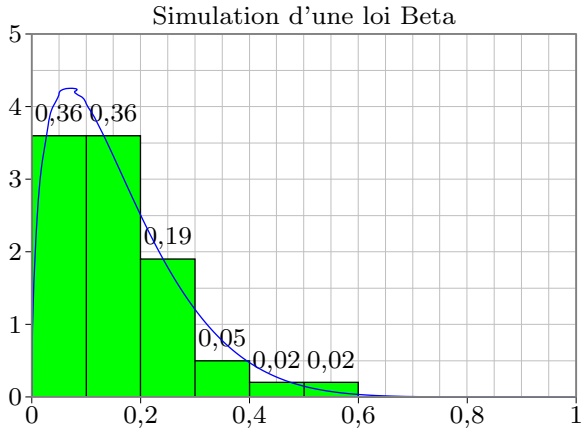
$$ub = \Big(\frac{a-1}{a+b-2}\Big)^{a-1} \times \Big(\frac{b-1}{a+b-2}\Big)^{b-1}$$

de la fonction est atteint. Il nous suffit donc d'utiliser la méthode de rejet combinée avec la propriété de projection pour générer des nombres suivant une loi Beta de paramètres supérieurs ou égaux à 1.

```
// TP2, exercice 9

clear; mode(0);
exec kolmogorov.sci;
exec goodness-of-fit.sci;

function u = random()
    u = grand(1, 1, "def");
endfunction

function x = betadeviate(a, b)
    // local y ub;
    ub = (((a-1)/(a+b-2))^(a-1))*(((b-1)/(a+b-2))^(b-1));
    x = 0; y = ub;
    while y > (x**(a-1))*((1-x)**(b-1));
    x = random();
    y = random()*ub;
    end
endfunction

a = 1.5; b = 8; N = 1000;
x = []; for i = 1:N; x = [x; betadeviate(a, b)]; end
deff("p = F(x)", "p = cdfbet(''PQ'', x, 1-x, a, b)");

scf(3); clf();
cgoftest(x, F);
xtitle("Simulation d''une loi Beta");
```

```
scf(4); clf();
histplot(ceil(1+3.3*log10(N)), x);// sturges
x = linspace(0, 1, 100);
y = (x.^(a-1)).*((1-x).^(b-1))/beta(a, b);
plot(x, y, "-");
```





Annexe : équivalents Maple

Ce code ne marche pas complètement...

```
read("kolmogorov.txt"):
read("goodness-of-fit.txt");

randomseed := 1;

random := proc()
    global randomseed;
    randomseed := modp(427419669081*randomseed+0, 10^12-11);
    return evalf(randomseed/(10^12-11));
end proc:

N := 100: x := array(1..N):

# TP2, exercice 1
# noter qu'on laisse l'affichage pour les param\`etres

PouF := proc(p)
    if random() < 1-p then return 0; else return 1; end if;
end proc:

p := 0.25;
for i from 1 to N do x[i] := PouF(p); end do:
dgoftest(x, [0, 1], [1-p, p]);

discrete := proc(ptheo)
    local u, i;
    u := random();
    for i from 1 to size(ptheo) do
        u := u-ptheo[i];
        if u < 0 then break; end if;
    end do;
    return i;
end proc:
```

```
xtheo := [seq(i, i = 1..6)];
ptheo := [seq(1/6, i = 1..6)];
for i from 1 to N do x[i] := discrete(ptheo); end do:
dgoftest(x, xtheo, ptheo);

# TP2, exercice 3

# Question (i) Loi g\'eom\'etrique G(p).
# La fonction de r\'epartition est F(x)=1-(1-p)^floor(x) pour x >= 0 qui
# s'inverse assez tranquillement.

M := 10;

geometricdeviate := proc(p)
    if p <= 0 then return infinity;
    elif p >= 1 then return 1;
    else return ceil(ln(1-random())/ln(1-p));
    end if;
end proc:

p := 0.25; M := 10:
for i from 1 to N do x[i] := evalf(min(geometricdeviate(p), M)); end do:

xtheo := [seq(i, i = 1..M)]:
ptheo := [seq(p*(1-p)^(i-1), i = 1..M)]:
ptheo[M] := (1-p)^M:

dgoftest(x, xtheo, ptheo);

# Question (ii) Loi E(lambda)

exponentialdeviate := proc(lambda)
    return evalf(-ln(1-random())/lambda);
end proc:

lambda := 1;
for i from 1 to N do x[i] := exponentialdeviate(lambda); end do:

F := x -> 1-exp(-lambda*x):

cgoftest(x, F);

# Question (iii) Loi C(a)

cauchydeviate := proc(a)
    return evalf(a*tan(Pi*(random()-0.5)));
end proc:

a := 1;
for i from 1 to N do x[i] := cauchydeviate(a); end do:

F := x -> evalf(1/Pi*arctan(x/a)+0.5):

cgoftest(x, F);

# Question (iv) Loi N(0, 1)

normaldeviate := proc()
    return stats[statevalf, icdf, normald[0, 1]](random());
    # return evalf(Quantile(Normal(0,1), random()));
end proc:
```

```
for i from 1 to N do x[i] := normaldeviate(); end do:

F := x -> stats[statevalf, cdf, normald[0,1]](x):
# F := x -> evalf(CDF(Normal(0,1), x)):

cgoftest(x, F);

# Question (v) Loi N(m, sigma^2)

gaussiandeviate := proc(m, sigma)
    return stats[statevalf, icdf, normald[m, sigma]](random());
    # return evalf(Quantile(Normal(m,sigma), random()));
end proc:

m := 5; sigma := 2;
for i from 1 to N do x[i] := gaussiandeviate(m, sigma); end do:

F := x -> stats[statevalf, cdf, normald[m,sigma]](x):
# F := x -> evalf(CDF(Normal(m,sigma),x)):

cgoftest(x, F);

# TP2, exercice 5

# Question (i)  Lois binomiales par sch\'ema de Bernoulli fini

binomialdeviate := proc(n, p)
    local x, i; x := 0;
    for i from 1 to n do
        if random() >= 1-p then x := x+1; end if;
    end do;
    return x;
end proc:

n := 10; p := 0.5;
for i from 1 to N do
    x[i] := binomialdeviate(n, p);
end do:

xtheo := [seq(i, i=0..n)]:
ptheo := [seq(binomial(n,i)*p^i*(1-p)^(n-i), i=0..n)]:
dgoftest(x, xtheo, ptheo);

# Question (ii) Lois g\'eom\'etriques par sch\'ema de Bernoulli infini

geometricdeviate := proc(p)
    local x;
    if p <= 0 then return infinity;
    elif p >= 1 then return 1;
    else x := 1;
        while random() < 1-p do x := x+1; end do;
        return x;
    end if;
end proc:

p := 0.25; M := 10:
for i from 1 to N do
    x[i] := evalf(min(geometricdeviate(p), M));
end do:
```

```
xtheo := [seq(i, i = 1..M)];
ptheo := [seq(p*(1-p)^(i-1), i = 1..M)];
ptheo[M] := (1-p)^M;
dgoftest(x, xtheo, ptheo);

# Question (iii) Lois Poisson par files d'attente

poissondeviate := proc (lambda)
    local x, t;
    if lambda <= 0 then return 0;
    else x := -1; t := lambda;
        while t > 0 do
            t := evalf(t+log(1-random()));
            x := x+1;
        end do;
        return x;
    end if;
end proc:

# en passant \'a l'exponentielle on \'economise des calculs

poissondeviate := proc(lambda)
    local x, t;
    if lambda <= 0 then return 0;
    else x := -1; t := evalf(exp(lambda));
        while t > 1 do
            t := evalf(t*random());
            x := x+1;
        end do;
    return x;
    end if;
end proc:

lambda := 1; M := 10;
for i from 1 to N do
    x[i] := evalf(min(poissondeviate(lambda), M));
end do:

xtheo := [seq(i, i=0..M)];
ptheo := [seq(evalf(exp(-lambda)*lambda^i/factorial(i)), i =0..M)];
ptheo[M+1] := 1-add(ptheo[i], i = 1..M);
dgoftest(x, xtheo, ptheo);

# TP2, exercice 6

lastnormaldeviate := infinity:

normaldeviate := proc()
    global lastnormaldeviate;
    local u, v;
    if lastnormaldeviate = infinity then
        u := random(); v := random();
        lastnormaldeviate := evalf(sqrt(-2*ln(u))*sin(2*Pi*v));
        return evalf(sqrt(-2*ln(u))*cos(2*Pi*v));
    else
```

```
        u := lastnormaldeviate; lastnormaldeviate := infinity;
        return u;
    end if;
end proc:

for i from 1 to N do x[i] := normaldeviate(); end do:
F := x -> stats[statevalf, cdf, normald[0,1]](x):
# F := x -> evalf(CDF(Normal(0,1), x)):
cgoftest(x, F);

gaussiandeviate := proc(m, sigma)
    return m+sigma*normaldeviate();
end proc:

for i from 1 to N do x[i] := gaussiandeviate(5, 2); end do:
F := x -> stats[statevalf, cdf, normald[5, 2]](x):
# F := x -> evalf(CDF(Normal(5,2), x)):
cgoftest(x, F);

chisquaredeviate := proc(n)
    local cs, i;
    cs :=0; for i from 1 to n do cs := cs+(normaldeviate())^2; end do;
    return cs;
end proc:

# test ?

studentdeviate := proc(n)
    return normaldeviate()/sqrt(chisquarerand()/n);
end proc:

# test ?

# ETC.
```