Fragment 1

Qu'est-ce que la simulation ?

Consirérons une suite *finie* $x := x_1, \ldots, x_n \in \{0,1\}^n$. Quand peut-on dire qu'elle constitue une réalisation de n variables aléatoires de Bernoulli symétriques indépendantes ? Et bien dans tout les cas bien sûr, et de plus, toutes les suites finies de ce type sont *équiprobables* et de probabilité 2^{-n}. La théorie des probabilité nous enseigne que ces lois uniformes sur $\{0,1\}^n$ s'étendent en une loi $\mathbb{P}$ sur l'ensemble des suites infines $\{0,1\}^{\mathbb{N}^*}$. Cependant, la probabilité d'*une suite infinie donnée* est nulle pour $\mathbb{P}$, et la loi des grands nombres, qui revient dans ce cadre précis à étudier les propriétés *asymptotiques* de la combinatoire des suites finies, nous enseigne que $\mathbb{P}$-presque-toutes les suites infinies sont telles que les fréquences de 0 et de 1 avant le rang n convergent vers 50% quand n tend vers $+\infty$. En d'autres termes, l'ensemble de ces suites infinies particulières est de probabilité 1 pour $\mathbb{P}$. On lira avec profit la remarque 1.2.3 page 24.

Mais qu'est-ce qu'une suite aléatoire au juste ? Pour certains, une suite finie ou infinie qui peut être générée par un algorithme n'est pas aléatoire. De ce point de vue, une suite finie n'est jamais aléatoire ! Deux problèmes différents mais connexes peuvent être distingués

1. Quel sens donner et comment quantifier le caractère « aléatoire » d'une *suite finie ou infinie donnée* ? La notion de complexité algorithmique de Kolmogorov, brièvement présentée dans la section 1.1 qui suit permet de donner une réponse satisfaisante à cette question.
2. Comment produire des suites *finies* qui sont de « bonnes » approximations finies des suites infinies probables correspondant à des réalisations i.i.d. d'une loi donnée ? Comment mesurer la qualité de ces algorithmes ?

Le premier problème pourrait être qualifié de *problème de la caractérisation du hasard absolu.* Le second problème, plus pragmatique mais loin d'être trivial, est celui de la *simulation.* Pour nous, *simuler* une loi de probabilité $\mathcal{L}$ consistera à mettre au point un algorithme utilisable pouvant générer des suites *finies* dont on considèrera que ce sont des réalisations indépendantes de loi $\mathcal{L}$. Une tentative de réponse à ce problème épineux est donnée dans la section 1.2 page 22. À ce stade, il est important de faire la distinction entre deux types de méthodes de génération de suites « aléatoires » :

1. **Les méthodes prédictibles.** Ce sont des méthodes *déterministes*, basées entièrement sur des algorithmes bien établis qui nécessitent d'être initialisés. Ils produisent des suites périodiques avec une très grande période, qui sont entièrement déterminées par l'initialisation utilisée. On parlera de suites *pseudo-aléatoires.* C'est ce type d'algorithmes que nous utiliserons. La *reproductibilité* des suites pseudo-aléatoires qu'ils produisent permettra en outre de tester différentes méthodes numériques avec la même suite pseudo-aléatoire, ce qui est bien paradoxalement bien commode.
2. **Méthodes non-prédictibles.** Ces méthodes sont surtout utiles en cryptographie, où il est capital que le « hasard » utilisé ne soit pas prédictible ni reproductible. Nous n'utiliserons pas ce type de méthodes. Certains parlent à ce propos de « hasard fort » (non-prédictible) par opposition au

« hasard faible » (reproductible). Bien entendu, il ne faut pas oublier que la notion de prévisibilité dépend de l'information dont on dispose[1].

Les générateurs de nombres *pseudo-aléatoires* sont très importants dans les applications. Ils permettent par exemple de simuler des phénomènes dont la modélisation fait appel à des lois de probabilités. La sécurisation des échanges de données dans les réseaux téléinformatiques fait en revanche appel à des méthodes non-prédictibles. Comme nous ne nous intéresserons qu'au problème de la simulation, nous n'utiliserons que des algorithmes prédictibles, qui sont déterministes !

1.1 Complexité algorithmique de Kolmogorov

Cette section peut être ignorée en première lecture et le lecteur intéressé avant tout par les aspects pratiques pourra passer directement à la section 1.2 page 22. Cependant, les notions évoquées ici enrichissent et éclaircissent l'intuition.

Commençons par une petite histoire. Andrei propose à Candide de jouer à pile ou face. Mais voilà, dès le départ, Andrei obtient vingt fois « face » à la suite. Candide est furieux, et à la suite de la vingtième « face » obtenue par Andrei, il accuse ce dernier d'avoir choisit une pièce truquée. Andrei rétorque qu'avec une pièce parfaitement équilibrée, la théorie des probabilités nous enseigne que toute les suites de vingt lancés sont équiprobables et de probabilité 2^{-20}. Ainsi, en codant 1 pour « face » et 0 pour « pile », la suite

$$1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, \tag{1.1}$$

qu'a obtenu Andrei n'est que l'un des 2^{20} éléments également possibles de l'ensemble $\{0,1\}^{20}$ de suites de 0 et de 1 de longueur 20. Andrei modélise les 20 lancés par des réalisations i.i.d. de variables aléatoires de Bernoulli de loi $2^{-1}(\delta_0+\delta_1)$. Et pourtant, la suite (1.1) nous semble, ainsi qu'à Candide, beaucoup moins « aléatoire » qu'une suite du type

$$1,0,0,1,0,1,1,0,1,0,1,0,1,1,1,0,1,0,0,1 \tag{1.2}$$

Y a-t-il un paradoxe ? Mais qu'est-ce qu'une « suite aléatoire » au juste ? La suite (1.1) est très *redondante*, et c'est pour cette raison que Candide ne la trouve pas *aléatoire*. La *fréquence* des 1 y est beaucoup plus élevée que celle des 0. En réalité, cela ne contredit pas la loi des grands nombres, qui nous enseigne que si l'on poursuit les lancés à l'*infini*, les *fréquences* d'apparition des 0 et des 1 sont asymptotiquement égales (et valent 50%). Ainsi, le paradoxe n'en est pas un, et l'intuition probabiliste de Candide est plus proche du résultat asymptotique de la loi des grands nombres que de la combinatoire des suites finies. Dans l'ensemble $\{0,1\}^n$ des suites *finies* de longueur n, la proportion de suites très *redondantes* comme (1.1) diminue avec la taille n des suites considérées et finit par tendre vers zéro quand n tend vers l'infini.

On pourrait croire que notre perception intuitive du caractère aléatoire est liée aux fréquences d'apparition de 0 et 1 dans (1.1) et (1.2). Il n'en est rien. La suite périodique suivante

$$1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0, \tag{1.3}$$

est très régulière, alors même qu'elle contient autant de 0 que de 1. Pour cette raison, elle ne nous parraît pas plus aléatoire que (1.1). Ainsi, notre intuition du caractère aléatoire fait plutôt appel à une sorte de régularité, de prédictibilité. Pour coder brièvement (1.1), il suffit d'écrire « vingt fois un », de même pour

[1] Du hasard fort peut être obtenu à partir de hasard faible en utilisant des « fonctions de hachage », difficilement inversibles en terme de temps de calcul. Les algorithmes non prédictibles font souvent appel à des « sources d'entropie » extérieures liées au fonctionnement de la machinerie informatique comme par exemple le temps séparant deux interruptions du processeur. Sur le système Linux par exemple, on dispose d'une source algorithmique prédictible /dev/urandom et d'une source « entropique » /dev/random utile pour la sécurisation car difficilement prédictible. Un « hasard absolument pas prédictible » pourrait être produit par des dispositifs quantiques, mais à l'heure actuelle, il n'existe pas de périphérique prévu à cet effet dans les ordinateurs vendus dans le commerce.

(1.3), il suffit de dire « cinq fois 1,1,0,0 », tandis que pour (1.2), il n'y a pas vraiment de manière plus courte qui la résume. D'une certaine façon, cette dernière est « incompressible » par un *algorithme*. Cette idée est à la base de la notion de complexité algorithmique.

Un *algorithme* n'est qu'une suite d'*instructions* élémentaires, on parle de *programme*, pouvant être exécutées par une *machine donnée* afin de produire une *sortie*. Considérons une machine informatique M pouvant exécuter des programmes. On dit que cette machine est universelle lorsqu'elle peut émuler n'importe quelle autre machine informatique « raisonnable ». La machine universelle de Turing en est un exemple, et on en trouvera une présentation accessible dans [TG95] par exemple.

On note $\mathcal{P}_M$ l'ensemble des programmes écrits pour la machine M. Pour un programme $p \in \mathcal{P}_M$, on note $l(p)$ sa longueur en nombre d'instructions pour la machine M et $s(p)$ sa sortie. La *complexité de Kolmogorov* $\mathbf{K}_M(x)$, ou complexité algorithmique, d'une suite $x := (x_n)_n$ (finie ou infinie) pour une machine M est définie par :

$$\mathbf{K}_M(x) := \min_{p \in \mathcal{P}_M,\, s(p)=x} l(p).$$

C'est donc la longueur du plus petit programme écrit pour la machine M qui génère la suite x. Les suites (1.1) et (1.3) ont une complexité faible car les programmes qui les génèrent peuvent être très courts. On perçoit alors qu'une suite finie est d'autant plus « complexe » que sa longueur est proche de sa complexité algorithmique.

Reste à savoir dans quelle mesure la fonction $\mathbf{K}_M$ dépend de la machine M, car on peut tout à fait imaginer une machine possèdant des instructions simples pour générer certaines suites complexes. La réponse est la suivante : soit U une machine universelle et M une machine, alors il existe une constante c_M telle que pour toute suite x, on ait :

$$\mathbf{K}_U(x) \leqslant \mathbf{K}_M(x) + c_M.$$

On parle alors d'*universalité* de la complexité de Kolmogorov, en ce sens qu'elle ne dépend pas, à une constante additive près, de la machine considérée. Ainsi, on peut se ramener à une machine universelle dont la définition est élémentaire, comme celle de Turing évoquée plus haut.

Si $x := (x_i)_{i \in \mathbb{N}^*}$ est une suite infinie et n un entier naturel, on note $\tau_n(x) := x_1, \ldots, x_n$ la suite finie de ses n premier termes. La suite x peut alors être considérée comme « aléatoire » si $n \mapsto K_M(\sigma_n(x))$ est du même ordre que la longueur de $\tau_n(x)$. De ce point de vue, les décimales des nombres π, e ou $\sqrt{2}$ ne sont pas aléatoires puisqu'il existe des algorithmes très simples pour les générer ! Cela dit, on montre qu'il n'existe pas d'algorithme pour calculer la complexité de Kolmogorov elle même en toute généralité ! On peut cependant obtenir des encadrements et des estimations. Mais arrêtons là cette digression culturelle, elle nous mènerait trop loin.

Nous n'avons donné ici qu'un maigre apperçu assez rudimentaire de la théorie de la complexité algorithmique de Kolmogorov. Le lecteur intéressé est invité à se plonger dans la lecture de [LV97], qui constitue un traité exhaustif et récent sur cette théorie. Il pourra lire également avec profit [CT91, chap. 7], [Raa01] et enfin se reporter au fragment 7 page 111 qui donne une présentation de l'entropie en théorie de l'information.

Remarque 1.1.1 *(Complexité d'un algorithme).* On parle souvent de complexité d'un algorithme pour désigner le nombre d'opérations élémentaires qu'il nécessite en fonction de la taille des données. La multiplication standard de deux matrices carrées $n \times n$ nécessite environs n^3 additions et multiplications, et l'on dit alors que cet algorithme est de complexité n^3. En quelque sorte, la complexité d'une suite est la plus petite complexité des algorithmes qui la génèrent.

Un peu d'Histoire. Kolmogorov a été le premier à fournir une axiomatisation satisfaisante des probabilités dans les années 1930. Le problème consistant à donner un sens au caractère aléatoire des suites a été étudié par de nombreux scientifiques talentueux comme par exemple von Mises, Wald, Church, et Martin-Löf. Mais l'approche la plus satisfaisante est de loin celle obtenue par la théorie de la complexité,

bâtie sur la théorie de la calculabilité initiée par Church et Turing dès la fin des années 1930. La notion de complexité algorithmique été introduite indépendamment dans les années 1960 par Solomonoff, Chaitin, et Kolmogorov par des biais différents. Kolmogorov était en particulier motivé par le soucis de quantifier le caractère aléatoire des suites finies, qui échappe à la théorie des probabilités. La théorie actuelle est appelée « théorie algorithmique de l'information » ou encore « théorie de la complexité de Kolmogorov ». Elle est reliée à la fois à l'informatique théorique, à la théorie des probabilités, à la statistique, à la théorie de l'information, et enfin à la logique formelle et aux théorème d'incomplétude. La mathématisation des notions de complexité et d'information a fait l'objet de nombreuses recherches récentes reliées à l'informatique et au socle logique des mathématiques. La relative jeunesse de cette théorie fondamentale fait malheureusement qu'elle est rarement enseignée dans les formations généralistes, tout au moins en France, ce qui explique qu'elle ne fasse souvent pas partie du bagage culturel des mathématiciens actuels.

1.2 Loi uniforme et théorème fondamental de la simulation

Un algorithme est toujours déterministe lorsqu'on le connaît ainsi que ses données initiales. Ainsi, les suites *pseudo-aléatoires* générées par les algorithmes prédictibles sont par définition déterministes et n'ont donc rien d'« aléatoire » ni d'« indépendant » – si tant est que cela ait un sens pour une suite finie – et c'est précisément pour cette raison que l'on parle de suites *pseudo*-aléatoires. Dans la pratique, la loi uniforme est la première loi que l'on simule. Comme expliqué par la suite, diverses méthodes justifiées théoriquement permettent ensuite de simuler à partir de la loi uniforme, un grand nombre de lois, plus ou moins quelconques. Simuler la loi uniforme consistera à produire par un algorithme des suites finies de nombres que nous pouvons considérer comme autant de réalisations indépendantes de variables aléatoires uniformes sur $[0,1]$. Mathématiquement, les n sorties successives d'un tel générateur seront considérées comme la donnée de $U_1(\omega),\dots,U_n(\omega)$ pour un $\omega\in\Omega$ où les U_i sont des v.a.r. i.i.d. $U_i:(\Omega,\mathcal{A},\mathbb{P})\to[0,1]$ de loi uniforme.

Matlab permet de simuler une loi uniforme (resp. normale) via la fonction `rand` (resp. `randn`). La bibliothèque Stixbox fournit de nombreux autres générateurs de lois discrètes et continues qui en sont dérivés : faites `help stixbox` ou reportez vous à la table 1.1, page 39, qui donne la liste des générateurs aléatoires de la bibliothèque Stixbox. Scilab fournit également un certain nombre d'exemples de générateurs aléatoires dans le fichier de démonstration `demos/random/random.sci`. De nombreuses méthodes permettent de confronter les suites pseudo-aléatoires fournies par de tels générateurs avec les prédictions de la théorie des probabilités. Les tests statistiques par exemple, comme celui du « chi-deux » ou le test de Kolmogorov-Smirnov, permettent de comparer l'adéquation de la loi empirique avec la loi véritable, ou encore de tester l'indépendance, cf. chapitre 3 page 61. En première lecture, on pourra passer directement au théorème 1.2.2 page 24.

Réductionnisme. La méthode la plus satisfaisante intellectuellement pour aborder le problème de la simulation est de tout ramener à la simulation de la loi de Bernoulli symétrique sur $\{0,1\}$. À partir de là, on peut aisément construire un algorithme de simulation de la loi uniforme sur les dyadiques de $[0,1]$, qui constituent une approximation des nombres réels entre 0 et 1, cf. par exemple section 1.5.1 page 28. Mais une telle approche ne résout rien et ne fait que réduire le problème de la simulation, qui peut alors être scindé en deux sous problèmes bien distincts :

- Simuler la loi uniforme discrète sur $\{0,1\}$;
- En déduire des simulations de toutes les autres lois.

Comme nous allons le voir, le second sous problème s'avère être beaucoup plus facile que le premier ! Dans la pratique, on simule directement la loi uniforme sur les dyadiques. L'informatique étant le règne du fini, les nombres réels ne sont accessibles qu'à travers des approximations. En base 2, L'invervalle $[0,1]$ par exemple est approximé par un ensemble dyadique D_r définit par

$$\mathrm{D}_r := \{k\,2^{-r},\ k\in\mathbb{N},\ 0\leqslant k\leqslant 2^r-1\}. \tag{1.4}$$

Sous Matlab ou Octave, $r = 53$. Ainsi, nous ne pourrons véritablement simuler que des lois sur des ensembles finis. Simuler n réalisations de la loi uniforme sur $\{0, \ldots, m\}$ pourrait consister à choisir l'une des $(m+1)^n$ suites équiprobables dans $\{0, \ldots, m\}^n$. Mais voilà, tout réside dans ce choix. Comment choisir une suite dans cet ensemble ? La réponse n'est pas évidente dès que $m > 0$. Nous savons par la loi des grands nombres que les seules suites *infinies* ($n = +\infty$) probables possèdent une propriété d'équi-répartition. Les générateurs de suites *pseudo-aléatoires* tentent de produire des suites *finies* qui ont des propriétés similaires, tout en ayant une loi empirique assez proche de la loi uniforme, ce qui suppose une sorte de propriété d'homogénéité.

Un *algorithme* dont la sortie ne peut prendre qu'un nombre *fini* de valeurs a forcément une sortie *périodique*. Ainsi, le premier critère de qualité pour un algorithme de ce type est d'avoir une *très grande période*. Il n'y a pas vraiment de définition universelle de ce que serait un « bon générateur de suites pseudo-aléatoires », et il est toujours possible de construire un test qui montre les limites d'un générateur algorithmique. Cela dit, plusieurs chercheurs[2] ont proposé des critères que devront satisfaire des algorithmes « décents », souvent liés à des propriétés d'équi-répartition. Dans la pratique, le problème est de trouver des algorithmes rapides qui satisfont au plus de critères possibles. Cet épineux problème fait toujours l'objet de recherches à l'heure actuelle, en connexion avec la théorie des nombres et l'arithmétique des corps finis[3]. Mais que sont donc ces critères ? Deux des plus connus sont le test spectral et le test de k-distribution, cf. [Knu81]. Expliquons brièvement en quoi consiste ce dernier.

Définition 1.2.1 (Critère de k-distribution). Soit r et k des entiers strictement positifs et $v \in \{1, \ldots, r\}$. Soit $\mathbf{x} := (x_n)_{n\in\mathbb{N}}$ une suite *pseudo-aléatoire* de période p, et à valeurs dans l'ensemble $\{i\,2^{-r},\ i \in \mathbb{N},\ 0 \leqslant i \leqslant 2^r - 1\}$. Pour tout $j \in \{0, \ldots, p-1\}$, on définit le vecteur $y_j \in [0,1]^k \subset \mathbb{R}^k$ par $y_j := (x_j, x_{j+1}, \ldots, x_{j+k-1})$. Partitionnons à présent le cube $[0,1]^k$ de $\mathbb{R}^k$ en 2^{kv} petits cubes de même volume, correspondant à la subdivision uniforme de chacune de ses arrêtes en 2^v sous-intervalles de même longueur 2^{-v}. On dit alors que la suite $\mathbf{x}$ est *k-distribuée avec une précision de v bits* si et seulement si chaque petit cube contient le même nombre de points de la suite $(y_j)_{0\leqslant j\leqslant p-1}$, sauf peut être le petit cube contenant l'origine, qui peut en contenir un de moins. Pour chaque v, on note $k(v)$ le plus grand entier k pour lequel cette propriété a lieu.

Bien entendu, la période p impose une limite supérieure, et par un raisonnement élémentaire on montre que l'on a toujours la majoration $2^{k(v)v} - 1 \leqslant p$. La propriété de k-distribution correspond à une sorte d'*équi-répartition*. De telles suites ont une mesure empirique qui ressemble à celle de la loi uniforme, et de plus, les valeurs que prend la suite sont relativement bien mélangées. Des tests statistiques peuvent être utilisés pour vérifier si ces suites sont acceptables, cf. fragment 3 page 61. Une discussion plus approfondie nous mènerait bien trop loin, le lecteur curieux trouvera de nombreux développements récents dans la littérature. Le très bon livre [Knu81] de Knuth est malheureusement un peu ancien.

Lehmer a été l'un des premiers à proposer, dans les années 1950, des algorithmes générant des nombres *pseudo-aléatoires*. Ses algorithmes, dits « multiplicatif à congruences », consistent typiquement à calculer une suite récurrente $(x_n)_n$. Sur un ordinateur, les nombres réels sont toujours approximés, avec une certaine précision. Reprenons l'exemple où l'intervalle $[0,1]$ est approximé par l'ensemble fini D_r définit en (1.4), et l'entier $p = 2^r \in \mathbb{N}^*$ joue le rôle de précision. Un *pseudo-réel* x dans $[0,1]$ s'écrit donc y/p où $y \in \{0, \ldots, p\}$. On construit alors x_n par la relation de récurrence $x_n = y_n/p$ où $y_{n+1} \equiv a y_n [m]$ et où a et m sont des entiers fixés et m est assez grand. En général, la période de $(y_n)_n$ est plus petite que m. Plus précisément, si a et m sont premiers entre eux, et si $0 < y_0 < m$, alors y_n n'est jamais nul et la période de $(y_n)_n$ n'est rien d'autre que l'ordre de a dans $\mathbb{Z}_m$. On rappelle que d'après le (petit) théorème de Fermat-Euler, l'ordre de a divise l'indicateur d'Euler de a (qui vaut $m-1$ quand m est premier). Encore une fois, il est important que la période soit très grande, de manière à fournir de très longues suites de nombres pseudo-aléatoires non périodiques.

[2] Knuth, Marsaglia, L'Écuyer, Matsumoto, ...
[3] En particulier, l'algèbre des polynomes sur $\mathbb{F}_2$.

Dans certaines anciennes bibliothèques de calcul scientifique, on utilise $(a, m) = (13^{13}, 2^{59})$ ou encore $(a, m) = (7^5, 2^{31} - 1)$, cf. [Bou86, page 74]. Ce dernier couple correspond exactement à l'algorithme multiplicatif à congruences utilisé par Matlab 4 pour implémenter la fonction `rand`. Certains algorithmes anciens ont été abandonnés après avoir montré leur limites dans des simulations complexes[4]. À partir de la version 5, Matlab utilise un algorithme différent inspiré par Marsaglia, qui ne fait pas appel aux multiplications ou divisions. Il est nettement plus performant que les algorithmes multiplicatifs à congruence et sa période est proche de 2^{1492}, soit environ 10^{449}. L'implémentation de la fonction `rand` dans Octave fait appel à un algorithme dû à Matsumoto & Nishimura appelé « Mersenne Twister[5] », de période $2^{19937} - 1$ soit environ 10^{6001}, et qui est de plus 623-distribué avec une précision de 32 bits.

Notons que contrairement à la véritable loi uniforme sur $[0, 1]$, la loi uniforme sur D_r peut prendre la valeur 0 avec probabilité 2^{-r}. Pour $r = 53$, ceci vaut environ 10^{-16}, ce qui n'est pas si mal.

Le théorème suivant nous assure la possibilité, théorique, de simuler n'importe quelle loi de probabilité sur $\mathbb{R}^d$ à partir de la loi uniforme sur $[0, 1]$. Il ne donne pas vraiment d'algorithme de simulation, mais plutôt une réduction générale du problème théorique de la simulation à celui de la simulation de la loi uniforme.

Théorème 1.2.2 (Théorème fondamental de la simulation). *Soit $m > 1$ un entier, et $U_1, \ldots, U_m$ des variables aléatoires i.i.d. de loi uniforme sur $[0, 1]$. Pour toute loi de probabilité μ sur $\mathbb{R}^d$, il existe une fonction borélienne $f_{\mu,m} : \mathbb{R}^m \to \mathbb{R}^d$ dont l'ensemble des points de discontinuité est Lebesgue-négligeable telle que la variable aléatoire $Y := f_{\mu,m}(U_1, \ldots, U_m)$ suit la loi μ.*

Ce théorème donne un résultat théorique intéressant, bien que l'on ne s'en serve pas dans la pratique. On en trouvera une preuve dans [Bou86, chap. X page 267]. En lisant la preuve, on s'aperçoit que la fonction f peut être explicitée. On peut donc théoriquement simuler toute loi à partir d'une simulation de la loi uniforme. En effet, supposons que nous désirions obtenir n réalisations indépendantes de loi μ. Prenons alors $m = 1$ dans le théorème précédent. Un générateur pseudo-aléatoire uniforme, du type `rand`, nous fournit une suite $U_1(\omega), \ldots, U_n(\omega)$, qui sont considérées comme les réalisations pour $\omega \in \Omega$ de n v.a.r. i.i.d. suivant une loi uniforme sur $[0, 1]$. Le théorème précédent affirme alors l'existence d'une fonction f_μ telle que $f_\mu(U_1(\omega)), \ldots, f_\mu(U_n(\omega))$ soient les réalisations de n v.a. i.i.d. de loi μ. Dans la pratique, on préférera utiliser des méthodes plus efficaces, au cas par cas.

Remarque 1.2.3 *(Échantillonage fini et infini).* Remarquons que d'après la loi des grands nombres, si

$$(X_n : (\Omega, \mathcal{A}, \mathbb{P}) \to \mathbb{R}^d, n \in \mathbb{N}^*)$$

est une suite de v.a. i.i.d. de loi μ, en d'autre termes un « échantillon de taille infinie », alors pour μ-presque tout $\omega \in \Omega$, la mesure empirique

$$\mathcal{E}_n(\omega) := \frac{1}{n} \sum_{i=1}^{n} \delta_{X_i(\omega)}$$

converge étroitement vers la loi μ. Il en découle que pour μ-presque tout $\omega \in \Omega$, la donnée de la *suite infinie* $(X_n(\omega), n \in \mathbb{N}^*)$ caractérise *théoriquement* la loi μ. Un générateur de nombres pseudo-aléatoires ne fournit que des *suites finies* $(X_1(\omega), \ldots, X_n(\omega))$, qui ne suffisent pas à caractériser la loi μ. De même, les échantillons provenant de données réelles utilisés en statistique sont toujours *finis*, et l'un des objectifs principaux de la statistique est de se servir de ces suites finies de ce type pour approximer ou estimer la loi μ de certains phénomènes, à laquelle on n'a pas directement accès.

Remarque 1.2.4 *(Initialisation des générateurs aléatoires).* La valeur initiale du générateur pseudo-aléatoire de loi uniforme `rand` peut être choisie en général. Dans Matlab 5, l'état interne du générateur consiste

[4] Simulation du modèle d'Ising en physique par exemple.

[5] Publié en 1998, cf. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

en un vecteur de longueur 35, que l'on peut obtenir par la commande `s = rand('state')` et modifier par la commande `rand('state',s)`. Ce vecteur est utilisé pour produire la sortie de la fonction `rand`, qui peut prendre toutes les valeurs de l'ensemble dyadique $\{k\,2^{-53},\ k\in\mathbb{N},\ 0\leqslant k\leqslant 2^{53}-1\}$. Les états internes du générateur sont numérotés et peuvent être sélectionnés en utilisant la commande `rand('state',s)` avec un `s` entier, et le cas `s=0` correspond alors à la valeur initiale par défaut. On peut rendre le générateur un peu moins prédictible en utilisant l'heure courante pour sélectionner l'état interne : `rand('state',sum(100*clock))`, mais comme expliqué précédemment, cela est aussi naïf qu'inutile ! Dans Octave 2.1, l'état interne du générateur consiste en un vecteur de taille 625, et peut être obtenu par la commande `s = rand("state")`, et modifié par la commande `rand("state",s)`. Idem pour Scilab. Enfin, il faut se souvenir qu'il peut être très utile de sauvegarder l'état interne du générateur avant de faire une simulation, de façon à pouvoir reproduire exactement la même suite pseudo-aléatoire ultérieurement pour faire des comparaisons par exemple.

1.3 Simulation de lois par leur fonction de répartition

Soit μ une loi sur $\mathbb{R}$ de fonction de répartition F. Si G désigne l'inverse continue à gauche de F, définie pour tout y dans $[0,1]$ par

$$G(p) := \inf\{x : F(x) \geqslant p\},$$

alors, pour toute v.a. uniforme U sur $[0,1]$, la v.a. $G(U)$ est de loi μ. Par exemple, pour la loi de Cauchy, de densité $(\pi(1+x^2))^{-1}$ sur $\mathbb{R}$, on a $F(x) = \arctan(x)/\pi + 1/2$ et $G(p) = \tan(\pi(p-1/2))$. La bibliothèque Stixbox pour Matlab fournit les fonctions de répartitions inverses pour quelques lois usuelles, cf. tableau 1.1 page 39.

Cette méthode, dont la simplicité peut séduire, souffre malheureusement de certains handicaps. En effet, la fonction F^{-1} peut être coûteuse à simuler, par exemple lorsque son expression fait intervenir des fonctions transcendantales comme log, $\sqrt{}$, cos, etc. D'autre part, comme U est simulée par un générateur de la loi uniforme, les valeurs possibles sont en réalité en nombre fini sur $[0,1]$, et leur répartition est alors « dilatée » par F^{-1} lors de l'évaluation de $F^{-1}(U)$. Ainsi, cette méthode n'est pas très satisfaisante par exemple pour les valeurs de U proches des points où F a une dérivée nulle ou très petite.

Théorème 1.3.1 (Simulation de la loi exponentielle). *Soit U une v.a.r. de loi uniforme sur $[0,1]$ et $\lambda > 0$ un réel, alors la v.a.r. $-\lambda^{-1}\log U$ suit la loi exponentielle de paramètre λ.*

Un exemple d'illustration en Matlab est donné ci-dessous, et sa sortie graphique par la figure 1.1. Cette méthode s'applique bien entendu aux lois de Weibull, cf. section 9.4.8 page 139, dont les lois exponentielles ne constituent qu'un cas très particulier. La biliothèque Stixbox fournit la fonction `rexpweib` qui permet d'obtenir des réalisations de loi de Weibull et donc en particulier de loi exponentielle. Bien entendu, sa description s'obtient par `help rexpweib` et le code correspondant par `type rexpweib`.

```
lambda=0.5; clf; hold on;
title('Simulation d''une loi exponentielle')
ylabel('Densite'); xlabel('Valeurs');
[E,C]=histo(-log(rand(500,1))/lambda,100,0,1);
plot(C,lambda*exp(-lambda*C),'k-*')
legend('Empirique','Theorique')
```

Comme expliqué précédemment, le logarithme utilisé pour simuler la loi exponentielle dilate la répartition des valeurs prises par le générateur de la loi uniforme, provoquant en particulier une raréfaction des grandes valeurs possibles, et donc un mauvais comportement « autour de l'infini ». En d'autres termes, si le générateur uniforme U prend ses valeurs dans l'ensemble dyadique D_r définit par (1.4), alors $-\log U$

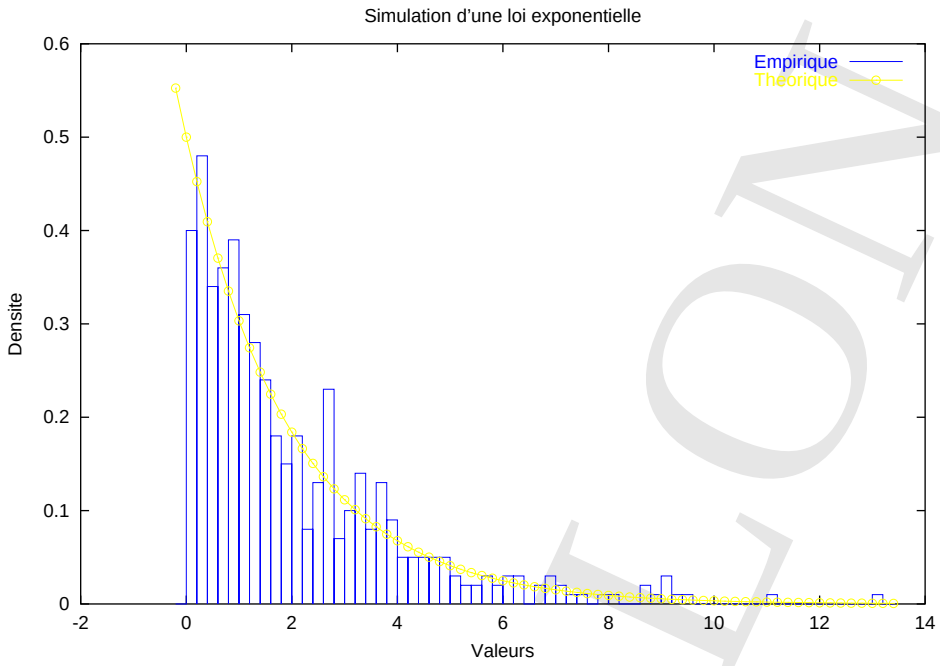

FIG. 1.1 – Simulation de la loi exponentielle de paramètre $1/2$ au moyen de sa fonction de répartition.

prendra ses valeurs dans l'ensemble

$$-\log \mathrm{D}_r = \{r\log 2 - \log k \;\; k \in \mathbb{N},\; 0 \leqslant k \leqslant 2^r - 1\},$$

ce qui est d'autant plus mauvais que k est proche de 0. Une méthode pour contourner au moins en partie ce problème consisterait à utiliser un algorithme de « polygonalisation » du même type que celui utilisé pour la loi gaussienne, cf. section 1.6.3 page 32. Contrairement au cas gaussien, la tâche est beaucoup plus simple ici car la fonction de répartition inverse est connue explicitement et est très simple.

1.4 Simulation de lois par la méthode du rejet

On attribue parfois cette méthode à J. von Neumann. Soit une loi μ de densité f sur $\mathbb{R}$, que l'on désire simuler.

1. Supposons que f est continue et à support compact $[a,b]$. Le graphe de f est donc contenu dans un rectangle $[a,b]\times[0,M]$. On considère alors des vecteurs aléatoires indépendants $((X_n,Y_n))_{n\geqslant 0}$ suivant une loi uniforme sur $[a,b]\times[0,M]$. On définit ensuite T comme le plus petit n tel que $f(X_n)\leqslant Y_n$. Alors on montre que T est fini presque sûrement et que X_T a pour loi μ.
2. Supposons qu'il existe une densité g facile à simuler telle que $f\leqslant cg$ pour une constante $c>0$. Soit alors $(W_n)_n$ et $(U_n)_n$ deux suites de v.a.r. i.i.d. indépendantes avec les W_n de loi de densité g et les U_n de loi uniforme sur $[0,1]$. On pose $Y_n = cU_ng(W_n)$ et T le plus petit n tel que $Y_n < f(W_n)$. Alors la loi de U_T a pour densité f.

Ces deux méthodes s'étendent sans difficultés à des lois à densité sur $\mathbb{R}^d$. Les méthodes de rejet ont été développées et améliorées, et sont largement utilisées dans la pratique. Elles ont l'avantage de nécessiter

peu de calculs lorsqu'elles sont couplées à des tables. La bibliothèque Stixbox fournit trois fonctions `rjbinom`, `rjgamma` et `rjpoiss` qui permettent d'obtenir des réalisations de lois binomiales, gamma et de Poisson avec la méthode du rejet. On pourra consulter [Bou86, IV.4.3, p. 79] ou encore [BL98, VI.5.15, p. 180] pour des preuves.

1.5 Simulation de lois discrètes

Soit $p_1, \dots, p_n$ des nombres dans $[0,1]$ tels que $p_1 + \cdots + p_n = 1$. En partitionnant l'intervalle $[0,1]$ en morceaux adjacents de longueurs $p_1, \dots, p_n$, il vient :

Théorème 1.5.1 (Simulation d'une loi discrète à support fini). *Soit U une v.a.r. de loi uniforme sur $[0,1]$ et $P := p_1\delta_{x_1} + \cdots + p_n\delta_{x_n}$ une loi discrète où les x_i sont tous différents. Alors la v.a.r.*

$$x_1 \mathrm{I}_{\{U<p_1\}} + x_2 \mathrm{I}_{\{p_1 \leqslant U \leqslant p_1+p_2\}} + \cdots + x_n \mathrm{I}_{\{p_1+\cdots+p_{n-1} \leqslant U \leqslant 1\}}$$

suit la loi P.

Le code Matlab suivant fournit la fonction `rdist` qui implémente le résultat du théorème 1.5.1 :

```
function realis = rdist(x,p)
%RDIST  réalisation aléatoire d'une loi discrète à support fini (atomique).
% realis = rdist(x,p)
% p     est un vecteur de réels positifs tel que sum(p)=1.
% x     est un vecteur de réels de même taille que p, donnant le support.
% realis = réalisation de la loi discrète tq P(x_1)=p_1,...,P(x_n)=p_n.
% Les réalisations successives sont considérées comme indépendantes car
% elles correspondent à des appels successifs de rand.

n=length(p);
r=rand;
a=0;
b=p(1);
for i=1:n-1,
  if ((r>=a)&(r<=b))
        realis=x(i);
        return;
  end
  a=b;
  b=b+p(i+1);
end
realis=x(n);
return;
```

Exercice 1.5.2. Montrez que cette méthode s'étend sans difficultés aux lois discrètes à support dénombrable. Donnez un petit programme d'illustration.

La bibliothèque Stixbox fournit la fonction `quantile` qui permet d'obtenir les quantiles d'une loi à partir d'un échantillon empirique. Faites `help quantile`.

Le nombre moyen d'instructions `if` exécutées dans le code de la fonction `rdist` est voisin de $\log n$. On pourra consulter par exemple [Knu81] pour des méthodes plus rapides faisant intervenir des tables précalculées à partir des probabilités p_i. La loi uniforme discrète peut être simulée très rapidement, sans aucune instruction conditionnelle, comme le montre le code suivant :

```
function realis = randiscr(x,n,m)
%RANDISCR
% Similaire à rand. Renvoie des réalisations i.i.d. de loi uniforme
% sur x(1),...,x(length(x)).
% n et m sont des entiers optionnels, valant 1 par défaut.

if (nargin==0),
  error('Pas assez de paramètres');
elseif (nargin==1),
  n=1; m=1;
elseif (nargin==2)
  m=1;
elseif (nargin>3)
  error('Trop de paramètres');
end
realis=reshape(x(ceil(length(x)*rand(n,m))),n,m);
return;
```

Remarque 1.5.3. La commande Matlab `randperm` fournit une permutation pseudo-aléatoire. En d'autres terme, elle constitue un générateur de la loi uniforme sur le groupe symétrique.

Théorème 1.5.4 (Simulation de la loi binomiale). *Soient $U_1, \dots, U_n$ des v.a.r. i.i.d. de loi uniforme sur $[0,1]$, alors la v.a.r.*

$$\sum_{i=1}^{n} \mathrm{I}_{\{U_i<p\}}$$

suit une loi binomiale de taille n et de paramètre p.

Théorème 1.5.5 (Simulation de la loi de Poisson). *Soient $(E_i)_{i\geqslant 1}$ des v.a.r. i.i.d. de loi exponentielle de paramètre $\lambda > 0$, alors la v.a.r.*

$$\mathrm{I}_{\{E_1\leqslant 1<E_1+E_2\}} + 2\mathrm{I}_{\{E_1+E_2\leqslant 1<E_1+E_2+E_3\}} + 3\mathrm{I}_{\{E_1+E_2+E_3\leqslant 1<E_1+E_2+E_3+E_4\}} + \cdots$$

suit une loi de Poisson de paramètre λ. Il en est de même pour :

$$\mathrm{I}_{\{E_1\leqslant 1\}} + \mathrm{I}_{\{E_1+E_2\leqslant 1\}} + \mathrm{I}_{\{E_1+E_2+E_3\leqslant 1\}} + \cdots$$

Comme nous le verrons dans la section 6.1 page 101, cette méthode correspond exactement à la simulation de la loi de N_1 où $(N_t)_{t\geqslant 0}$ est un processus de Poisson simple d'intensité λ. L'événement $\{E_1 + \cdots + E_i \leqslant 1 < E_1 + \cdots + E_{i+1}\}$ correspond au cas où le processus a sauté exactement i fois avant le temps 1.

1.5.1 Lois fractales

Soit $n \in \mathbb{N}^*$ une base de numération. Chaque résultat d'un schéma de Bernoulli consistant en une suite d'expériences i.i.d. $(X_i, i \in \mathbb{N}^*)$ à n issues possibles de loi uniforme sur $\{0, \dots, n-1\}$ peut se représenter par l'écriture en base n d'un nombre réel dans $[0,1]$. En d'autres termes, on a la surjection $\mathcal{I}_n$ suivante :

$$\mathcal{I}_n : (x_1, \dots, x_k, \dots) \in \{0, \dots, n-1\}^{\mathbb{N}^*} \mapsto \sum_{i=1}^{+\infty} n^{-i}\, x_i = \underbrace{0, x_1 \cdots x_k \cdots}_{\text{en base } n} \in [0,1].$$

On pourra penser à la base $n = 2$ et au schéma de Bernoulli classique de pile ou face avec une pièce équilibrée. La variable aléatoire U définie par :

$$U := \mathcal{I}_n(X_1, \ldots, X_k, \ldots) := \sum_{i=1}^{+\infty} n^{-i} X_i,$$

suit une loi uniforme sur $[0, 1]$. En effet, si $a := a_1 n^{-1} + \cdots + a_m n^{-m}$ est un nombre n-adique, on a :

$$\mathbb{P}(a < U < a + n^{-m}) = \mathbb{P}(X_1 = a_1, \ldots, X_m = a_m) = \mathbb{P}(X_1 = a_1) \cdots \mathbb{P}(X_m = a_m) = n^{-m}.$$

Réciproquement, ce calcul montre que les coefficients $(X_m)_{m \geqslant 1}$ de l'écriture en base n d'une variable aléatoire uniforme sur $[0, 1]$ sont i.i.d. de loi uniforme sur $\{0, \ldots, n-1\}$. Remarquons que l'ensemble des nombres n-adiques est de mesure nulle pour la loi uniforme. Ainsi, les nombres de $[0, 1]$ dont l'écriture en base n est constante à partir d'un certain rang « ne comptent pas » en quelque sorte, et $\mathcal{I}_n$ est presque-sûrement une injection, et donc une bijection p.s.

Exercice 1.5.6. En déduire une méthode pour générer d'un seul coup k réalisations indépendantes de loi uniforme sur $\{1, \ldots, n\}$ à partir d'une réalisation de précision k en base n d'une loi uniforme sur $[0, 1]$. Implémenter ce résultat en écrivant une fonction Matlab.

Que se passe-t-il pour U si l'on suppose que les X_i ne suivent plus une loi uniforme sur $\{0, n-1\}$ mais plutôt une loi $\{p_1, \ldots, p_n\}$ quelconque fixée d'avance ? En d'autres termes, quelles sont les lois images possibles pour la fonction $\mathcal{I}_n$ sur $[0, 1]$? Comme $(n^r U) \bmod n^r$ a la même loi que U pour tout $r \in \mathbb{N}$, le graphe de la fonction de répartition de U possède une propriété d'invariance d'échelle : c'est un fractal. La fonction de répartition de U est continue, mais la la loi de U est étrangère à la mesure de Lebesgue et les lois de U sont toutes étrangères entre elles quand la loi des X_i balaye l'ensemble des lois discrètes sur $\{0, \ldots, n-1\}$. L'exemple de code pour Octave qui suit avec sa sortie graphique 1.2 page 30 illustre ce phénomène. On pourra consulter [DCD82b, exe. 3.3.15 page 81], [BL98, expl. IV.3.6.iii pages 100-103 et V.5.3 page 144] et enfin [BL98, exe. V.6.15 page 155] à propos de ces « lois fractales »

```
%% Lois Fractales dans un schéma de Bernoulli à n issues non uniformes
clear;
more off;
nr=1000;
prec=100;
%
base1=2;
p1=1/4;
sprintf('Cas des lois de Bernoulli de parametre %f.\n',p1)
sprintf('%d réal. indép. de loi de Bernoulli sur {0,1}...\n',\
 prec*nr)
coefs=repmat(cumprod(ones(prec,1)/base1),1,nr);
realis=sum(rbinom([prec,nr],base1-1,p1).*coefs);
[effectifs,classes]=hist(realis,prec*10);
clf;
title('Differents cas pour la fonction de repartition de U');
hold on;
plot(classes,classes);
plot(classes,cumsum(effectifs)/nr);
%
base2=7;
p2=5/9;
sprintf('Cas des lois binomiales de taille %d et de parametre %f.\n',\
```

```
 base2,p2)
sprintf('%d réal. indép. de loi binomiales sur {0,1}...\n',\
 prec*nr)
coefs=repmat(cumprod(ones(prec,1)/base2),1,nr);
realis=sum(rbinom([prec,nr],base2-1,p2).*coefs);
[effectifs,classes]=hist(realis,prec*10);
plot(classes,cumsum(effectifs)/nr);
%
base3=2;
p3=1/2;
sprintf('Cas des lois de Bernoulli symétriques %f.\n',base3,p3)
sprintf('%d réal. indép. de loi de Bern. sym. sur {0,1}...\n',\
 prec*nr)
coefs=repmat(cumprod(ones(prec,1)/base3),1,nr);
realis=sum(rbinom([prec,nr],base3-1,p3).*coefs);
[effectifs,classes]=hist(realis,prec*10);
plot(classes,cumsum(effectifs)/nr);
%
xlabel('Valeur');
ylabel('Fonction de repartition');
legend('Loi uniforme',\
 sprintf('Avec des Bernoulli %1.2f',p1),\
 sprintf('Avec des binomiales (%d, %1.2f)',base2,p2),\
 sprintf('Avec des Bernoulli symetriques'),\
 4);
```

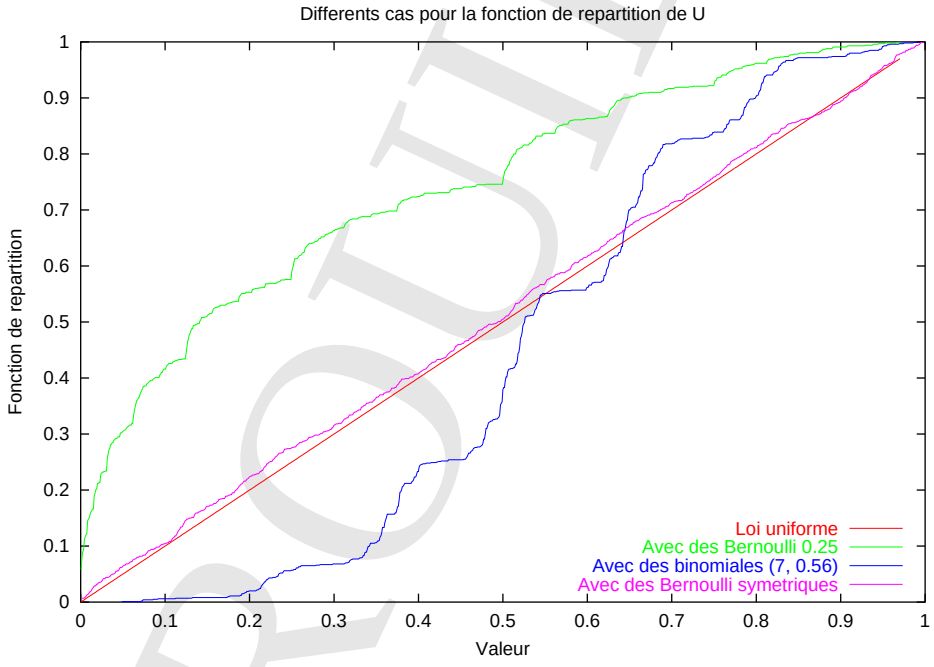

FIG. 1.2 – Lois fractales associées à un schéma de Bernoulli.

1.6 Simulation des lois gaussiennes

Les meilleures méthodes, aussi bien pour leur rapidité que pour leur qualité, sont sans doute celles présentées dans la section 1.6.3 page 32. Cependant, nous commençons par deux méthodes basées sur l'invariance par rotation des lois gaussiennes, qui sont à la fois très classiques et... très lentes.

1.6.1 Méthode polaire pure (algorithme de Box-Muller)

Théorème 1.6.1 (Simulation d'une loi normale). *Soit* $(X,Y) = (r\cos(\theta), r\sin(\theta))$ *un vecteur aléatoire de* $\mathbb{R}^2$. *Alors* (X,Y) *suit une loi* $\mathcal{N}(0,\mathrm{I}_2)$ *si et seulement si* $r^2 := X^2+Y^2$ *et* $\theta := \arg(X+iY)$ *sont indépendantes et suivent respectivement une loi exponentielle de paramètre* $1/2$ *et une loi uniforme sur* $[0,2\pi]$.

Il en découle que si U_1 et U_2 sont indépendantes et uniformes sur $[0,1]$, alors les v.a.r.

$$\sqrt{-2\log U_1}\cos(2\pi U_2) \quad \text{et} \quad \sqrt{-2\log U_1}\sin(2\pi U_2)$$

sont i.i.d. de loi normale centrée réduite. Bien entendu, pour tout $(m,\sigma)\in\mathbb{R}\times\mathbb{R}_+^*$, une variable aléatoire réelle Z suit la loi $\mathcal{N}(0,1)$ si et seulement si la v.a.r. $m+\sigma Z$ suit la loi $\mathcal{N}(m,\sigma^2)$. Nous avons donc là une méthode de simulation de lois gaussiennes unidimensionnelles quelconques. Voici un exemple d'illustration dont la sortie graphique est donnée par la figure 1.3 :

```
clf; hold on;
title('Simulation d''une loi normale');
ylabel('Effectifs'); xlabel('Valeurs');
[E,C]=histo(sqrt(-2*log(rand(5000,1))).*cos(2*pi*rand(5000,1)),100,0,1);
plot(C,(2*pi)^(-1/2)*exp(-C.^2/2),'k-*');
legend('Empirique','Theorique');
```

Cette méthode de simulation polaire est coûteuse en temps de calcul car elle nécessite l'évaluation des fonctions transcendantales log, $\sqrt{}$ et cos. En réalité, on peut supposer que la partie radiale $r^2 = -\frac{1}{2}\log U_1$, qui est de loi exponentielle, est fournie par un générateur auxiliaire pour la loi exponentielle. D'autre part, l'évaluation de la fonction cos peut être évitée en faisant appel partiellement à la méthode du rejet, comme décrit ci-après.

1.6.2 Méthode polaire - rejet

Si $(X,Y) = (\rho\cos(\theta), \rho\sin(\theta))$ suit la loi uniforme sur le disque unité du plan, alors le vecteur aléatoire

$$\frac{\sqrt{-4\log(\rho)}}{\rho}\,(X,Y)$$

suit une loi normale centrée réduite bidimensionnelle. La loi de (X,Y) est facile à simuler par la méthode du rejet à partir d'une loi uniforme sur le carré $[-1,+1]^2$ (rejet dans 21% des cas puisque $\pi/4 \simeq 0.79$).

On remarquera qu'ici, θ et ρ sont indépendantes, que ρ suit une loi de densité $u\mapsto 2u\mathrm{I}_{[0,1]}(u)$ et que θ suit une loi uniforme sur $[0,2\pi]$. D'autre part, on a

$$\frac{\sqrt{-4\log(\rho)}}{\rho}\,(X,Y) = \sqrt{-4\log(\rho)}\,(\cos(\theta),\sin(\theta)).$$

On vérifie alors facilement que $-4\log(\rho)$ suit une loi exponentielle de paramètre $1/2$:

$$\int_0^1 f(-4\log(\rho))\,2\rho\,d\rho = \int_0^{+\infty} f(u)\,\frac{1}{2}\,e^{-u/2}\,du.$$

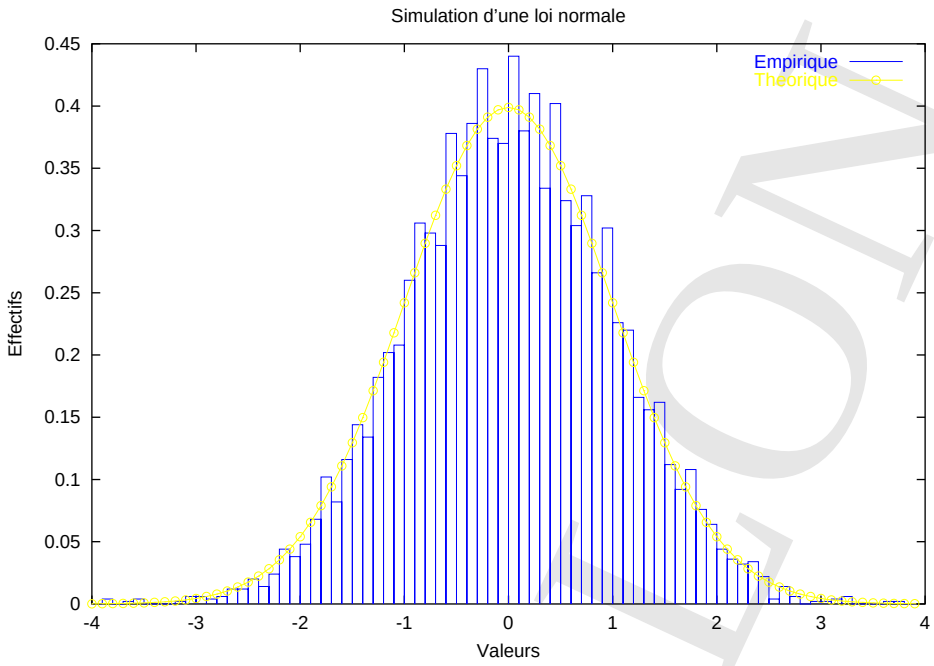

FIG. 1.3 – Simulation de la loi normale par méthode « polaire »

Cette méthode, que l'on peut qualifier de « polaire - rejet », est par exemple utilisée dans Matlab 4 pour implémenter la fonction `randn`. Elle demeure cependant assez lente, en raison à la fois des 21% de rejet et de l'évaluation des fonctions transcendantales. Les défauts de la méthode polaire pure ne sont donc pas totalement éliminés.

1.6.3 Méthode de polygonalisation (algorithme de Marsaglia)

La fonction `rnorm` de Stixbox fait en réalité appel à la fonction `randn` de Matlab, et cette dernière est une fonction interne (« built-in »), comme on le comprend immédiatement en exécutant les deux commandes `type randn` et `type rnorm` sous Matlab. Pour implémenter `randn`, les versions 5 et 6 de Matlab ainsi que la plupart des logiciels professionnels utilisent une méthode due à Marsaglia, appelée parfois « algorithme du Ziggurat[6] », qui consiste en une « polygonalisation » précalculée de la densité gaussienne. Cette méthode, lorsqu'elle est bien programmée, est quasiment aussi rapide que le générateur de loi uniforme sur lequel elle est basée. Il en existe de nombreuses variantes, et l'on pourra par exemple en trouver une version un peu ancienne dans le tome II du livre de Knuth : « The Art of Computer Programming » [Knu81].

L'idée à la base de ces méthodes n'est pas vraiment spécifique à la loi gaussienne. Si μ est une loi de probabilité sur $\mathbb{R}$, de densité f continue par rapport à la mesure de Lebesgue, nous savons que l'aire de la région $\mathcal{R}_\mu$ du plan $\mathbb{R}^2$ figurant entre l'axe des abscisses et la courbe de f vaut exactement 1. La méthode du rejet présentée dans la section 1.4 nous enseigne que simuler selon la loi μ correspond à considérer l'abscisse d'un point aléatoire de $\mathbb{R}^2$ choisit selon la loi uniforme sur $\mathcal{R}_\mu$. La méthode consiste alors à

[6] Il s'agit de temples de Mésopotamie, dont la forme consiste en un empilement de parallélépipèdes rectangles de plus en plus petits.

approximer cette région par une réunion au plus dénombrable de polygones P_i dont l'aire p_i est connue, avec $\sum_{i=0}^{+\infty} p_i = 1$. Pour obtenir une réalisation x de loi μ, il suffit alors de « choisir » l'un des polygones P_j en simulant une réalisation j de loi discrète $\sum_{i=0}^{+\infty} p_i \, \delta_i$ sur $\mathbb{N}$, puis de simuler une réalisation (x, y) de loi uniforme sur P_i et de considérer son abscisse x. Les deux simulations nécessaires ne font appel qu'à la loi uniforme, et les p_i sont précalculés, ainsi que les sommets des polygones P_i.

Étudions de plus près le cas de la loi gaussienne standard. On observe tout d'abord que par symétrie, on peut se contenter de simuler la loi de densité

$$f(u) := \sqrt{\frac{\pi}{2}}\, e^{-u^2/2}\, \mathrm{I}_{[0,+\infty[}(u),$$

car le signe peut être obtenu par une loi de Bernoulli symétrique. Considérons à présent n points $x_1 < \cdots < x_n$. En prenant $x_1 = 0$ et $x_{n+1} = +\infty$, les points du plan de coordonnées $(x_i, f(x_i))$ constituent les coins inférieurs droits de n rectangles plats adjacents et horizontaux, qui recouvrent par excès l'aire sous la courbe de f. La hauteur du $k^{\text{ième}}$ rectangle vaut $f(x_k) - f(x_{k+1})$.

On choisit les x_i de telle sorte que la portion d'aire de f située dans chaque rectangle soit constante (et égale à $1/n$). Ensuite, l'algorithme consiste à choisir l'un des rectangles en simulant une loi uniforme sur les entiers $\{1, \ldots, n\}$. Une fois le rectangle choisi, on utilise :

- la méthode du rejet exacte pour le premier et le dernier rectangle, via l'expression de f;
- la méthode du rejet approximative pour les rectangles intermédiaires, qui consiste à utiliser la partie du rectangle sous la courbe de f, via le rapport $\sigma_k := x_{k-1}/x_k$, sans faire appel à l'expression de f.

Les x_i et les σ_i sont précalculés une fois pour toutes et stockés dans une table, et n est choisi suffisamment grand[7] pour que les taux de rejet soient faibles et les approximations bonnes. Les calculs sont très réduits une fois la phase d'initialisation passée.

1.6.4 Simulation d'un vecteur gaussien

Soit $m \in \mathbb{R}^n$ et $\Gamma \in \mathcal{S}_n^+(\mathbb{R})$ une matrice réelle de taille n, symétrique positive, et soit Y est un vecteur aléatoire gaussien standard de moyenne nulle et de matrice de covariance $\mathbf{Id}_n$, alors le vecteur aléatoire $X := \Gamma^{\frac{1}{2}} Y + m$ est un vecteur gaussien de moyenne m et de matrice de covariance Γ. Réciproquement, si X est un vecteur gaussien de loi $\mathcal{N}(m, \Gamma)$, alors $\Gamma^{-1/2}(X - m)$ est un vecteur aléatoire gaussien de loi $\mathcal{N}(0, \mathbf{Id}_n)$.

On peut également utiliser la décomposition de Cholesky de $\Gamma = AA^\top$ où A est une matrice triangulaire inférieure, en lieu et place de la décomposition en racine carrée $\Gamma = \Gamma^{1/2}\Gamma^{1/2}$, ce qui moins coûteux en terme de calculs (on évite de diagonaliser Γ et A est creuse).

Si Y est un vecteur gaussien de loi $\mathcal{N}(0, \mathbf{Id}_n)$, ses coordonnées sont i.i.d. de loi $\mathcal{N}(0, 1)$. Ainsi, on peut simuler une réalisation de Y en simulant n réalisations indépendantes $Y_1(\omega), \ldots, Y_n(\omega)$ de loi $\mathcal{N}(0, 1)$. Il suffit alors de calculer $\Gamma^{1/2} Y(\omega) + m$ pour obtenir une réalisation de loi $\mathcal{N}(m, \Gamma)$.

Voici un exemple de simulation de vecteur gaussien écrit en Matlab :

```
% Simulation d'une loi gaussienne multivariée
clear
% G = une matrice symétrique positive quelconque
X = rand(4,4);
G = X'*X/4
% m = un vecteur moyenne quelconque
m = rand(4,1)
% A = Racine carrée de G
A=sqrtm(G);
```

[7] Dans Matlab, $n = 128$.

```
%
n = 10000;
moyemp = zeros(4,1);
realis = zeros(4,n);
for i=1:n,
 realis(:,i) = m + A*randn(4,1);
 moyemp = moyemp + realis(:,i);
end
moyemp = moyemp/n;
% Variation relative par rapport à la moyenne théorique
rap_moy_relat=max(abs(moyemp-m))/norm(m)
% Calcul de la matrice de covariance empirique
covemp=zeros(4,4);
for i=1:4,
  for j=1:4,
        covemp(i,j)=dot(realis(i,:),realis(j,:))/n-moyemp(i)*moyemp(j);
  end
end
covemp
% Degrés de liberté par rapport à la matrice théorique
X=covemp/G % Faire 'help slash' pour en savoir plus sur cette division
dfs=trace(eye(4,4)-X)
% Réduction d'entropie
ent=-.5*log2(det(X))
```

1.7 Simulation de la loi uniforme sur les p-sphères

Nous savons que si (X, Y) est un vecteur gaussien de loi normale centrée réduite, et si (r, θ) désigne son écriture en coordonnées polaires : $(X, Y) = (r\cos(\theta), r\sin(\theta))$, alors r^2 et θ sont des v.a.r. indépendantes qui suivent respectivement une loi exponentielle de paramètre $1/2$ et une loi uniforme sur $[0, 2\pi]$. Le vecteur aléatoire $(X, Y)/r$ suit la loi uniforme sur le cercle unité. De façon générale, si $(X_1, \ldots, X_n)$ est un vecteur gaussien de loi $\mathcal{N}(0, \mathrm{I}_n)$ et

$$\|X\|_2 := \sqrt{X_1^2 + \cdots + X_n^2},$$

alors $(X_1/\|X\|_2, \ldots, X_n/\|X\|_2)$ et $\|X\|_2$ sont indépendants, $\|X\|_2^2$ suit une loi du χ^2 à n degrés de liberté, et $(X_1/\|X\|_2, \ldots, X_n/\|X\|_2)$ suit la loi uniforme sur la sphère euclidienne unité de $\mathbb{R}^n$. Attention, les v.a.r. $X_i/(\sqrt{n}\|X\|_2)$ ne suivent pas des lois de Student, car $\|X\|_2$ et X_i ne sont pas indépendantes !

Ce qui précède peut s'étendre à des normes non euclidiennes : pour tout $n \in \{2, 3, \ldots\}$, tout $p \in \mathbb{R}_+^*$ et tout $x \in \mathbb{R}^n$, on pose

$$\|x\|_p := (|x_1|^p + \cdots + |x_n|^p)^{1/p}.$$

On étend cette définition au cas $p = \infty$ en posant

$$\|x\|_\infty := \max(|x_1|, \ldots, |x_n|).$$

La p-sphère positive de $\mathbb{R}^n$, notée $\mathbb{S}(n, p)$, est définie par :

$$\mathbb{S}_+(n, p) := \left\{x \in \mathbb{R}_+^n,\ \|x\|_p = 1\right\}.$$

Pour $p = 1$, on obtient le « simplexe » standard, pour $p = 2$, la portion à coordonnées positives de la sphère euclidienne de rayon 1, et pour $p = \infty$, la surface du cube unité $[0, 1]^n$. Des dessins peuvent

aider à mieux se repérer. $\mathbb{S}_+(n,p)$ est donc une surface compacte de $\mathbb{R}^n$, de dimension $n-1$. Il serait possible de définir une mesure uniforme sur $\mathbb{S}_+(n,p)$, en considérant la trace de la mesure de Lebesgue sur $\mathbb{S}_+(n,p)$. Par soucis de simplicité, nous ne le feront pas ici, et nous nous restreignons volontairement aux cas $p \in \{1,2,\infty\}$.

En tant que surface du cube $[0,1]^n$, $\mathbb{S}_+(n,\infty)$ peut être vu comme la réunion disjointe, mais connexe, de $2n$ faces, chacune étant identifiable à $[0,1]^{n-1}$. La mesure uniforme sur $\mathbb{S}_+(n,\infty)$ peut donc être définie en considérant la mesure uniforme sur chaque face. Cette loi uniforme sur $\mathbb{S}_+(n,\infty)$ est très simple à simuler puisqu'il suffit de choisir la face uniformément sur $\{1,\dots,2n\}$, puis de générer une réalisation d'un vecteur aléatoire $(X_1,\dots,X_{n-1})$ dont les entrées sont i.i.d. et de loi uniforme sur $[0,1]$, obtenues par `rand` par exemple. Si l'on désigne par $(A,I) \in \{0,1\} \times \{1,\dots,n\}$ la face aléatoirement choisie, le vecteur aléatoire $(Y_1,\dots,Y_n)$ obtenu en insérant A en position I de la suite $X_1,\dots,X_{n-1}$ suit la loi uniforme sur $\mathbb{S}_+(n,\infty)$.

Pour $p=2$, la loi uniforme sur $\mathbb{S}_+(n,2)$ est la restriction normalisée de la mesure uniforme sur la sphère. Cette dernière se simule comme expliqué à la fin de la section 1.6, à partir d'une loi gaussienne. En fait, si $(X_1,\dots,X_n)$ est un vecteur aléatoire à composantes i.i.d. *positives* de loi de densité $\sqrt{2/\pi}\,\exp(-x^2/2)\,\mathrm{I}_{[0,+\infty[}\,dx$, alors le vecteur aléatoire $X/\sqrt{X_1^2+\cdots+X_n^2}$ suit la loi uniforme sur $\mathbb{S}_+(n,2)$.

Pour $p=1$, $\mathbb{S}_+(n,1)$ est un simplexe, qui est constitué de la partie à coordonnées positives de l'hyperplan $\mathbb{H}_n$ d'équation $x_1+\cdots+x_n=1$. Pour $n=2$, on obtient un segment de longueur $\sqrt{2}$, pour $n=3$, un triangle équilatéral de côté $\sqrt{2}$, pour $n=4$, un tétrahèdre régulier de côté $\sqrt{2}$, etc. Mentalement, nous effectuons naturellement une identification entre $\mathbb{H}_n$ et $\mathbb{R}^{n-1}$ lorsque n vaut 2 et 3, en identifiant par rotation de centre $(1,0,\dots,0)$ l'hyperplan $\mathbb{H}_n$ à $\mathbb{R}^{n-1}\times\{0\}$. En considérant la mesure Lebesgue λ_{n-1} sur $\mathbb{H}_n$ découlant de celle de $\mathbb{R}^{n-1}$ par cette identification, l'ensemble $\mathbb{S}_+(3,2)$ par exemple a pour mesure $\sqrt{3}/2$, qui est la surface du triangle équilatéral de côté $\sqrt{2}$ (souvenez-vous, « base fois hauteur sur 2 »). De façon plus générale, on montre que $\lambda_{n-1}(\mathbb{S}_+(n,1)) = \sqrt{n}/(n-1)!$. Il est naturel d'appeler loi uniforme sur $\mathbb{S}_+(n,1)$ la restriction de λ_{n-1} à $\mathbb{S}_+(n,1)$, normalisée en la divisant par $\lambda_{n-1}(\mathbb{S}_+(n,1))$. Comment simuler cette loi ? En d'autres termes, comment faire pour obtenir des réalisations i.i.d. d'un vecteur aléatoire $(X_1,\dots,X_n)$ dont la loi serait la loi uniforme sur $\mathbb{S}_+(n,1)$ que nous venons de décrire ?

On pourrait croire que si $X := (X_1,\dots,X_n)$ est un vecteur aléatoire de $\mathbb{R}^n$ à coordonnées i.i.d. et uniformes sur $[0,1]$, alors le vecteur aléatoire $X/(X_1+\cdots+X_n)$, qui est à valeurs dans $\mathbb{S}_+(n,1)$, suit la loi uniforme sur $\mathbb{S}_+(n,1)$. C'est faux malheureusement, comme on peut s'en convaincre déjà lorsque $n=2$.

La solution consiste à imiter la méthode gaussienne utilisée pour $p=2$. Si $X := (X_1,\dots,X_n)$ est un vecteur aléatoire à coordonnées i.i.d. de loi exponentielle de paramètre 1, alors le vecteur aléatoire $X/(X_1+\cdots+X_n)$ suit la loi uniforme sur $\mathbb{S}_+(n,1)$. La preuve n'est pas difficile, mais nous l'omettons ici.

De manière plus générale, pour tout $p \in \mathbb{R}_+$, on peut montrer que si $X := (X_1,\dots,X_n)$ est un vecteur aléatoire à composantes i.i.d. suivant une loi de densité

$$\frac{p}{\Gamma(p)}\,\exp(-t^p)\,\mathrm{I}_{\mathbb{R}_+}(t)$$

alors le vecteur aléatoire $X/\|X\|_p$ suit la loi uniforme sur $\mathbb{S}_+(n,p)$, et qu'il est de plus indépendant de la v.a.r. $\|X\|_p$.

Le cas $p=1$, pour lequel $\mathbb{S}_+(n,1)$ est un simplexe, est très important car $\mathbb{S}_+(n,1)$ représente l'ensemble des lois de probabilités discrètes sur $\{1,\dots,n\}$. Simuler la loi uniforme sur $\mathbb{S}_+(n,1)$ revient à choisir uniformément une loi de probabilité discrète sur $\{1,\dots,n\}$. La loi uniforme sur $\mathbb{S}_+(n,1)$ est un cas particulier de la loi de Dirichlet, cf. 9.4.6. Voici un exemple de fonction Matlab qui renvoie une réalisation de la loi de Dirichlet :


```
function rdir=rdirichlet(n,alpha);
  %RDIRICHLET renvoie une réalisation de la loi de Dirichlet de taille n
  %et de paramètre alpha. C'est la loi du vecteur aléatoire
  % (x_1/(x1+...+xn), ... , x_n/(x1+...+xn)) lorsque les xi sont i.i.d.
  %et suivent des lois gamma de paramètre respectifs (alpha(i),1).
  %Ainsi, lorsque alpha=(1,...,1), on obtient la loi uniforme sur le
  %simplexe des lois de probabilités discrètes sur {1,...,n}.
  %Le paramètre alpha doit être un vecteur de taille n à coordonnées
  %strictement positives. Il est optionnel est vaut par
  %défaut (1,...,1). La valeur de retour rdir est un vecteur ligne.
  %Les appels successifs donnent des réalisations pseudo-i.i.d. puisque
  %rdirichlet fait appel à rgamma pour simuler la loi gamma et à
  %rexpweib(.,1,1) si alpha est omis.
  %rdir=rdirichlet(n);
  if (nargin==1),
        % Loi expo, on peut aussi utiliser gamma(n,1)
        x=rexpweib(n,1,1);
        rdir=reshape(x/sum(x),[1,n]);
  else
        if (length(alpha)==~n), error('alpha doit etre de dim n'); end;
        if ~(all(alpha>0)), error('alpha doit etre a coords >0'); end;
        for i=1:n,
          x(i)=rgamma(1,alpha(i));
        end
        rdir=reshape(x/sum(x),[1,n]);
  end
  return;
```

Cette méthode permet également de choisir de manière « uniforme » la matrice de transition d'une chaîne de Markov, en utilisant des réalisations indépendantes de la loi uniforme sur le simplexe. Voici un exemple de code Matlab :

```
function P = rmatmar(n)
  %RMATMAR renvoie une matrice de Markov aléatoire de taille n x n
  %dont les lignes sont des probabilités discrètes choisies de façon
  %uniforme et indépendante en utilisant rdirichlet.
  %P = rmatmar(n);
  for i=1:n, P(i,:)=rdirichlet(n); end;
  return;
```

Bien entendu, tout repose sur le sens que l'on donne à la « loi uniforme » sur les classes d'objets considérés. Il y a une part d'arbitraire que l'on ne peut pas évacuer.

1.8 Un exemple de simulation d'une loi compliquée

On se propose de transposer en Matlab un petit exercice qui figure dans [Bou86, page 82]. La figure 1.4 représente un réseau électrique constitué de 7 éléments entre deux points A et B. Les durées de vie des éléments sont représentées par des variables aléatoires $T_1, \ldots, T_7$. La durée de vie du système est alors donnée par

$$T := T_7 \vee ((T_1 \vee T_2 \vee T_3) \wedge (T_6 \vee (T_4 \wedge T_5)))$$

où $\vee$ désigne le supremum et $\wedge$ l'infimum. On peut donc simuler T à partir d'une simulation de la loi jointe des T_i. On considère le cas où (T_1, T_2, T_3, T_4) sont indépendants de loi exponentielles de paramètres

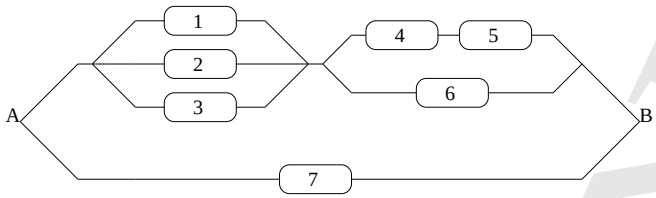

FIG. 1.4 – Schéma d'un réseau

respectifs $(1/2, 1/2, 1/2, 1/3)$ et que le couple (T_5, T_6) sont indépendantes des précédentes, de loi jointe de densité

$$\frac{1}{6} 1_{\{2 \leqslant x+y \leqslant 8\}} 1_{\{-1 \leqslant x-y \leqslant 1\}} dxdy,$$

et enfin, $T_7 = T_4$. Notons que le couple $((T_5 + T_6 - 2)/6, (T_5 - T_6 + 1)/2)$ suit une loi uniforme sur $[0,1]^2$, produit tensoriel de deux lois uniformes sur $[0,1]$ (donc indépendantes).

```
n=input('Taille des échantillons ?');
R=rand(n,6);                % n réal. indép. de loi unif sur [0,1]
T1=-2*log(R(:,1));          % n réal. indép. de T4
T2=-2*log(R(:,2));          % n réal. indép. de T4
T3=-2*log(R(:,3));          % n réal. indép. de T4
T4=-3*log(R(:,4));          % n réal. indép. de T4
T7=T4;                      % n réal. indép. de T7
T5=3*R(:,5)+R(:,6)+1/2;     % n réal. indép. de T5
T6=3*R(:,5)-R(:,6)+3/2;     % n réal. indép. de T6
% n réal. indép. de T
T=max(T7,min(max(T1,max(T2,T3)),max(T6,min(T4,T5))));
% moyenne et écart type empiriques de T
M=[mean(T),std(T)];
% histogramme de l'échantillon
hist(T,fix(sqrt(n)));
```

Intéressons nous à la durée de vie moyenne du système, qui n'est rien d'autre que $\mathbb{E}(T)$. Si $X_1, \ldots, X_n$ sont n v.a.r. i.i.d. de même loi que T, nous savons simuler des réalisations $X_1(\omega), \ldots, X_n(\omega)$. L'inégalité de Chebychev donne :

$$\mathbb{P}\left(\left| \frac{X_1 + \cdots + X_n}{n} - \mathbb{E}(T) \right| > \varepsilon \right) \leqslant \frac{\mathbf{Var}(T)}{n\varepsilon^2}.$$

La quantité $(X_1 + \cdots + X_n)/n$ représente la moyenne empirique que nous pouvons obtenir par simulation. Reste à majorer la variance $\mathbf{Var}(T)$. L'encadrement $T_7 \leqslant T \leqslant T_7 + T_1 + T_2 + T_3$ permet d'écrire :

$$\begin{aligned}
\mathbf{Var}(T) &= \mathbf{E}\left(T^2\right) - \mathbf{E}(T)^2 \\
&\leqslant \mathbb{E}((T_7 + T_1 + T_2 + T_3)^2) - \mathbb{E}(T_7)^2 \\
&= \mathbf{Var}(T_7 + T_1 + T_2 + T_3) + (\mathbb{E}(T_7) + \mathbb{E}(T_1) + \mathbb{E}(T_2) + \mathbb{E}(T_3))^2 - \mathbb{E}(T_7)^2 \\
&= \frac{1}{\lambda_1^2} + \frac{1}{\lambda_2^2} + \frac{1}{\lambda_3^2} + \frac{1}{\lambda_4^2} + \left(\frac{1}{\lambda_1} + \frac{1}{\lambda_2} + \frac{1}{\lambda_3} + \frac{1}{\lambda_4} \right)^2 - \frac{1}{\lambda_4^2} \\
&= 93.
\end{aligned}$$

On constate empiriquement que cette majoration de la variance est très mauvaise puisque la variance empirique est plutôt de l'ordre de quelques unités. Cependant, elle a le mérite d'exister. On peut donc

affirmer que l'on a

$$\mathbb{P}\left(\left|\frac{X_1+\cdots+X_n}{n}-\mathbb{E}(T)\right|>\frac{1}{10}\right)\leqslant\frac{1}{10},$$

pour $n\geqslant n_0:=93000$. Bien évidemment, ce seuil n_0 sur n sera d'autant meilleur (i.e. petit) que notre majoration de la variance $\mathbf{Var}(T)$ sera bonne.

Exercice 1.8.1. Supposons que T_7 suive une loi exponentielle indépendante des autres de paramètre θ. Écrire un programme Matlab qui détermine une valeur de θ telle que $\mathbb{E}(T_7)$ soit minimale et $\mathbb{E}(T)\geqslant 10$.

1.8.1 Rudiments de fiabilité

Soit T la date de première défaillance d'un système mis en marche à la date $t=0$, alors :
- La *durée de vie du système* est la variable T.
- La *fiabilité* à la date t est : $R(t):=\mathbb{P}(T\geqslant t)$.
- La *durée de survie* du système à l'instant t est la variable T_t de distribution :

$$\mathbb{P}(T_t\geqslant s)=\mathbb{P}(T-t\geqslant s\mid T\geqslant t)=\frac{R(t+s)}{R(t)}\text{ pour }s\geqslant 0.$$

- Le *taux de défaillance* ou *taux de panne* du système est la fonction :

$$h(t):=-\frac{R'(t)}{R(t)}.$$

- Le *temps moyen de panne* du système (Mean Time To Failure, MTTF) est : $m:=\mathbb{E}(T)$.

Soit un système à n composantes, de durée de vie respectives $T_1,\ldots,T_n$. On note T la durée de vie du système. Alors:
- Si les composants sont en série, $T=\min(T_1,\ldots,T_n)$ et $R(t)=R_1(t)R_2(t)\ldots R_n(t)$.
- Si les composants sont en parallèle, $T=\max(T_1,\ldots,T_n)$ et $R(t)=1-(1-R_1(t))(1-R_2(t))\cdots(1-R_n(t))$.

La loi exponentielle (de paramètre λ) est souvent utilisée pour modéliser une durée de vie T, sa fiabilité est $R(t)=e^{\lambda t}\,\mathrm{I}_{\mathbb{R}_+}(t)$. Sa durée de survie à l'instant t est indépendante de t, et l'on parle alors de *système sans mémoire* :

$$\mathbb{P}(T\geqslant t+s\,|\,T\geqslant t)=\mathbb{P}(T\geqslant s).$$

On peut montrer que la loi exponentielle est la seule loi non triviale à support dans $\mathbb{R}^+$ qui a cette propriété. La somme de n v.a. i.i.d. de loi exponentielle de paramètre λ est une loi Gamma de paramètres (λ,n), dite aussi loi de Erlang. Une autre loi utilisée pour modéliser les durées de vie est la de Weibull $W(a,b)$, définie à partir de sa fiabilité, qui vaut par définition $R(t)=1-F(t)=e^{-(bt)^a}\mathrm{I}_{\{t\geqslant 0\}}$. Elle modélise bien le comportement de a composants de loi exponentielle de paramètre b en parallèle pour t pas trop grand. La loi exponentielle de paramètre $\lambda>0$ correspond à $W(\lambda,1)$.

Remarque 1.8.2 *(Terminologie anglaise).* En anglais, la densité est souvent appelée « probability density function » (pdf), et la fonction de répartition « distribution function » ou encore « cumulative distribution function » (cdf). La connaissance de cette terminologie peut vous aider à la compréhension de l'aide des fonctions de Stixbox. Les noms des générateurs aléatoires de Stixbox commencent par la lettre `r` comme « random », les densités ou lois par `d` comme « density », les fonctions de répartitions inverses par `q` comme « quantile », les fonctions de répartitions par `p` comme « probability » et enfin les générateurs aléatoires via la méthode de rejet par `rj` comme « reject ».

Loi	Densité	F. de Rép.	F. R. Inverse	Générateur	Remarques
Beta	dbeta	pbeta	qbeta	rbeta	
Binômiale	dbinom	pbinom	qbinom	rbinom	Voir aussi rjbinom
Chi-2	dchisq	pchisq	qchiq	rchiq	
Fisher	df	pf	qf	rf	
Gamma	dgamma	pgamma	qgamma	rgamma	Voir aussi rjgamma
Géométrique				rgeom	
Hyper géom.	dhypg	phypg	qhypg	rhypg	
Normale	dnorm	pnorm	qnorm	rnorm	Voir aussi randn de Matlab
Poisson				rpoiss	Voir aussi rjpoiss
Student	dt	pt	qt	rt	
Weibull & exp.				rexpweib	
Kolmog.-Smir.		pks			

TAB. 1.1 – Générateurs aléatoires de la bibliothèque Stixbox.