Simulation : application au système bonus-malus en responsabilité civile automobile

Robert Langmeier

Travail de séminaire réalisé sous la supervision
du professeur François Dufresne

Ecole des HEC
Université de Lausanne

Avril 1998

TABLE DES MATIERES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

1 Introduction

La simulation de problèmes scientifiques sur les systèmes de traitement d'informations nécessite de pouvoir disposer de générateurs de nombres pseudo-aléatoires distribués uniformément ou selon d'autres lois de probabilité telles que Poisson ou Normale, par exemple.

Nous examinerons plusieurs méthodes pour obtenir ces distributions à partir de nombres aléatoires uniformément distribués, comme par exemple la méthode de l'inverse ou la méthode d'acceptation-rejet.

Dans le cas d'espèce qui nous occupe, nous appliquerons les techniques de simulation au système bonus-malus existant dans l'assurance responsabilité civile automobile pour estimer les distributions en régimes transitoire et stationnaire.

Pour la mise en œuvre de nos simulations, nous avons utilisé le langage de programmation APL qui permet de créer des programmes qui sont rapidement testés et opérationnels. Un autre atout non négligeable de ce langage est la puissance des opérateurs sur les vecteurs ou les matrices qui dans d'autres langages nécessiterait d'utiliser des bibliothèques de fonctions et qui alourdirait la programmation.

2 Nombres pseudo-aléatoires

Les simulations nécessitent l'utilisation de nombres aléatoires, que ce soit pour simuler des problèmes scientifiques ou tout simplement pour mettre en œuvre des jeux interactifs.

Il est par contre une question qui vient immédiatement à l'esprit, comment une machine qui ne peut ni raisonner, ni réfléchir, peut-elle réussir à tirer au sort un nombre alors qu'elle est "juste" capable d'exécuter des programmes selon une logique qu'on lui a fournie ?

La génération des nombres aléatoires sur un ordinateur n'a en fait rien d'aléatoire, c'est pour cela que l'on appelle ceux-ci pseudo-aléatoires. Ils sont générés de manière déterministe et ont l'apparence d'être distribués uniformément sur (0,1). La suite des nombres générés démarre une fois que le germe a été fourni. Ce germe peut être initialisé en fonction de la date et de l'heure de la machine pour que la suite soit différente à chaque exécution et donne ainsi l'illusion du hasard.

2.1 Algorithme "Congruentiel multiplicatif"

Une des approches des plus simples est l'algorithme "congruentiel multiplicatif", il implique l'utilisation d'une multiplication et du reste de la division entière (modulo).

La génération de la suite de nombres pseudo-aléatoires débute avec la valeur initiale x_0, appelée germe, qui doit être fournie par l'utilisateur. A partir d'un même germe, la suite des nombres sera identique. La suite des nombres aléatoires est exprimée par la formule de récurrence

$$x_n = ax_{n-1} \quad \text{modulo } m. \tag{1}$$

Il faut remarquer que x_0 doit être différent de 0, sinon la suite $\{x_n\}$ sera nulle. Pour donner l'illusion d'une distribution uniforme sur (0,1) on doit normaliser la valeur de x_n qui est distribuée sur (0,m) en effectuant une division en virgule flottante par la valeur m.

Pour une implémentation sur un processeur 32 bits, les valeurs $a = 7^5 = 16'807$ et $m = 2^{31} - 1 = 2'147'483'647$ donnent de bons résultats, tels que pour chaque germe, la suite de nombres aléatoires a l'apparence d'être uniformément distribuée et ne se répète qu'après un nombre important de tirages. La suite de nombres aléatoires commence à se répéter après $m - 1 = 2^{31} - 2$ tirages, c'est pourquoi l'on dit que ce générateur est de cycle complet.

2.2 Algorithme "Congruentiel mixte"

L'algorithme "congruentiel mixte" implique l'utilisation d'une multiplication, d'une addition et du reste de la division entière.

Contrairement à l'algorithme précédent, la valeur initiale x_0 de la suite des nombres aléatoires peut être nulle. Ceci est la conséquence de l'addition d'une constante c dans la formule de récurrence suivante

$$x_n = \left(ax_{n-1} + c\right) \text{modulo } m. \tag{2}$$

Pour ce type de générateur, on peut choisir m en fonction de la taille d'un mot du processeur, si cette taille est de 32 bits, on aura $m = 2^{32}$. Cette façon de faire rendra l'algorithme extrêmement performant, puisque la fonction sera assurée implicitement par le processeur.

Nous donnons un exemple d'implémentation de ces deux algorithmes ainsi qu'un exemple d'exécution dans les annexes **A** et **B**. Ces exemples ont étés programmés dans le langage C qui se prête bien à ce genre d'exemples.

Dans nos simulations, nous partons du principe que nous disposons d'un bon générateur de nombres aléatoires et le considérons simplement comme une boîte noire qui nous les fournit à volonté.

2.3 Utilisation des nombres aléatoires pour l'évaluation des intégrales

L'évaluation des intégrales était une des premières applications des nombres aléatoires. Soit $g(x)$ une fonction dont on veut évaluer θ où

$$\theta = \int_0^1 g(x)\,dx. \tag{3}$$

Pour évaluer la valeur de θ, comme U est uniformément distribuée sur (0,1) alors nous pouvons exprimer θ comme

$$\theta = E\big[g(U)\big]. \tag{4}$$

Si $U_1, \ldots, U_k$ sont des variables indépendantes distribuées uniformément sur (0,1), il s'ensuit que les variables aléatoires $g(U_1), \ldots, g(U_k)$ sont indépendantes et identiquement distribuées. La loi forte des grands nombres nous dit alors que

$$\sum_{i=1}^{k} \frac{g(U_i)}{k} \to E\big[g(U)\big] \quad \text{quand } k \to \infty. \tag{5}$$

Nous pouvons donc approximer θ en générant un grand nombre de nombres aléatoires et en prenant comme approximation la moyenne des $g(U_i), \quad i = 1,\ldots,k$.

Cette approche, pour approximer les intégrales, est appelée méthode de Monte-Carlo. Un exemple d'intégration écrit en APL est donné dans l'annexe **C** pour l'intégrale

$$\int_0^1 \sin(x)\,dx. \tag{6}$$

Si nous désirons intégrer une fonction dont les bornes d'intégration sont comprises entre a et b, avec $a \neq 0$ et $b \neq 1$, tel que

$$\theta = \int_a^b h(x)\,dx, \tag{7}$$

nous devons effectuer la substitution suivante

$$y = \frac{x-a}{b-a}, \tag{8}$$

d'où

$$x = a + y \cdot (b - a) \tag{9}$$

et

$$dy = \frac{dx}{b - a}. \tag{10}$$

D'où il s'ensuit que

$$\theta = \int_0^1 \frac{h\big(a + y \cdot (b - a)\big)}{b - a} dy. \tag{11}$$

3 Méthode de l'inverse

Nous désirons générer des nombres aléatoires non plus de manière uniforme, mais distribuée selon une loi particulière dont nous connaissons la fonction de répartition continue ou les masses de probabilité discrètes.

3.1 Distributions discrètes

Supposons que nous voulions générer les valeurs d'une variable aléatoire discrète X dont les masses de probabilité sont

$$P\{X = x_j\} = p_j, \quad j = 0, 1, \ldots, \sum_j p_j = 1. \tag{12}$$

Pour obtenir ce résultat, nous générons un nombre aléatoire U distribué uniformément sur (0,1) et posons

$$X = \begin{cases} \\ x_0 & \text{si } U < p_0 \\ x_1 & \text{si } p_0 \le U < p_0 + p_1 \\ \vdots & \\ x_j & \text{si } \sum_{i=1}^{j-1} p_i \le U < \sum_{i=1}^{j} p_i \\ \vdots & \end{cases} \tag{13}$$

Puisque, pour $0 < a < b < 1$, on a

$$P\{a \le U < b\} = b - a \tag{14}$$

il s'ensuit immédiatement que

$$P\{X = x_j\} = P\left\{\sum_{i=1}^{j-1} p_i \le U < \sum_{i=1}^{j} p_i\right\} = p_i\,. \tag{15}$$

Ainsi *X* aura la distribution souhaitée si on applique la transformation (13).

3.1.1 Génération d'une variable distribuée uniformément sur un support discret

Nous désirons générer une variable aléatoire *X* dont la distribution est uniforme, mais contrairement au générateur uniforme *U* dont les valeurs sont distribuées sur [0,1), le support de *X* est $1,...,n$. Ce type de générateur peut être utilisé, par exemple pour simuler un tirage de loterie à numéro ou tirer les cartes d'un jeu.

Nous pouvons donc écrire

$$P\{X = j\} = \frac{1}{n}, \quad j = 1, \ldots, n \tag{16}$$

d'où nous déduisons que

$$X = j \quad \text{si } \frac{j-1}{n} \le U < \frac{j}{n}. \tag{17}$$

Donc, *X* est égal à *j* si $j - 1 \le nU < j$ en d'autres termes,

$$X = \lfloor n \cdot U \rfloor + 1 \quad \text{avec } U \text{ distribuée uniformément sur [0,1)}, \tag{18}$$

où [*x*] est la partie entière de *x*.

3.1.2 Rappel sur la formule de récurrence de la famille a-b

Pour les lois Poisson, Binomiale, Binomiale Négative et Géométrique, nous pouvons trouver *a* et *b* tel que la formule de récurrence suivante est vérifiée

$$P\{N = n\} = \left(a + \frac{b}{n}\right) \cdot P\{N = n-1\} \qquad n = 1, 2, 3, \ldots \tag{19}$$

Les valeurs de *a* et *b* ainsi que de $P\{N=0\}$ sont récapitulées dans le tableau 1 et les fonctions de densité de probabilité se retrouvent dans le tableau 2.

Distribution	$P\{N=0\}$	a	b
Poisson(λ)	$e^{-\lambda}$	0	λ
Binomiale(n,p)	q^n	$-\frac{p}{q}$	$\frac{p}{q}(n+1)$
Binomiale Négative(r,p)	p^r	q	$q(r-1)$
Géométrique(p)	p	q	0

Tableau 1 : Paramètres de la famille a-b

Distribution	fdp
Poisson(λ)	$f(x)=\frac{e^{-\lambda}\cdot\lambda^x}{x!}$
Binomiale(n,p)	$f(x)=\binom{n}{x}p^x q^{n-x}$
Binomiale Négative(r,p)	$f(x)=\binom{x+r-1}{x}p^r q^x$
Géométrique(p)	$f(x)=p\cdot q^x$

Tableau 2 : Fonctions de densité de probabilité de la famille a-b

3.1.3 Générer une variable aléatoire de Poisson

La clé pour utiliser la transformation inverse afin de générer une variable aléatoire distribuée selon la loi de Poisson réside dans l'utilisation de la formule de récurrence que nous avons vue au point 3.1.2. Nous avons que

$$P\{N=n\}=\frac{\lambda}{n}\cdot P\{N=n-1\},\qquad n=1,2,3,\ldots \tag{20}$$

et

$$P\{N=0\}=e^{-\lambda}. \tag{21}$$

Afin de faciliter l'implémentation de l'algorithme, le pseudo-code est donné ci-dessous et permet sa mise en oeuvre dans le langage utilisé par le lecteur. Un exemple de mise en oeuvre en APL est donné dans l'annexe **D** pour la transformation inverse d'une loi de Poisson.

Pseudo-code

1. Générer un nombre aléatoire uniforme *U*.
2. Poser $n = 1$, $p = e^{-\lambda}$, $F = p$.
3. Si $U < F$ alors poser $X = n - 1$ et arrêter.
4. Poser $p = \frac{\lambda}{n} \cdot p$, $F = F + p$ et incrémenter *n*.
5. Continuer au point 3.

Il est très facile d'étendre le même algorithme en faisant les adaptations nécessaires pour les autres lois de probabilités de la famille a-b.

3.2 Distributions continues

La transformation inverse d'une distribution continue suppose que l'on connaisse sa fonction de répartition.

Si *U* est une variable aléatoire distribuée uniformément sur (0,1), alors pour une fonction de répartition *F* quelconque nous proposons de définir la variable aléatoire *X* par

$$X = F^{-1}(U). \tag{22}$$

Preuve

$$\begin{aligned} F(x) &= P\{X \le x\}, \\ &= P\{F^{-1}(U) \le x\}. \end{aligned} \tag{23}$$

Puisque *F* est une fonction de répartition, il découle que $F(x)$ est une fonction monotone croissante de *x*, alors l'inégalité $a \le b$ est équivalente à $F(a) \le F(b)$.

$$\begin{aligned} F(x) &= P\{F(F^{-1}(U)) \le F(x)\}, \\ &= P\{U \le F(x)\} \quad \text{puisque } F(F^{-1}(x)) = U, \\ &= F(x) \quad\quad\quad \text{puisque } U \text{ est distribué uniformément sur } (0,1). \end{aligned} \tag{24}$$

3.2.1 Distribution exponentielle

La distribution exponentielle est un cas particulier, car il est possible de trouver l'inverse analytiquement. Soit la fonction de répartition

$$F(x) = 1 - e^{-\beta x}, \quad x > 0. \tag{25}$$

Si nous posons $x = F^{-1}(u)$, alors

$$u = F(x) = 1 - e^{-\beta x} \tag{26}$$

donc

$$1 - u = e^{-\beta x}. \tag{27}$$

Si nous prenons le logarithme des deux cotés, nous obtenons

$$-\beta x = \log(1-u). \tag{28}$$

Nous pouvons maintenant isoler *x* qui est égal à

$$x = -\frac{\log(1-u)}{\beta}. \tag{29}$$

Il est donc possible de générer une variable aléatoire exponentielle de paramètre β en générant un nombre aléatoire *U* et en posant

$$X = F^{-1}(U) = -\frac{\log(1-U)}{\beta}. \tag{30}$$

Etant donné que $1-U$ est distribuée uniformément sur (0,1), on a que *U* est également distribuée uniformément sur (0,1), donc nous pouvons écrire que

$$X = -\frac{\log(U)}{\beta}. \tag{31}$$

Cette transformation permet de gagner une opération de calcul, ce qui en soit peut paraître insignifiant mais, si on utilise un nombre important de nombres aléatoires cela peut permettre de diminuer la durée des simulations.

3.2.2 Distribution gamma pour α entier

Supposons que nous désirons générer une variable aléatoire Gamma($\alpha = n; \beta$) avec $\alpha > 1$ entier et $\beta > 0$. La fonction de répartition d'une telle distribution est donnée par

$$F(x) = \int_0^x \frac{\beta \cdot e^{-\beta y} \cdot (\beta y)^{n-1}}{(n-1)!} dy. \tag{32}$$

Pour cette distribution, il n'est pas possible d'exprimer la fonction inverse analytiquement. Par contre, en utilisant le résultat qu'une variable aléatoire *X* ayant une distribution Gamma($\alpha = n; \beta$) peut être vue comme la somme de *n* exponentielles indépendantes de paramètre β et, vu le résultat obtenu au point 3.2.1, nous pouvons écrire

$$\begin{aligned} X &= -\frac{\log(U_1)}{\beta} - \frac{\log(U_2)}{\beta} - \ldots - \frac{\log(U_n)}{\beta}, \\ &= -\frac{1}{\beta}\log(U_1 \cdot U_2 \cdots U_n). \end{aligned} \tag{33}$$

L'utilisation de l'identité $\sum_{i=1}^{n}\log(x_i) = \log(x_1 \cdot x_2 \cdots x_n)$ permet d'économiser du temps de calcul, car au lieu de *n* logarithmes, un seul est requis.

4 Méthode d'acceptation-rejet

La technique d'acceptation-rejet est puissante pour générer une variable aléatoire dont la fonction de densité de probabilité est connue et calculable mais pas sa fonction de répartition contrairement à la méthode de l'inverse.

4.1 Modèle discret

Supposons que nous avons une méthode efficiente pour simuler une variable aléatoire dont les masses de probabilité sont $\{q_j,\ j \geq 0\}$. Nous pouvons alors l'utiliser comme base pour simuler la distribution dont les masses de probabilité sont $\{p_j,\ j \geq 0\}$ en simulant en premier lieu une variable aléatoire *Y* avec masses de probabilité $\{q_j\}$ et en acceptant cette valeur avec une probabilité proportionnelle à $\frac{p_Y}{q_Y}$.

Cette proportion est exprimée plus précisément par *c*, une constante telle que

$$\frac{p_j}{q_j} \leq c \quad \text{pour tous les } j \text{ tel que } p_j \geq 0. \tag{34}$$

Nous avons maintenant la technique suivante, appelée méthode d'acceptation-rejet, pour simuler une variable aléatoire *X* dont les masses de probabilité sont $p_j = P\{X = j\}$.

Pseudo-code

1. Simuler la valeur de *Y* distribuée selon les masses de probabilité q_j.
2. Générer un nombre aléatoire *U*.
3. Si $U \le \frac{p_Y}{cq_Y}$ alors poser *X* = *Y* et arrêter sinon aller à l'étape 1.

L'organigramme de cette méthode est représenté dans la figure 1 ci-dessous.

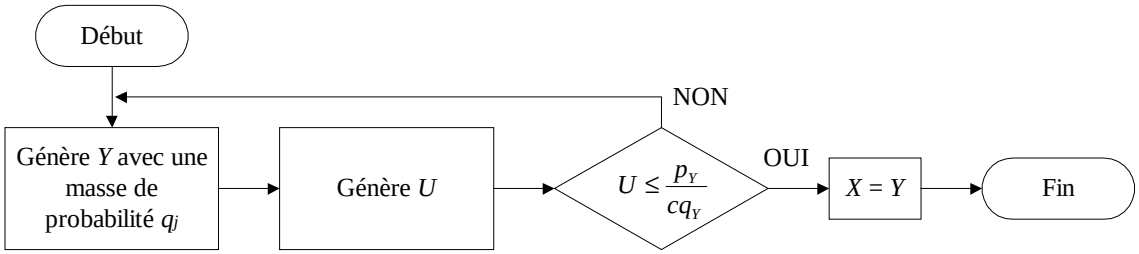

Figure 1 : Organigramme de la méthode d'acceptation-rejet discrète

Théorème

L'algorithme d'acceptation-rejet génère une variable aléatoire *X* telle que

$$P\{X = j\} = p_j, \quad j = 0,1,... \tag{35}$$

De plus, le nombre d'itérations nécessaires pour obtenir la variable aléatoire *X* est distribuée selon une loi géométrique de paramètre *c*.

Preuve

Pour commencer, nous devons déterminer la probabilité qu'une seule itération produise l'acceptation pour la valeur de *j*.

$$\begin{aligned} P\{Y = j, \text{est accepté}\} &= P\{Y = j\} \cdot P\{\text{est accepté} \mid Y = j\}, \\ &= q_j \frac{p_j}{cq_j}, \\ &= \frac{p_j}{c}. \end{aligned} \tag{36}$$

En effectuant la somme sur toutes les réalisations possibles de *j*, nous obtenons la probabilité que la variable aléatoire soit acceptée

$$P\{\text{accepté}\} = \sum_j \frac{p_j}{c} = \frac{1}{c}. \tag{37}$$

Comme les itérations sont indépendantes et que chacune d'entre elles résulte en l'acceptation de la valeur avec une probabilité $\frac{1}{c}$, nous pouvons voir que le nombre d'itérations nécessaires est distribué selon une loi géométrique de paramètre *c*. Nous pouvons donc écrire

$$\begin{aligned} P\{X = j\} &= \sum_n P\{j \text{ accepté à l'itération } n\}, \\ &= \sum_n \left(1-\frac{1}{c}\right)^{n-1} \frac{p_j}{c}, \\ &= p_j . \end{aligned} \tag{38}$$

4.1.1 Exemple

Supposons que nous désirons simuler la valeur d'une variable aléatoire *X* dont les réalisations et leurs probabilités respectives sont

x	1	2	3	4	5	6	7	8	9	10
p(x)	0.11	0.12	0.09	0.08	0.12	0.10	0.09	0.09	0.10	0.10

La première possibilité serait d'utiliser la technique de l'inverse que nous avons vue au point 3.1. L'autre possibilité est d'utiliser la méthode d'acceptation-rejet avec les masses de probabilité *q* distribuées uniformément sur 1, . . . , 10. Nous avons donc

$$q_j = \frac{1}{10}, \quad j = 1,..,10.$$

En fonction de ce choix pour $\{q_j\}$, il s'ensuit que *c* est égal à

$$c = \max \frac{p_j}{q_j} = 1.2.$$

L'algorithme pour cet exemple est le suivant

Pseudo-code

1. Générer un nombre aléatoire uniforme U_1 et poser $Y = \text{Entier}(10U_1) + 1$.
2. Générer un second nombre aléatoire U_2.
3. Si $U_2 \le \frac{p_Y}{0.12}$ alors poser *X* = *Y* et arrêter sinon aller à l'étape 1.

La constante 0.12 dans l'étape 3 provient de $c \cdot q_Y = \frac{1.2}{10} = 0.12$.

En moyenne, cet algorithme nécessite seulement 1.2 itérations pour obtenir la génération de la variable aléatoire *X*.

4.2 Modèle continu

Nous supposons en premier lieu que nous sommes capables de générer une variable aléatoire ayant comme fonction de densité de probabilité $g(x)$. Nous pouvons alors l'utiliser comme point de départ pour générer la fonction de densité de probabilité $f(x)$ en générant *Y* à partir de *g* et en acceptant la valeur de Y avec une probabilité proportionnelle à $\frac{f(Y)}{g(Y)}$.

Cette proportion est exprimée par *c*, une constante telle que

$$\frac{f(y)}{g(y)} \leq c \quad \text{pour tous les } y. \tag{39}$$

Le pseudo-code ci-dessous résume la technique de génération d'une variable aléatoire avec une fonction de densité de probabilité *f*.

Pseudo-code

1. Simuler la valeur de *Y* avec comme fonction de densité de probabilité *g*.
2. Générer un nombre aléatoire *U*.
3. Si $U \leq \frac{f(Y)}{cg(Y)}$ alors poser *X* = *Y* et arrêter sinon aller à l'étape 1.

L'organigramme de cette méthode est représenté dans la figure 2 ci-dessous.

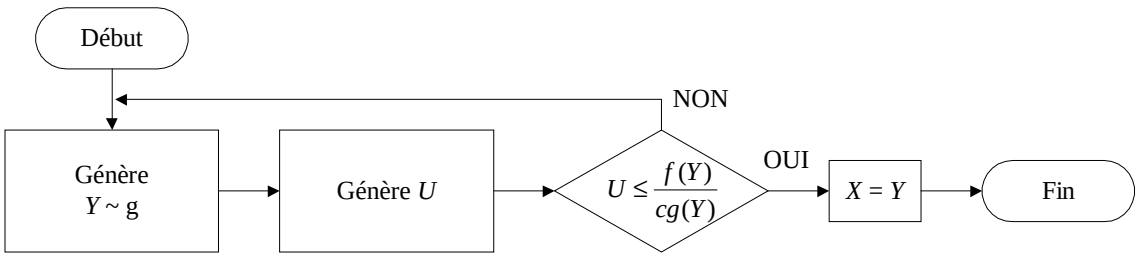

Figure 2 : Organigramme de la méthode d'acceptation-rejet continue

Le lecteur notera que la méthode d'acceptation-rejet continue est exactement la même que pour le modèle discret, les masses de probabilité étant remplacées par les fonctions

de densité de probabilité. Il s'ensuit que la démonstration du modèle discret est également valable pour le modèle continu.

4.2.1 Distribution Normale

Nous désirons générer une variable aléatoire *Z* ayant une distribution Normale standard (c'est-à-dire de moyenne $\mu = 0$ et de variance $\sigma^2 = 1$). En premier lieu, il faut noter que la fonction de densité de probabilité de la valeur absolue de *Z* est

$$f(x) = \frac{2}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}, \quad 0 < x < \infty. \tag{40}$$

Pour générer un nombre aléatoire distribué selon une loi Normale standard par la méthode d'acceptation-rejet, nous commençons par poser la fonction *g* comme étant une fonction de densité de probabilité exponentielle de moyenne 1, c'est-à-dire

$$g(x) = e^{-x}, \quad 0 < x < \infty. \tag{41}$$

Il s'ensuit que le rapport de *f* sur *g* devient

$$\frac{f(x)}{g(x)} = \sqrt{\frac{2}{\pi}} e^{x - \frac{x^2}{2}} \tag{42}$$

et que la valeur maximale de $\frac{f(x)}{g(x)}$ se produit pour la valeur de *x* qui maximise $x - \frac{x^2}{2}$. En résolvant, on constate que cela se produit pour $x = 1$. Nous pouvons donc déterminer notre constante *c* qui sera égale à

$$c = \max \frac{f(x)}{g(x)} = \frac{f(1)}{g(1)} = \sqrt{\frac{2e}{\pi}}. \tag{43}$$

Puisque

$$\begin{aligned} \frac{f(x)}{cg(x)} &= \exp\left\{ x - \frac{x^2}{2} - \frac{1}{2} \right\}, \\ &= \exp\left\{ -\frac{(x-1)^2}{2} \right\}, \end{aligned} \tag{44}$$

il s'ensuit que nous pouvons générer la valeur absolue d'une variable aléatoire Normale selon l'algorithme indiqué dans le pseudo-code ci-dessous.

Pseudo-code

1. Générer *Y*, une variable aléatoire exponentielle de paramètre 1.
2. Générer un nombre aléatoire U_1.
3. Si $U_1 \leq \exp\left\{\frac{-(Y-1)^2}{2}\right\}$ alors poser *X* = *Y* et arrêter sinon aller à l'étape 1.
4. Générer un nombre aléatoire U_2 et poser

$$Z = \begin{cases} X & \text{si } U_2 \leq \frac{1}{2}, \\ -X & \text{si } U_2 > \frac{1}{2}. \end{cases}$$

Lorsque la variable aléatoire *X* est générée, elle a une distribution selon la fonction de densité de probabilité définie à l'équation (40). Celle-ci est donc distribuée comme la valeur absolue d'une variable aléatoire Normale standard. Nous pouvons obtenir la variable aléatoire *Z* en laissant celle-ci prendre soit la valeur de *X*, soit la valeur de -*X* et ce de façon équiprobable.

Un exemple de programmation de l'algorithme décrit ci-dessus est donné dans l'annexe **E**.

Dans l'étape 3, la valeur de *Y* est acceptée si

$$U \leq \exp\left\{\frac{-(Y-1)^2}{2}\right\}, \tag{45}$$

ce qui est équivalent à

$$-\log U \geq \frac{(Y-1)^2}{2}. \tag{46}$$

Or on a vu au point 3.2.1 que $-\log U$ est une variable aléatoire exponentielle de paramètre 1, le pseudo-code devient donc

Pseudo-code

1. Générer 2 exponentielles indépendantes de paramètre 1, soit Y_1 et Y_2.
2. Si $Y_2 \geq \frac{(Y_1 - 1)^2}{2}$ alors poser $X = Y_1$ et arrêter sinon aller à l'étape 1.
3. Générer un nombre aléatoire U et poser

$$Z = \begin{cases} X & \text{si } U \leq \frac{1}{2}, \\ -X & \text{si } U > \frac{1}{2}. \end{cases}$$

Cet algorithme de génération d'une nombre aléatoire distribué selon une loi Normale standard est d'une très bonne efficacité, puisque le nombre moyen d'itérations, qui est donné par c, est égal à $c = \sqrt{\frac{2e}{\pi}} \cong 1.32$.

5 Méthode polaire pour générer une variable aléatoire distribuée selon une loi Normale

Dans cette section, nous allons voir comment nous pouvons générer des variables aléatoires distribuées selon une loi Normale en partant de nombres aléatoires distribués uniformément.

Soit X et Y deux variables aléatoires Normales de moyenne $\mu = 0$ et de variance $\sigma^2 = 1$. Si R et θ représentent les coordonnées polaires du vecteur (X,Y) alors

$$R^2 = X^2 + Y^2 \tag{47}$$

et

$$\tan\theta = \frac{Y}{X}. \tag{48}$$

Comme *X* et *Y* sont indépendantes, la distribution conjointe de leurs fonctions de densité de probabilité est donnée par

$$\begin{aligned} f(x,y) &= \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}}, \\ &= \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}}, \qquad -\infty < x, y < \infty. \end{aligned} \tag{49}$$

Pour déterminer la distribution conjointe de R^2 et de θ, nous effectuons le changement de variables suivant

$$d = x^2 + y^2 \tag{50}$$

et

$$\theta = \tan^{-1}\left(\frac{x}{y}\right). \tag{51}$$

Le Jacobien de cette transformation est égal à

$$\begin{vmatrix} \frac{\partial d}{\partial x} & \frac{\partial d}{\partial y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} \end{vmatrix} = \begin{vmatrix} 2x & 2y \\ \frac{-y}{x^2+y^2} & \frac{x}{x^2+y^2} \end{vmatrix} = 2. \tag{52}$$

D'où

$$f(d,\theta) = \frac{1}{2} \cdot \frac{1}{2\pi} e^{-\frac{d}{2}}, \; 0 < d < \infty, \; 0 < \theta < 2\pi. \tag{53}$$

Ce résultat est égal au produit d'une variable aléatoire exponentielle de moyenne 2 et d'une variable aléatoire uniforme distribuée sur $(0,2\pi)$, il s'ensuit que

> R^2 et θ sont indépendantes, avec R^2 ayant une distribution exponentielle de moyenne 2 et θ ayant une distribution uniforme sur $(0,2\pi)$. (54)

En utilisant ce résultat, nous pouvons générer une paire de nombres aléatoires standard *X* et *Y*. En premier lieu, nous générons les coordonnées polaires à partir de nombres aléatoires uniformes et les transformons ensuite en coordonnées rectangulaires. Le pseudo-code qui permet d'obtenir cette transformation est donné ci-dessous.

Pseudo-code

1. Générer deux nombres aléatoires uniformes U_1 et U_2.
2. Poser $R^2 = -2\log U_1$ et $\theta = 2\pi U_2$.
3. Poser $X = R\cos\theta = \sqrt{-2\log U_1}\cos(2\pi U_2)$.
4. Poser $Y = R\sin\theta = \sqrt{-2\log U_1}\sin(2\pi U_2)$.

Les transformations données aux étapes 3 et 4 sont connues sous le nom de transformations de Box-Muller. Un exemple de mise en oeuvre en APL est donné dans l'annexe **F** ainsi qu'un exemple d'exécution.

Malheureusement, la transformation de Box-Muller est consommatrice en temps d'exécution et cela vient des opérations trigonométriques entre particulier. Il est possible de déterminer les valeurs du sinus et du cosinus de manière indirecte pour un angle aléatoire.

Comme U est uniformément distribué sur (0,1) alors $2U$ est uniforme sur (0,2) et donc $2U-1$ est uniforme sur (-1,1). Donc, si nous générons les nombres aléatoires uniformes U_1 et U_2, nous pouvons poser

$$V_1 = 2U_1 - 1 \tag{55}$$

et

$$V_2 = 2U_2 - 1. \tag{56}$$

Nous obtenons donc que (V_1, V_2) est uniformément distribué dans un carré d'une surface totale de 4 centré en (0, 0). Supposons maintenant que nous générons des paires (V_1, V_2) jusqu'à ce qu'une d'entre elles soit contenue dans un cercle de rayon 1 et centré en (0, 0) ou, autrement dit, que la paire (V_1, V_2) satisfait la condition $V_1^2 + V_2^2 \le 1$. Nous disposons maintenant d'une paire (V_1, V_2) qui est uniformément distribuée dans le cercle.

Si R et θ représentent les coordonnées polaires de cette paire, il est alors aisé de vérifier que R et θ sont indépendantes avec R^2 distribuée uniformément sur (0,1) et avec θ distribuée uniformément sur $(0,2\pi)$. Comme θ est un angle aléatoire, il s'ensuit que nous pouvons générer le sinus et le cosinus de l'angle aléatoire θ en générant un point aléatoire (V_1, V_2) dans le cercle et nous pouvons poser

$$\sin\theta = \frac{V_2}{R} = \frac{V_2}{\sqrt{V_1^2 + V_2^2}} \tag{57}$$

et

$$\cos\theta = \frac{V_1}{R} = \frac{V_1}{\sqrt{V_1^2 + V_2^2}}. \tag{58}$$

De la transformation de Box-Muller, il s'ensuit que nous pouvons générer une paire de nombres aléatoires distribués selon la loi Normale en générant un nombre aléatoire *U* et en posant

$$X = \sqrt{-2\log U}\,\frac{V_1}{\sqrt{V_1^2 + V_2^2}} \tag{59}$$

et

$$Y = \sqrt{-2\log U}\,\frac{V_2}{\sqrt{V_1^2 + V_2^2}}. \tag{60}$$

Comme $R^2 = V_1^2 + V_2^2$ est également distribuée uniformément sur (0, 1) et est indépendante de la variable aléatoire θ, nous pouvons l'utiliser en lieu et place du nombre aléatoire *U*. En posant $S = R^2$, nous obtenons que

$$X = \sqrt{-2\log S}\,\frac{V_1}{\sqrt{S}} = V_1\sqrt{\frac{-2\log S}{S}} \tag{61}$$

et

$$Y = \sqrt{-2\log S}\,\frac{V_2}{\sqrt{S}} = V_2\sqrt{\frac{-2\log S}{S}}. \tag{62}$$

Ces deux variables aléatoires Normales standard sont indépendantes lorsque (V_1, V_2) est un point choisi aléatoirement dans un cercle de rayon 1 centré à l'origine et que $S = V_1^2 + V_2^2$.

Le pseudo-code qui permet d'obtenir deux nombres aléatoires distribués selon une loi Normale standard est donné ci-dessous.

Pseudo-code

1. Générer deux nombres aléatoires uniformes U_1 et U_2.
2. Poser $V_1 = 2U_1 - 1$, $V_2 = 2U_2 - 1$ et $S = V_1^2 + V_2^2$.
3. Si $S > 1$ alors retour à l'étape 1.
4. Poser $X = V_1\sqrt{\frac{-2\log S}{S}}$.
5. Poser $Y = V_2\sqrt{\frac{-2\log S}{S}}$.

L'algorithme examiné ci-dessus est appelé méthode polaire. Un exemple de mise en oeuvre en APL est donné dans l'annexe **G** ainsi qu'un exemple d'exécution.

Comme la génération des nombres aléatoires V_1 et V_2 est soumise à la contrainte que $S = V_1^2 + V_2^2 \leq 1$, la probabilité d'accepter cette valeur de S est égale au rapport de l'aire d'un cercle de rayon 1 à l'aire d'un carré dont les côtés sont de longueur 2 (la définition de S suggère de centrer ces figures à l'origine). Cette probabilité est égale à $\frac{\pi}{4} \cong 0.785$, le nombre moyen d'itérations est donc égal à l'inverse, soit 1.273.

6 Buts et mécanismes des systèmes bonus-malus

On examinera le mécanisme appliqué de manière uniforme en Suisse par toutes les compagnies d'assurances automobiles avant le 1er janvier 1996.

Depuis cette date, le marché a été libéralisé et chaque compagnie a adopté des stratégies différentes en classant les conducteurs en fonction du sexe, de l'âge, du type de voiture, de la cylindrée, de la nationalité, etc. Chaque compagnie peut également avoir des échelles de primes différentes, l'entrée dans le système ne se faisant pas forcément à 100% de la prime base.

Le système bonus-malus est basé sur les chaînes de Markov dont les différents états sont décrits dans le tableau 3 ainsi que les pourcentages de la prime de base. Ce tableau est valable pour une voiture de tourisme dont la cylindrée est comprise entre 1393 et 2963 centimètres cubes. L'entrée dans le système par un nouvel automobiliste se fait à l'état 9 et il paie donc de ce fait la prime de base pour la première année d'assurance.

Etat	Prime	Etat	Prime	Etat	Prime	Etat	Prime
0	45%	6	75%	12	130%	18	215%
1	50%	7	80%	13	140%	19	230%
2	55%	8	90%	14	155%	20	250%
3	60%	9	100%	15	170%	21	270%
4	65%	10	110%	16	185%		
5	70%	11	120%	17	200%		

Tableau 3 : Echelle des primes

Les règles de transitions suivantes s'appliquent pour toutes les années subséquentes, si on identifie l'état de l'année précédente par *x* le nouvel état sera :

$\max(x-1;0)$	si aucun sinistre n'est survenu pendant l'année,
$\min(x+n\cdot s;21)$	si *n* sinistres sont annoncés à l'assureur.

Dans le modèle utilisé en Suisse entre 1990 et 1996, chaque sinistre nous fait avancer le compteur d'état d'une valeur $s = 4$.

Nous remarquons également qu'un assuré qui est déjà au bénéfice du bonus maximal le restera l'année suivante pour autant qu'il n'ait aucun sinistre dans l'année en cours. De même l'assuré qui paie la prime la plus élevée restera à ce niveau de prime s'il cause un nouveau sinistre pendant l'année, à moins que la compagnie ne l'exclut suite à ce sinistre, ce dont nous ne tiendrons pas compte dans le modèle théorique.

7 Estimation des distributions en régimes transitoire et stationnaire grâce aux simulations

Dans cette section, nous allons voir que nous pouvons obtenir la distribution stationnaire du système bonus-malus de manière récursive sans faire appel à la théorie des chaînes de Markov.

Nous avons un système bonus-malus avec $n+1=22$ états compris entre 0 et 21. Nous désignons par X_t son état au temps *t*, $t=0,1,2,\ldots$, et $X_0 = x_0$ son état initial qui dans notre système vaut 9. Nous pouvons écrire l'état au temps $t+1$ comme

$$X_{t+1} = \begin{cases} 0, & \text{si } X_t + Y_{t+1} = -1, \\ X_t + Y_{t+1}, & \text{si } 0 \le X_t + Y_{t+1} \le n, \\ n & \text{si } X_t + Y_{t+1} > n. \end{cases} \tag{63}$$

avec la variation de l'état définie comme

$$Y_{t+1} = \begin{cases} s \cdot N_{t+1} & \text{si } N_{t+1} \geq 1, \\ -1 & \text{si } N_{t+1} = 0. \end{cases} \tag{64}$$

avec $s = 4$ et

$$N_{t+1} \sim \text{Poisson}(\lambda) \qquad \text{nombre d'accidents.} \tag{65}$$

Statistiquement le conducteur européen a un accident en moyenne tous les 10 ans, cela nous amène à prendre $\lambda = 0.10$ dans notre simulation. Il est à noter qu'en Suisse, cette fréquence est actuellement d'un accident tous les 14 ans, ce qui donne $\lambda = 0.07$.

Nous effectuerons la simulation 100'000 fois, comme s'il s'agissait de plusieurs assurés. On peut également imaginer qu'il s'agit du même assuré pour lequel on effectue 100'000 simulations.

Pseudo-code

1. Initialisation de la matrice de résultat *z* à 0.
2. Pour *k* = 1 à 100000 par pas de 1 répéter
    3. Poser *x* = 9
    4. Poser $z[x;0] = z[x;0] + 1$
    5. Pour t = 1 à 1000 par pas de 1 répéter
        6. Poser *n* = InversePoisson(0.10)
        7. Poser *y* = -1 si *n* = 0 et $y = 4 \cdot n$ sinon
        8. Poser $x = \min(\max(x + y;0);21)$
        9. Poser $z[x;t] = z[x;t] + 1$
    10. Fin pour
11. Fin pour

Dans le pseudo-code décrit ci-dessus, *z* représente la matrice des résultats. Il vient immédiatement que $z[9;0]$ doit contenir, dans notre cas, la valeur 100'000. La répartition dans les autres colonnes dépend de la simulation, mais le total de chaque colonne doit être égal à 100'000 par définition.

En divisant toutes les valeurs par 100'000, on obtient la fonction de densité de probabilité correspondante, la fonction de répartition est obtenue en effectuant une somme progressive dans chacune des colonnes de la matrice et, par définition, nous aurons $F(21,t) = 1$, $t = 0,1,2,...,1000$.

La distribution stationnaire $F(x)$ est donnée par

$$F(x) = \lim_{t \to \infty} F(x,t). \tag{66}$$

Dans notre simulation, il est évident que nous ne pouvons pas faire tendre le temps jusqu'à l'infini, c'est pour cela que nous nous sommes volontairement arrêtés au temps $t = 1000$ alors qu'une valeur de $t = 100$ aurait été suffisante. On voit par contre clairement qu'entre cette borne minimale et celle que nous avons utilisée, la distribution stagne et apparaît comme étant stationnaire.

Dans l'annexe **H** et **I** figurent respectivement le listing du programme de simulation et le détail des simulations. Un extrait des résultats figure dans le tableau 4 ci-dessous.

Etat	temps									
	0	1	2	9	10	20	30	50	100	1000
0	0.0000	0.0000	0.0000	0.4073	0.3674	0.5088	0.5428	0.5575	0.5594	0.5606
1	0.0000	0.0000	0.0000	0.4073	0.3674	0.5439	0.5926	0.6133	0.6180	0.6180
2	0.0000	0.0000	0.0000	0.4073	0.3674	0.5815	0.6466	0.6752	0.6825	0.6832
3	0.0000	0.0000	0.0000	0.4073	0.3674	0.6361	0.7160	0.7479	0.7544	0.7539
4	0.0000	0.0000	0.0000	0.4073	0.7353	0.8033	0.8221	0.8304	0.8344	0.8325
5	0.0000	0.0000	0.0000	0.7727	0.7353	0.8208	0.8473	0.8614	0.8657	0.8643
6	0.0000	0.0000	0.0000	0.7727	0.7353	0.8358	0.8695	0.8897	0.8947	0.8939
7	0.0000	0.0000	0.8210	0.7727	0.7353	0.8480	0.8895	0.9151	0.9203	0.9199
8	0.0000	0.9045	0.8210	0.7727	0.7534	0.8748	0.9162	0.9383	0.9420	0.9428
9	1.0000	0.9045	0.8210	0.7906	0.9188	0.9388	0.9484	0.9540	0.9551	0.9557
10	1.0000	0.9045	0.8210	0.9364	0.9188	0.9436	0.9562	0.9642	0.9654	0.9661
11	1.0000	0.9045	0.8210	0.9364	0.9188	0.9469	0.9626	0.9729	0.9740	0.9748
12	1.0000	0.9045	0.9827	0.9364	0.9194	0.9505	0.9681	0.9792	0.9811	0.9813
13	1.0000	0.9952	0.9827	0.9370	0.9366	0.9627	0.9769	0.9847	0.9860	0.9861
14	1.0000	0.9952	0.9827	0.9520	0.9812	0.9839	0.9870	0.9889	0.9898	0.9900
15	1.0000	0.9952	0.9827	0.9867	0.9815	0.9854	0.9894	0.9918	0.9924	0.9928
16	1.0000	0.9952	0.9910	0.9870	0.9821	0.9870	0.9914	0.9940	0.9948	0.9950
17	1.0000	0.9999	0.9988	0.9877	0.9841	0.9893	0.9935	0.9956	0.9965	0.9964
18	1.0000	0.9999	0.9988	0.9896	0.9905	0.9940	0.9962	0.9973	0.9979	0.9975
19	1.0000	0.9999	0.9988	0.9945	0.9975	0.9976	0.9985	0.9986	0.9988	0.9986
20	1.0000	0.9999	0.9990	0.9989	0.9986	0.9987	0.9994	0.9993	0.9996	0.9994
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Tableau 4 : Distributions transitoires et stationnaires de la simulation

Dans notre simulation, on constate que 56% des assurés se retrouvent dans l'état 0 dans le cas de la distribution stationnaire que cela soit après 100 ou 1000 unités de temps.

La détermination de la classe moyenne pour chaque unité de temps est donnée par

$$E[X_t] = \sum_{x=0}^{21} x \cdot f(x,t). \tag{67}$$

où $f(x,t)$ est la fonction de densité de probabilité de la distribution transitoire et *x* est la classe concernée.

Pour la détermination de la prime moyenne par unité de temps, on procède par analogie avec la formule précédente, $b(x)$ donnant le taux de prime de la classe *x*, donc la formule est

$$E\left[b_{X_t}\right] = \sum_{x=0}^{21} b(x) \cdot f(x,t). \tag{68}$$

En appliquant ces deux formules sur la fonction de densité de probabilité, nous avons calculé la classe et la prime moyenne en fonction du temps. Un extrait des résultats obtenus est donné dans le tableau 5.

Temps	0	1	2	3	4	5	6	7	8	9	10	20	50	100	1000
Classe	9.00	8.50	7.98	7.47	6.97	6.47	5.96	5.46	4.95	4.44	4.31	2.87	2.15	2.10	2.10
Prime	1.00	0.95	0.90	0.89	0.87	0.84	0.81	0.78	0.76	0.73	0.72	0.63	0.58	0.57	0.57

Tableau 5 : Classe et prime moyenne

On voit que la prime moyenne diminue avec le temps. Si la totalité des primes encaissées tend à devenir trop bas par rapport à l'évolution du coût total des sinistres, la compagnie d'assurance sera obligée de réajuster la prime de base à la hausse.

8 Conclusion

Dans le cadre de ce travail, nous avons eu l'occasion d'examiner différentes méthodes pour obtenir des nombres aléatoires distribués selon les lois de Poisson, Exponentielle, Normale et Gamma.

La pierre angulaire pour pouvoir utiliser ces techniques est de disposer d'un générateur de nombres uniformes de bonne qualité, ce que nous avons considéré comme acquis dans le cadre de l'utilisation du langage de programmation APL PLUS III sous Windows. Néanmoins, ce point est très important, car il conditionne la qualité des simulations qui peuvent être obtenues. Il est à noter qu'il existe des ouvrages traitant des générateurs uniformes de manière très détaillée.

Les méthodes de génération de nombres distribués de manière non uniforme ont chacune leurs qualités et leurs défauts. La méthode de l'inverse a comme principal défaut la nécessité de connaître l'inverse de la fonction de répartition pour la distribution souhaitée, ce qui n'est pas toujours possible. La méthode d'acceptation-rejet nécessite quant à elle de déterminer une fonction de comparaison, ce qui peut se révéler plus ou moins difficile. De plus, ces deux algorithmes, à l'exception notoire de l'exponentielle, nécessitent de faire une ou plusieurs itérations pour obtenir le résultat souhaité.

Pour le cas particulier de la loi Normale, nous avons eu l'occasion d'examiner une méthode basée sur la transformation du référentiel, puisque nous passons des coordonnées polaires aux coordonnées rectangulaires. Cet algorithme ne nécessite pas de boucle si l'on utilise les fonctions trigonométriques sinus et cosinus. Une variante de cet algorithme effectue une ou plusieurs itérations pour remplacer les fonctions trigonométriques.

A partir de ce travail de base, nous nous sommes intéressés à utiliser ces connaissances dans le cadre du sujet de ce travail de séminaire, à savoir la simulation des distributions en régimes transitoire et stationnaire du système bonus-malus tel qu'il a été appliqué en Suisse de manière uniforme entre 1990 et 1996. Nous avons pu constater que 56% des assurés se retrouvent dans la classe 0 après 100 ou 1000 ans, durée qui représente la distribution stationnaire.

9 Références

- Dufresne, F.(1995) *The Efficiency of the Swiss Bonus-malus System*, Bulletin de l'Association Suisse des Actuaires 1, p. 29-42
- Dufresne, F (1988) *Distributions stationnaires d'un système bonus-malus et probabilité de ruine*, ASTIN Bulletin 18, p. 31-46
- Lemaire, J. (1985) *Automobile Insurance*, Kluwer, 248 pp.
- Ross, S.M. (1997) *Simulation*, Academic Press, 282 pp.

10 Annexes

A Listing du programme générateur de nombres aléatoires uniformes en langage C

```
/*    Générateurs de nombres pseudo-aléatoires
      Congruentiel multiplicatif et Congruentiel mixte     */

#include "stdio.h"
#include "math.h"


#define     rlRndCongrMult()  rlrnd(&lwRndCongrMult,0)
#define     rlRndCongrMultF() rlrndf(&lwRndCongrMult,0)

#define     rlRndCongrMixt()  rlrnd(&lwRndCongrMixt,c_)
#define     rlRndCongrMixtF() rlrndf(&lwRndCongrMixt,c_)


#define     a_                16807L
#define     m_                2147483647L
#define     c_                123456L


static unsigned long    lwRndCongrMult;
static unsigned long    lwRndCongrMixt;
static unsigned long    lwA;
static unsigned long    lwM;


/*    Cette fonction implémente l'algorithme
      congruentiel multiplicatif si lwC = 0 et
      congruentiel mixte si lwC != 0                       */

long  rlrnd (unsigned long    *lwRnd,
             unsigned long     lwC)
{
      return *lwRnd = (lwA * *lwRnd + lwC) % lwM;
}


/*    Cette fonction retourne le nombre aléatoire distribué
      sur (0,1) en divisant la valeur retournée par la
      fonction précédente par lwM                   */

double      rlrndf      (unsigned long       *lwRnd,
                         unsigned long lwC)
{
      return (double) rlrnd(lwRnd,lwC) / lwM;
}


/*    Test des algorithmes                          */

int   main  (int  argc,
             char  *argv[])
{
unsigned long     c, p, mc;
int         n;
double            r1, r2, s1, s2, ss1, ss2,
            moy1, moy2, var1, var2;

      printf("Nombre de boucles exprimé en puissance de 10 : ");
      scanf("%lu",&n);
      printf("Initialisation des générateurs (1-%ld) : ",m_ - 1);
      scanf("%lu",&lwRndCongrMult);
      lwRndCongrMixt = lwRndCongrMult;
      lwA = a_;
      lwM = m_;
      p = 10;
      s1 = s2 = 0;
      ss1 = ss2 = 0;
```

```
      mc = pow10(n);

      printf("Test des algorithmes congruentiel multiplicatif et mixte\n\n");
      printf("Germe: %10ld\n\n",lwRndCongrMult);
      printf("                Congruentiel multiplicatif  |          Congruentiel mixte     \n");
      printf("   tirages      somme       moyenne    var. |      somme       moyenne    var.\n");
      printf("-----------------------------------------------+--------------------------------\n");
      for(c = 1; c <= mc; c++)
      {
            r1 = rlRndCongrMultF();
            r2 = rlRndCongrMixtF();
            s1 += r1;
            s2 += r2;
            ss1 += r1 * r1;
            ss2 += r2 * r2;
            if (c == p)
            {
                  moy1 = s1 / c;
                  moy2 = s2 / c;
                  var1 = c / (c - 1) * (ss1 / c - moy1 * moy1);
                  var2 = c / (c - 1) * (ss2 / c - moy2 * moy2);
                  printf("%10ld %16.5f %7.5f %7.5f | %16.5f %7.5f
%7.5f\n",c,s1,moy1,var1,s2,moy2,var2);
                  p *= 10;
            }
      }
}
```

B Exécution du programme générateur de nombres aléatoires uniformes

```
Nombre de boucles exprimé en puissance de 10 : 8
Initialisation des générateurs (1-2147483646) : 1234567
Test des algorithmes congruentiel multiplicatif et mixte

Germe:    1234567

                Congruentiel multiplicatif  |          Congruentiel mixte
   tirages      somme       moyenne    var. |      somme       moyenne    var.
-----------------------------------------------+--------------------------------
        10          4.94160 0.49416 0.08831 |          4.58373 0.45837 0.06636
       100         50.24398 0.50244 0.08355 |         46.95508 0.46955 0.07491
      1000        502.46619 0.50247 0.08396 |        497.82485 0.49782 0.08134
     10000       4989.45113 0.49895 0.08443 |       4996.48545 0.49965 0.08297
    100000      50309.88417 0.50310 0.08348 |      50226.87750 0.50227 0.08368
   1000000     504499.47266 0.50450 0.08361 |     503399.73299 0.50340 0.08380
  10000000    5046459.06512 0.50465 0.08363 |    5035451.26589 0.50355 0.08381
 100000000   50466031.33565 0.50466 0.08363 |   50355823.66940 0.50356 0.08381
```

C Intégration par la méthode de Monte Carlo

```
j ExempleMonteCarlo n; approx; exact; loop

'Calcul de l''intégrale de Sin(x) entre 0 et 1'
'Valeur exacte : ', 8 6 ⍕ exact ← 1 - 2○1

'Valeur approximée par ',n,' nombres aléatoires'
'itér.   approximation     erreur   erreur en %'
:FOR loop :IN 1+⍳j
    approx ← MonteCarloSinus n
    5 0 14 6 14 6 10 4 ⍕ loop,approx,approx - exact,((approx - exact) ÷ exact) ×
100
:ENDFOR

z ← MonteCarloSinus n

z ← (+/1○((?n⍴1E6)÷1E6)) ÷ n

10 ExempleMonteCarlo 100000
Calcul de l'intégrale de Sin(x) entre 0 et 1
Valeur exacte : 0.459698
Valeur approximée par  100000  nombres aléatoires
itér.   approximation     erreur   erreur en %
    1      0.458571     ¯0.001127    0.7036
    2      0.461269      0.001572    0.1194
    3      0.459625     ¯0.000073    0.4755
    4      0.460091      0.000393    0.3746
    5      0.459262     ¯0.000436    0.5540
    6      0.458548     ¯0.001150    0.7087
    7      0.460109      0.000411    0.3706
    8      0.460278      0.000581    0.3340
    9      0.460744      0.001046    0.2332
   10      0.460696      0.000999    0.2435
```

D Transformation inverse selon une loi de Poisson

```
z ← U InvPoisson lambda; n; p; F

n ← 1
F ← p ← ⋆-lambda
loop:
    →(U < F)/found
    F ← F + p ← p × lambda ÷ n
    n ← n + 1
→loop

found:
    z ← n - 1

((?1E3)÷1E3) InvPoisson 10
6
```

E Génération de variables aléatoires Normales selon la méthode d'acceptation-rejet

```
z ← beta Expon n; u

⍝ Génère n exponentielle

u ← ((?(n)⍴¯2+1E6)+1)÷1E6
z ← (⍟u)÷-beta
```

```
z ← NormaleAcceptRejet n; y; u1; u2

⍝ Calcule des nombres aléatoires distribués selon une loi Normale standard

z ← ⍴0
:FOR k :IN ⍳n
    encore:
        y ← 1 Expon 1
        u1 ← ((?¯2+1E6)+1)÷1E6
    →((u1>⋆(¯1+y)⋆2)÷2)/encore
    u2 ← ((?¯2+1E6)+1)÷1E6
    z ← z,((u2≤0.5),(u2>0.5))/y,-y
:ENDFOR

⍝ Calcul de la moyenne de 20000 nombres aléatoires

(+/NormaleAcceptRejet 20000) ÷ 20000
¯0.001235707886
```

F Génération de variables aléatoires Normales selon la méthode de Box-Muller

```
z ← BoxMuller n; u; r2; theta

z ← (2×n)⍴0
u ← ((?(2×n)⍴¯2+1E6)+1)÷1E6
r2 ← ¯2×⍟u[2×⍳n]
theta ← ○2×u[1+2×⍳n]
z[2×⍳n] ← (r2⋆0.5)×2○theta
z[1+2×⍳n] ← (r2⋆0.5)×1○theta

⍝ Calcul de la moyenne de 20000 nombres aléatoires

(+/BoxMuller 10000) ÷ 20000
0.004730451507
```

G Génération de variables aléatoires Normales selon la méthode polaire

```
z ← Polar

encore:
    v ← ¯1 + 2×((?2⍴¯2+1E6)+1)÷1E6
    s ← +/v⋆2
→(s>1)/encore

z ← v × ((¯2×⍟s)÷s)⋆0.5

Polar
¯1.199224806 ¯0.1320628463
```

H Listing du programme de simulation

```
z ← par SimulationBonusMalus K ;T;C;C0;L;s;k;t;x;n;vn;d

⍝ Extraction des paramètres de simulation

T ←  par[0] + 1
C ←  par[1] + 1
C0 ← par[2]
s ←  par[3]
L ←  par[4]

⍝ Diverses initialisations

d ← 1E13        ⍝ étendue des nombres uniformes
z ← (C,T)⍴0     ⍝ matrice des résultats

⍝ Initialisation du vecteur de correspondance accidents <-> recul ou avance
⍝ avec un réserve pour .9999999999999 InvPoisson 0.10 qui donne 8

vn ← s × ⍳20
vn[0] ← ¯1      ⍝ Dans le cas ou il n'y a pas d'accident, on recul d'un état

⍝ Simulation pour k assurés ou pour k situations d'un assuré

:FOR k :IN ⍳K

⍝ Etat initial de l'assuré au temps t=0 (prime de base)

    x ← C0
    z[x;0] ← z[x;0]+1

⍝ Simulation pour les années subséquentes

    :FOR t :IN 1+⍳T-1
        n ← ((?1E13)÷1E13) InvPoisson L     ⍝ Nombre d'accidents
        x ← ((x + vn[n])⌊21)⌈0
        z[x;t] ← z[x;t]+1
    :ENDFOR
:ENDFOR

ExempleSimulationBonusMalus ;K;T

⍝ Exécution de la simulation sur 1'000 unités de temps et
⍝ pour 100'000 simulations

K ← 10000
T ← 1000

m ← (T,21,9,4,0.10) SimulationBonusMalus K

fdp ← m ÷ K     ⍝ On calcule la fonction de densité de probabilité
fr ← +⍀fdp      ⍝ On calcule la fonction de répartition

PrintBonusMalus ;o

⍝ Affiche les tableaux de résultats des fonctions de répartitions
⍝ et prépare un extrait pour Word

:FOR o :IN 10 × ⍳10
    '     Temps'
    'Etat', 7 0 ⍕ o+⍳10
    3 0 9 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 ⍕ (⍳22),fr[;o+⍳10]
:ENDFOR
```

```
'     Temps'
'Etat', 7 0 ⍕ 100×1+⍳10
3 0 9 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 7 4 ⍕ (⍳22),fr[;100×1+⍳10]

o ← 0,1,2,9,10,20,30,50,100,1000
fr_word ← (¯1+⍳23),o⍪fr[;o]

PrimeEtClasseMoyenne ;premium; o

⍝ Calcule la prime et la classe moyenne et prépare un extrait pour Word

clmoy ← +⌿(⍉((1001,22)⍴(⍳22))) × fdp
premium ←
.45,.5,.55,.6,.65,.7,.75,.8,.9,1,1.1,1.2,1.3,1.4,1.55,1.7,1.85,2,2.15,2.3,2.5,2.7
prmoy ← +⌿(⍉((1001,22)⍴premium)) × fdp

o ← (⍳11),20,50,100,1000
clprmoy_word ←(⍉o,[0.5]clmoy[o])⍪prmoy[o]
```

I Détail des distributions transitoires simulées

	Temps									
Etat	0	1	2	3	4	5	6	7	8	9
0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.4073
1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.4496	0.4073
2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.4970	0.4496	0.4073
3	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.5495	0.4970	0.4496	0.4073
4	0.0000	0.0000	0.0000	0.0000	0.0000	0.6080	0.5495	0.4970	0.4496	0.4073
5	0.0000	0.0000	0.0000	0.0000	0.6724	0.6080	0.5495	0.4970	0.4496	0.7727
6	0.0000	0.0000	0.0000	0.7434	0.6724	0.6080	0.5495	0.4970	0.8086	0.7727
7	0.0000	0.0000	0.8210	0.7434	0.6724	0.6080	0.5495	0.8427	0.8086	0.7727
8	0.0000	0.9045	0.8210	0.7434	0.6724	0.6080	0.8773	0.8427	0.8086	0.7727
9	1.0000	0.9045	0.8210	0.7434	0.6724	0.9088	0.8773	0.8427	0.8086	0.7906
10	1.0000	0.9045	0.8210	0.7434	0.9385	0.9088	0.8773	0.8427	0.8265	0.9364
11	1.0000	0.9045	0.8210	0.9636	0.9385	0.9088	0.8773	0.8604	0.9524	0.9364
12	1.0000	0.9045	0.9827	0.9636	0.9385	0.9088	0.8945	0.9654	0.9524	0.9364
13	1.0000	0.9952	0.9827	0.9636	0.9385	0.9245	0.9773	0.9654	0.9524	0.9370
14	1.0000	0.9952	0.9827	0.9636	0.9523	0.9861	0.9773	0.9654	0.9530	0.9520
15	1.0000	0.9952	0.9827	0.9751	0.9920	0.9861	0.9773	0.9659	0.9660	0.9867
16	1.0000	0.9952	0.9910	0.9962	0.9920	0.9861	0.9778	0.9770	0.9913	0.9870
17	1.0000	0.9999	0.9988	0.9962	0.9920	0.9865	0.9862	0.9945	0.9917	0.9877
18	1.0000	0.9999	0.9988	0.9962	0.9924	0.9927	0.9969	0.9950	0.9924	0.9896
19	1.0000	0.9999	0.9988	0.9966	0.9969	0.9986	0.9973	0.9957	0.9943	0.9945
20	1.0000	0.9999	0.9990	0.9991	0.9994	0.9991	0.9982	0.9976	0.9978	0.9989
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	10	11	12	13	14	15	16	17	18	19
0	0.3674	0.3311	0.2998	0.2715	0.4925	0.4698	0.4454	0.4222	0.4070	0.5259
1	0.3674	0.3311	0.2998	0.5443	0.5182	0.4917	0.4656	0.4497	0.5811	0.5631
2	0.3674	0.3311	0.6016	0.5724	0.5425	0.5140	0.4957	0.6420	0.6223	0.6018
3	0.3674	0.6641	0.6327	0.5996	0.5670	0.5471	0.7084	0.6880	0.6652	0.6433
4	0.7353	0.6986	0.6627	0.6263	0.6035	0.7821	0.7593	0.7357	0.7110	0.7033
5	0.7353	0.6986	0.6627	0.6398	0.8150	0.7918	0.7684	0.7442	0.7366	0.8352
6	0.7353	0.6986	0.6775	0.8455	0.8236	0.8000	0.7756	0.7696	0.8650	0.8507
7	0.7353	0.7150	0.8746	0.8524	0.8300	0.8057	0.8013	0.8917	0.8780	0.8634
8	0.7534	0.8999	0.8792	0.8565	0.8340	0.8307	0.9147	0.9011	0.8873	0.8726
9	0.9188	0.8999	0.8792	0.8570	0.8555	0.9301	0.9180	0.9044	0.8910	0.8938
10	0.9188	0.8999	0.8797	0.8780	0.9443	0.9327	0.9204	0.9076	0.9109	0.9507
11	0.9188	0.9005	0.9000	0.9563	0.9460	0.9342	0.9230	0.9263	0.9610	0.9543
12	0.9194	0.9193	0.9664	0.9573	0.9470	0.9364	0.9397	0.9698	0.9634	0.9566
13	0.9366	0.9743	0.9667	0.9578	0.9487	0.9513	0.9766	0.9713	0.9650	0.9595
14	0.9812	0.9746	0.9672	0.9597	0.9622	0.9822	0.9777	0.9725	0.9677	0.9703
15	0.9815	0.9752	0.9691	0.9712	0.9869	0.9832	0.9789	0.9750	0.9771	0.9880
16	0.9821	0.9772	0.9788	0.9904	0.9879	0.9843	0.9812	0.9828	0.9908	0.9893
17	0.9841	0.9852	0.9934	0.9913	0.9890	0.9865	0.9878	0.9934	0.9920	0.9907
18	0.9905	0.9957	0.9944	0.9925	0.9912	0.9920	0.9958	0.9945	0.9933	0.9929
19	0.9975	0.9967	0.9957	0.9945	0.9954	0.9976	0.9968	0.9958	0.9953	0.9967
20	0.9986	0.9980	0.9977	0.9978	0.9990	0.9985	0.9983	0.9977	0.9985	0.9990
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	20	21	22	23	24	25	26	27	28	29
0	0.5088	0.4923	0.4763	0.4711	0.5387	0.5298	0.5202	0.5091	0.5090	0.5480
1	0.5439	0.5263	0.5209	0.5954	0.5848	0.5741	0.5638	0.5624	0.6047	0.5981
2	0.5815	0.5756	0.6581	0.6463	0.6337	0.6216	0.6228	0.6679	0.6600	0.6532
3	0.6361	0.7273	0.7141	0.7003	0.6864	0.6868	0.7393	0.7289	0.7209	0.7126
4	0.8033	0.7887	0.7736	0.7586	0.7585	0.8151	0.8066	0.7959	0.7869	0.7896
5	0.8208	0.8056	0.7900	0.7910	0.8472	0.8372	0.8276	0.8176	0.8217	0.8540
6	0.8358	0.8202	0.8213	0.8767	0.8667	0.8560	0.8469	0.8506	0.8837	0.8768
7	0.8480	0.8495	0.9027	0.8930	0.8826	0.8734	0.8774	0.9092	0.9032	0.8961
8	0.8748	0.9244	0.9157	0.9055	0.8962	0.9006	0.9308	0.9249	0.9185	0.9122
9	0.9387	0.9306	0.9213	0.9123	0.9160	0.9444	0.9388	0.9330	0.9271	0.9308
10	0.9436	0.9350	0.9271	0.9305	0.9553	0.9508	0.9456	0.9406	0.9438	0.9594
11	0.9469	0.9396	0.9432	0.9653	0.9604	0.9560	0.9519	0.9553	0.9687	0.9656
12	0.9504	0.9540	0.9735	0.9692	0.9643	0.9609	0.9644	0.9761	0.9734	0.9705
13	0.9627	0.9792	0.9758	0.9716	0.9680	0.9712	0.9815	0.9792	0.9767	0.9745
14	0.9839	0.9810	0.9779	0.9747	0.9772	0.9859	0.9840	0.9818	0.9802	0.9821
15	0.9854	0.9828	0.9805	0.9827	0.9893	0.9879	0.9860	0.9846	0.9866	0.9903
16	0.9870	0.9852	0.9874	0.9921	0.9910	0.9895	0.9884	0.9899	0.9931	0.9922
17	0.9893	0.9910	0.9943	0.9935	0.9924	0.9916	0.9927	0.9951	0.9946	0.9937
18	0.9940	0.9963	0.9956	0.9949	0.9942	0.9952	0.9966	0.9963	0.9960	0.9956
19	0.9976	0.9974	0.9970	0.9965	0.9971	0.9980	0.9977	0.9975	0.9975	0.9978
20	0.9987	0.9987	0.9984	0.9987	0.9991	0.9989	0.9988	0.9989	0.9990	0.9993
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	30	31	32	33	34	35	36	37	38	39
0	0.5428	0.5370	0.5293	0.5298	0.5504	0.5446	0.5406	0.5386	0.5412	0.5558
1	0.5926	0.5859	0.5862	0.6083	0.6034	0.5977	0.5936	0.5978	0.6142	0.6104
2	0.6466	0.6489	0.6726	0.6663	0.6622	0.6564	0.6590	0.6783	0.6747	0.6702
3	0.7160	0.7446	0.7366	0.7313	0.7268	0.7289	0.7480	0.7454	0.7405	0.7363
4	0.8221	0.8155	0.8083	0.8028	0.8068	0.8269	0.8224	0.8180	0.8139	0.8174
5	0.8473	0.8405	0.8336	0.8380	0.8591	0.8544	0.8500	0.8455	0.8492	0.8610
6	0.8695	0.8635	0.8666	0.8882	0.8836	0.8792	0.8754	0.8786	0.8904	0.8881
7	0.8895	0.8937	0.9133	0.9101	0.9051	0.9015	0.9051	0.9163	0.9139	0.9107
8	0.9162	0.9354	0.9312	0.9272	0.9232	0.9269	0.9377	0.9354	0.9328	0.9305
9	0.9484	0.9450	0.9405	0.9373	0.9405	0.9507	0.9485	0.9461	0.9442	0.9460
10	0.9562	0.9527	0.9491	0.9528	0.9614	0.9598	0.9574	0.9558	0.9578	0.9630
11	0.9626	0.9596	0.9624	0.9704	0.9686	0.9668	0.9652	0.9674	0.9720	0.9704
12	0.9681	0.9706	0.9772	0.9756	0.9742	0.9726	0.9748	0.9790	0.9775	0.9763
13	0.9769	0.9826	0.9808	0.9797	0.9784	0.9801	0.9842	0.9829	0.9817	0.9808
14	0.9870	0.9854	0.9841	0.9833	0.9847	0.9881	0.9872	0.9864	0.9853	0.9865
15	0.9894	0.9881	0.9871	0.9886	0.9912	0.9904	0.9901	0.9892	0.9897	0.9915
16	0.9914	0.9906	0.9914	0.9938	0.9930	0.9924	0.9922	0.9926	0.9938	0.9934
17	0.9935	0.9941	0.9956	0.9954	0.9947	0.9943	0.9950	0.9958	0.9954	0.9951
18	0.9962	0.9972	0.9969	0.9966	0.9962	0.9967	0.9974	0.9970	0.9966	0.9965
19	0.9985	0.9983	0.9979	0.9979	0.9980	0.9985	0.9984	0.9979	0.9978	0.9981
20	0.9994	0.9991	0.9990	0.9992	0.9993	0.9993	0.9992	0.9990	0.9991	0.9993
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	40	41	42	43	44	45	46	47	48	49
0	0.5526	0.5486	0.5452	0.5487	0.5559	0.5543	0.5531	0.5511	0.5533	0.5585
1	0.6066	0.6036	0.6059	0.6152	0.6126	0.6112	0.6087	0.6105	0.6170	0.6162
2	0.6667	0.6706	0.6794	0.6777	0.6754	0.6727	0.6745	0.6810	0.6807	0.6781
3	0.7403	0.7521	0.7486	0.7469	0.7435	0.7457	0.7527	0.7516	0.7491	0.7470
4	0.8304	0.8292	0.8252	0.8222	0.8240	0.8315	0.8308	0.8274	0.8252	0.8276
5	0.8599	0.8578	0.8538	0.8554	0.8634	0.8620	0.8601	0.8576	0.8591	0.8631
6	0.8860	0.8840	0.8846	0.8915	0.8913	0.8890	0.8875	0.8892	0.8921	0.8915
7	0.9096	0.9116	0.9172	0.9162	0.9148	0.9133	0.9157	0.9187	0.9169	0.9164
8	0.9330	0.9391	0.9374	0.9357	0.9347	0.9368	0.9404	0.9388	0.9379	0.9369
9	0.9523	0.9507	0.9489	0.9477	0.9497	0.9533	0.9519	0.9508	0.9501	0.9517
10	0.9616	0.9603	0.9588	0.9606	0.9640	0.9632	0.9621	0.9610	0.9628	0.9650
11	0.9694	0.9681	0.9697	0.9729	0.9720	0.9714	0.9705	0.9714	0.9738	0.9729
12	0.9755	0.9770	0.9797	0.9789	0.9783	0.9775	0.9786	0.9802	0.9800	0.9794
13	0.9822	0.9848	0.9838	0.9832	0.9828	0.9835	0.9851	0.9846	0.9845	0.9840
14	0.9885	0.9879	0.9874	0.9869	0.9877	0.9889	0.9887	0.9883	0.9882	0.9885
15	0.9911	0.9906	0.9902	0.9910	0.9920	0.9918	0.9917	0.9914	0.9917	0.9919
16	0.9932	0.9929	0.9935	0.9943	0.9941	0.9940	0.9940	0.9941	0.9943	0.9941
17	0.9949	0.9954	0.9959	0.9959	0.9957	0.9959	0.9960	0.9962	0.9959	0.9959
18	0.9970	0.9974	0.9973	0.9973	0.9973	0.9975	0.9976	0.9974	0.9973	0.9972
19	0.9985	0.9984	0.9983	0.9985	0.9986	0.9986	0.9986	0.9985	0.9983	0.9986
20	0.9993	0.9992	0.9993	0.9994	0.9994	0.9994	0.9994	0.9992	0.9994	0.9995
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	50	51	52	53	54	55	56	57	58	59
0	0.5575	0.5539	0.5509	0.5525	0.5547	0.5564	0.5556	0.5536	0.5550	0.5551
1	0.6133	0.6100	0.6107	0.6135	0.6136	0.6147	0.6125	0.6137	0.6139	0.6133
2	0.6752	0.6762	0.6781	0.6784	0.6781	0.6777	0.6789	0.6789	0.6781	0.6787
3	0.7479	0.7506	0.7499	0.7498	0.7480	0.7510	0.7513	0.7497	0.7504	0.7495
4	0.8304	0.8297	0.8291	0.8270	0.8291	0.8312	0.8298	0.8299	0.8291	0.8299
5	0.8614	0.8607	0.8595	0.8604	0.8634	0.8619	0.8620	0.8613	0.8621	0.8634
6	0.8897	0.8891	0.8898	0.8926	0.8915	0.8916	0.8907	0.8916	0.8933	0.8924
7	0.9151	0.9160	0.9188	0.9175	0.9178	0.9167	0.9178	0.9193	0.9190	0.9185
8	0.9383	0.9407	0.9391	0.9392	0.9382	0.9395	0.9411	0.9406	0.9403	0.9395
9	0.9540	0.9529	0.9522	0.9514	0.9521	0.9539	0.9540	0.9532	0.9523	0.9535
10	0.9642	0.9637	0.9625	0.9633	0.9642	0.9647	0.9644	0.9633	0.9642	0.9652
11	0.9729	0.9720	0.9724	0.9732	0.9729	0.9732	0.9724	0.9728	0.9737	0.9738
12	0.9792	0.9798	0.9799	0.9798	0.9795	0.9794	0.9798	0.9802	0.9802	0.9805
13	0.9847	0.9850	0.9846	0.9844	0.9842	0.9846	0.9851	0.9849	0.9851	0.9851
14	0.9889	0.9888	0.9884	0.9882	0.9886	0.9888	0.9888	0.9889	0.9889	0.9891
15	0.9918	0.9916	0.9913	0.9918	0.9921	0.9918	0.9920	0.9918	0.9923	0.9921
16	0.9940	0.9937	0.9942	0.9943	0.9942	0.9943	0.9943	0.9945	0.9945	0.9943
17	0.9956	0.9959	0.9962	0.9958	0.9961	0.9960	0.9961	0.9962	0.9963	0.9961
18	0.9973	0.9975	0.9973	0.9974	0.9974	0.9975	0.9975	0.9977	0.9975	0.9975
19	0.9986	0.9984	0.9984	0.9985	0.9986	0.9986	0.9985	0.9987	0.9986	0.9986
20	0.9993	0.9994	0.9993	0.9994	0.9994	0.9994	0.9993	0.9995	0.9993	0.9995
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	60	61	62	63	64	65	66	67	68	69
0	0.5559	0.5578	0.5563	0.5578	0.5587	0.5570	0.5587	0.5577	0.5581	0.5597
1	0.6153	0.6153	0.6158	0.6174	0.6164	0.6156	0.6168	0.6171	0.6180	0.6161
2	0.6790	0.6810	0.6814	0.6812	0.6814	0.6799	0.6827	0.6833	0.6804	0.6820
3	0.7518	0.7536	0.7520	0.7531	0.7522	0.7527	0.7556	0.7528	0.7532	0.7539
4	0.8322	0.8314	0.8315	0.8315	0.8325	0.8331	0.8323	0.8327	0.8326	0.8327
5	0.8634	0.8633	0.8633	0.8642	0.8645	0.8642	0.8645	0.8641	0.8648	0.8647
6	0.8928	0.8924	0.8933	0.8940	0.8931	0.8936	0.8935	0.8939	0.8941	0.8941
7	0.9186	0.9193	0.9201	0.9194	0.9191	0.9192	0.9198	0.9195	0.9203	0.9199
8	0.9409	0.9413	0.9412	0.9410	0.9405	0.9412	0.9407	0.9414	0.9416	0.9405
9	0.9543	0.9540	0.9543	0.9537	0.9543	0.9540	0.9543	0.9544	0.9541	0.9543
10	0.9652	0.9649	0.9650	0.9654	0.9650	0.9652	0.9652	0.9652	0.9655	0.9650
11	0.9738	0.9736	0.9744	0.9740	0.9740	0.9742	0.9736	0.9743	0.9742	0.9740
12	0.9805	0.9808	0.9810	0.9807	0.9808	0.9805	0.9808	0.9809	0.9808	0.9805
13	0.9855	0.9856	0.9855	0.9855	0.9853	0.9857	0.9857	0.9854	0.9853	0.9854
14	0.9894	0.9892	0.9893	0.9892	0.9893	0.9896	0.9892	0.9891	0.9891	0.9895
15	0.9922	0.9922	0.9921	0.9923	0.9926	0.9923	0.9920	0.9920	0.9924	0.9923
16	0.9945	0.9945	0.9946	0.9948	0.9946	0.9944	0.9943	0.9945	0.9945	0.9945
17	0.9962	0.9962	0.9963	0.9962	0.9962	0.9962	0.9962	0.9961	0.9962	0.9963
18	0.9976	0.9977	0.9975	0.9975	0.9976	0.9977	0.9974	0.9975	0.9977	0.9977
19	0.9987	0.9987	0.9985	0.9986	0.9988	0.9986	0.9985	0.9986	0.9987	0.9987
20	0.9994	0.9995	0.9993	0.9996	0.9994	0.9993	0.9993	0.9995	0.9995	0.9994
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	70	71	72	73	74	75	76	77	78	79
0	0.5576	0.5592	0.5582	0.5576	0.5572	0.5561	0.5560	0.5564	0.5581	0.5580
1	0.6174	0.6179	0.6162	0.6165	0.6155	0.6150	0.6147	0.6162	0.6176	0.6162
2	0.6825	0.6821	0.6810	0.6809	0.6808	0.6793	0.6805	0.6818	0.6820	0.6812
3	0.7535	0.7538	0.7526	0.7533	0.7518	0.7520	0.7529	0.7539	0.7537	0.7513
4	0.8329	0.8325	0.8325	0.8318	0.8323	0.8316	0.8327	0.8330	0.8316	0.8319
5	0.8644	0.8640	0.8636	0.8646	0.8637	0.8635	0.8652	0.8641	0.8639	0.8635
6	0.8933	0.8926	0.8935	0.8937	0.8932	0.8935	0.8938	0.8932	0.8927	0.8938
7	0.9183	0.9192	0.9196	0.9198	0.9198	0.9190	0.9201	0.9187	0.9196	0.9199
8	0.9404	0.9407	0.9410	0.9417	0.9412	0.9408	0.9413	0.9404	0.9412	0.9416
9	0.9534	0.9540	0.9542	0.9548	0.9544	0.9539	0.9543	0.9538	0.9542	0.9547
10	0.9647	0.9649	0.9653	0.9656	0.9651	0.9647	0.9653	0.9648	0.9654	0.9652
11	0.9736	0.9739	0.9742	0.9742	0.9741	0.9736	0.9741	0.9742	0.9738	0.9742
12	0.9806	0.9809	0.9806	0.9811	0.9805	0.9806	0.9811	0.9807	0.9808	0.9810
13	0.9858	0.9857	0.9856	0.9854	0.9854	0.9855	0.9857	0.9855	0.9857	0.9857
14	0.9895	0.9896	0.9893	0.9894	0.9896	0.9893	0.9893	0.9893	0.9894	0.9897
15	0.9925	0.9924	0.9923	0.9925	0.9924	0.9921	0.9923	0.9923	0.9925	0.9926
16	0.9946	0.9946	0.9947	0.9946	0.9947	0.9945	0.9945	0.9946	0.9947	0.9947
17	0.9963	0.9964	0.9962	0.9964	0.9965	0.9960	0.9963	0.9962	0.9964	0.9964
18	0.9977	0.9975	0.9976	0.9977	0.9976	0.9975	0.9975	0.9976	0.9977	0.9977
19	0.9985	0.9985	0.9987	0.9986	0.9986	0.9985	0.9986	0.9986	0.9987	0.9987
20	0.9993	0.9994	0.9993	0.9994	0.9993	0.9994	0.9994	0.9995	0.9994	0.9995
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	80	81	82	83	84	85	86	87	88	89
0	0.5567	0.5579	0.5575	0.5584	0.5573	0.5589	0.5578	0.5565	0.5589	0.5586
1	0.6158	0.6151	0.6168	0.6177	0.6168	0.6169	0.6162	0.6178	0.6169	0.6174
2	0.6792	0.6805	0.6822	0.6834	0.6812	0.6816	0.6839	0.6822	0.6823	0.6829
3	0.7520	0.7527	0.7547	0.7543	0.7529	0.7566	0.7544	0.7546	0.7549	0.7541
4	0.8313	0.8328	0.8334	0.8334	0.8354	0.8343	0.8346	0.8347	0.8339	0.8342
5	0.8646	0.8646	0.8651	0.8668	0.8663	0.8665	0.8661	0.8666	0.8665	0.8650
6	0.8938	0.8940	0.8956	0.8953	0.8961	0.8960	0.8951	0.8964	0.8949	0.8947
7	0.9198	0.9209	0.9207	0.9215	0.9219	0.9215	0.9215	0.9210	0.9208	0.9208
8	0.9418	0.9416	0.9426	0.9431	0.9427	0.9430	0.9421	0.9423	0.9424	0.9422
9	0.9544	0.9547	0.9558	0.9556	0.9558	0.9554	0.9550	0.9552	0.9551	0.9552
10	0.9653	0.9659	0.9665	0.9666	0.9662	0.9661	0.9659	0.9658	0.9663	0.9660
11	0.9741	0.9743	0.9756	0.9750	0.9746	0.9747	0.9746	0.9746	0.9749	0.9741
12	0.9807	0.9813	0.9819	0.9813	0.9812	0.9813	0.9815	0.9813	0.9808	0.9808
13	0.9859	0.9858	0.9863	0.9860	0.9860	0.9863	0.9865	0.9856	0.9858	0.9858
14	0.9894	0.9894	0.9898	0.9898	0.9899	0.9900	0.9897	0.9895	0.9898	0.9898
15	0.9925	0.9922	0.9927	0.9927	0.9929	0.9925	0.9927	0.9927	0.9928	0.9926
16	0.9946	0.9946	0.9949	0.9949	0.9948	0.9948	0.9950	0.9950	0.9949	0.9945
17	0.9964	0.9963	0.9964	0.9965	0.9963	0.9964	0.9967	0.9967	0.9963	0.9964
18	0.9977	0.9977	0.9975	0.9976	0.9977	0.9977	0.9979	0.9977	0.9977	0.9976
19	0.9988	0.9986	0.9986	0.9987	0.9987	0.9987	0.9987	0.9987	0.9986	0.9987
20	0.9995	0.9994	0.9994	0.9994	0.9994	0.9994	0.9995	0.9994	0.9995	0.9995
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	90	91	92	93	94	95	96	97	98	99
0	0.5582	0.5596	0.5601	0.5607	0.5611	0.5607	0.5617	0.5607	0.5616	0.5619
1	0.6175	0.6184	0.6191	0.6196	0.6198	0.6184	0.6199	0.6199	0.6207	0.6190
2	0.6820	0.6839	0.6836	0.6843	0.6836	0.6830	0.6852	0.6849	0.6841	0.6840
3	0.7544	0.7559	0.7554	0.7547	0.7553	0.7552	0.7567	0.7554	0.7558	0.7554
4	0.8339	0.8351	0.8334	0.8335	0.8351	0.8344	0.8343	0.8343	0.8348	0.8348
5	0.8660	0.8665	0.8651	0.8660	0.8663	0.8661	0.8653	0.8662	0.8663	0.8664
6	0.8955	0.8954	0.8948	0.8948	0.8952	0.8951	0.8947	0.8951	0.8957	0.8955
7	0.9214	0.9215	0.9210	0.9205	0.9208	0.9209	0.9204	0.9214	0.9209	0.9208
8	0.9426	0.9425	0.9418	0.9418	0.9420	0.9421	0.9426	0.9415	0.9418	0.9422
9	0.9552	0.9550	0.9550	0.9551	0.9547	0.9554	0.9546	0.9541	0.9551	0.9548
10	0.9655	0.9661	0.9659	0.9658	0.9660	0.9654	0.9651	0.9654	0.9659	0.9659
11	0.9743	0.9748	0.9746	0.9749	0.9742	0.9739	0.9741	0.9743	0.9748	0.9741
12	0.9810	0.9814	0.9813	0.9811	0.9808	0.9809	0.9806	0.9812	0.9810	0.9808
13	0.9858	0.9861	0.9856	0.9859	0.9855	0.9856	0.9855	0.9855	0.9856	0.9858
14	0.9896	0.9894	0.9896	0.9897	0.9892	0.9894	0.9890	0.9895	0.9893	0.9899
15	0.9921	0.9924	0.9925	0.9925	0.9922	0.9921	0.9924	0.9922	0.9925	0.9928
16	0.9944	0.9946	0.9948	0.9946	0.9942	0.9945	0.9946	0.9945	0.9946	0.9949
17	0.9961	0.9964	0.9964	0.9961	0.9962	0.9961	0.9965	0.9960	0.9963	0.9965
18	0.9976	0.9977	0.9975	0.9976	0.9975	0.9977	0.9976	0.9975	0.9977	0.9977
19	0.9986	0.9985	0.9986	0.9985	0.9987	0.9985	0.9987	0.9985	0.9987	0.9987
20	0.9993	0.9994	0.9993	0.9995	0.9993	0.9994	0.9994	0.9994	0.9994	0.9994
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

	Temps									
Etat	100	200	300	400	500	600	700	800	900	1000
0	0.5594	0.5575	0.5605	0.5594	0.5594	0.5611	0.5621	0.5574	0.5607	0.5606
1	0.6180	0.6152	0.6195	0.6181	0.6184	0.6192	0.6205	0.6148	0.6196	0.6180
2	0.6825	0.6804	0.6839	0.6840	0.6825	0.6851	0.6849	0.6796	0.6841	0.6832
3	0.7544	0.7524	0.7566	0.7561	0.7551	0.7554	0.7548	0.7525	0.7560	0.7539
4	0.8344	0.8324	0.8356	0.8357	0.8345	0.8337	0.8341	0.8327	0.8354	0.8325
5	0.8657	0.8640	0.8664	0.8672	0.8663	0.8651	0.8663	0.8637	0.8668	0.8643
6	0.8947	0.8938	0.8958	0.8969	0.8940	0.8950	0.8954	0.8940	0.8957	0.8939
7	0.9203	0.9202	0.9213	0.9225	0.9208	0.9206	0.9216	0.9195	0.9216	0.9199
8	0.9420	0.9421	0.9427	0.9430	0.9423	0.9417	0.9427	0.9415	0.9429	0.9428
9	0.9551	0.9544	0.9556	0.9564	0.9557	0.9547	0.9554	0.9550	0.9556	0.9557
10	0.9654	0.9655	0.9665	0.9671	0.9661	0.9651	0.9657	0.9658	0.9667	0.9661
11	0.9740	0.9741	0.9748	0.9756	0.9751	0.9737	0.9741	0.9749	0.9753	0.9748
12	0.9811	0.9809	0.9815	0.9819	0.9816	0.9801	0.9807	0.9816	0.9819	0.9813
13	0.9860	0.9858	0.9863	0.9864	0.9858	0.9852	0.9856	0.9861	0.9865	0.9861
14	0.9898	0.9897	0.9900	0.9900	0.9896	0.9891	0.9896	0.9898	0.9900	0.9900
15	0.9924	0.9927	0.9927	0.9927	0.9923	0.9921	0.9924	0.9926	0.9925	0.9928
16	0.9948	0.9950	0.9947	0.9951	0.9946	0.9942	0.9947	0.9947	0.9949	0.9950
17	0.9965	0.9966	0.9964	0.9966	0.9963	0.9960	0.9964	0.9964	0.9966	0.9964
18	0.9979	0.9978	0.9977	0.9979	0.9975	0.9973	0.9978	0.9977	0.9980	0.9975
19	0.9988	0.9988	0.9987	0.9987	0.9985	0.9983	0.9987	0.9986	0.9990	0.9986
20	0.9996	0.9995	0.9994	0.9995	0.9993	0.9992	0.9995	0.9993	0.9996	0.9994
21	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000