



Université Mohammed V de Rabat

Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes

---

# Simulation Stochastique

Prof. Ali HAMILILI

Do Not Copy - Copyright by Ali Hamlili

## Table des matières

Introduction .....	5
Chapitre 1 : Algorithmes et leur codage.....	11
1 Introduction .....	11
2 Notion de langage.....	11
3 Calcul et codage des nombres .....	12
3.1 Codage en base $B$ .....	13
3.2 Codage des entiers positifs en base $B$ .....	15
3.3 Passage du codage en base décimale au codage en base $B$ .....	15
3.4 Codage des nombres réels positifs en base $B$ .....	16
3.5 Méthode de conversion des nombres réels positifs en base $B$ .....	18
3.6 Codage des nombres réels en virgule flottante .....	19
3.7 Codage binaire des nombre entiers naturels signés.....	20
3.7.1 Méthode du signe et valeur absolue.....	20
3.7.2 Méthode du complément logique (ou complément à 1) .....	21
4 Notion de programmation infotmatique.....	21
5 Algorithmique et modélisation.....	22
6 Notions de programmation informatique .....	24
7 Code et pseudocode.....	25
7.1 Pseudocode et opérations .....	27
7.2 Structures de contrôle et pseudocode .....	27
7.2.1 Enchaînement ou block d'instructions.....	27
7.2.2 Répétition.....	28
7.2.3 Choix.....	29
7.3 Modularité ou structuration hiérarchique .....	30
8 Développements en pseudocode .....	32
8.1 Calcul de la moyenne d'un vecteur de valeurs.....	32
8.2 Problème de jeu de dés.....	33

8.3 Fonction factorielle .....	34
8.4 Echange de valeurs dans un tableau.....	35
8.5 Tri par insertion.....	35
9 Exercices de récapitulation.....	36
Chapitre 2 : Simulation de séquences de nombres au hasard.....	38
1 Notions de simulation.....	38
2 Techniques de génération de séquences de nombres au hasard.....	39
2.1 Méthodes .....	39
2.2 Générateurs procéduraux de nombres pseudo-aléatoires .....	41
2.2.1 Générateur pseudo-aléatoire .....	41
2.2.2 Méthode du carré médian de Von-Neumann .....	43
2.2.3 Méthodes congruencielles .....	44
2.2.4 Types de générateurs congruenciel .....	46
3 Critères d'existence d'une période maximale .....	47
3.1 Cas d'une base modulo générale .....	47
3.2 Cas d'une base modulo un nombre premier .....	48
3.3 Cas d'une base modulo une puissance de 2 .....	49
4 Critères d'ordre statistique et discrépance .....	50
4.1 Mesures de discrépance.....	50
4.2 Uniformité et calcul d'intégrales.....	51
4.3 Discrépance étoile .....	52
4.4 Etude d'exemples .....	53
5 Amélioration des générateurs congruenciel .....	56
5.1 Méthode du registre à décalage avec rétroaction linéaire .....	56
5.2 Méthodes congruencielles vectorielles .....	57
6 Approche Monte-Carlo pour l'évaluation d'intégrales .....	59
6.1 Forme canonique .....	60
6.2 Cas général des deux bornes finies .....	61
6.3 Autres cas unidimensionnels.....	62
6.4 Calcul des intégrales multiples.....	62
6.5 Autours de la vitesse de convergence de l'intégration Monte-Carlo .....	63

7 Simulation d'un vecteur de variables aléatoires .....	63
8 Méthode d'intégration à l'aide d'une fonction d'importance .....	64
9 Exercices de récapitulation.....	67
Chapitre 3 : Simulation de variables aléatoires réelles discrètes.....	71
1 Notions de simulation de variables aléatoires réelles discrètes.....	71
2 Méthode de transformation inverse.....	71
2.1 Principe de construction .....	72
2.2 Implémentation, en pseudo-code, d'un simulateur d'une v.a.r. discrète à domaine fini .....	73
2.2.1 Générateur de la variable .....	73
2.2.2 Générateur d'une séquence de réalisations.....	73
2.3 Exemple d'implémentation d'une v.a.r. discrète à support fini .....	74
2.3.1 Algorithme du générateur .....	74
2.3.2 Algorithme de génération de la séquence.....	74
2.3.3 Application .....	75
2.4 Implémentation d'un générateur pour une v.a. discrète à domaine infini .....	75
2.4.1 Algorithme de recherche de N0 .....	76
2.4.2 Algorithme de simulation d'une séquence .....	77
3 Méthode de rejet/acceptation .....	78
4 Méthode de composition .....	82
5 Exercices de récapitulation.....	84
Chapitre 4 : Simulation de variables aléatoires continues.....	88
1 Introduction .....	88
2 Méthode de transformation inverse .....	88
2.1 Principe de construction .....	88
2.2 Exemples d'application .....	89
3 Méthode de rejet/acceptation (variable de contrôle) .....	90
4 Méthode de composition .....	92
5 Exercices de récapitulation.....	92

## Introduction

Les ingénieurs et chercheurs dans les domaines des technologies de l'information et de communication (ICT) et des sciences des données (data science) sont souvent en présence de phénomènes qu'ils doivent observer, décrire, prédire voire même « imiter ». Il s'agit souvent d'analyse des résultats d'expériences relatives à l'évaluation et l'analyse de comportements sujets à des fluctuations imprévisibles.

Bien qu'intelligent et créatif, l'être humain a un savoir bien relatif, corrélatif et incomplet. Son savoir est très loin des vérités absolues et des connaissances typiquement hégémoniques. Ainsi, lorsque l'Homme n'arrive pas à cerner tous les facteurs qui régissent les lois de certains phénomènes ou lorsqu'il n'arrive pas à expliquer la diversité des résultats pour une même expérience répétée dans des conditions similaires, il attribue de tels phénomènes au hasard<sup>1</sup>. Pourtant, il existe certaines causes – « inconnues » de lui – qui sont à l'origine de ces différences. En fait, c'est l'imperfection de la connaissance de toutes les causes relatives à l'observation d'un phénomène et ses interactions avec les changements qui affectent son environnement qui sont souvent extériorisées sous des appellations « hasard », « incertitude » et « aléatoire » pour indiquer le même précepte.

Mais, « *Qu'est-ce que le hasard ?* » pour nous qui devons décrire son attitude à la machine (ordinateur la plupart du temps).

Comme, dans ce cours, il s'agit d'implémentation informatique de méthodes mathématiques et non de philosophie, nous allons adopter une définition empirique, c'est-à-dire basée sur l'expérience.

<sup>1</sup> Le mot *hasard* est un emprunt de la langue arabe apparue en Français au Moyen Age. Il provient du nom d'un château en Syrie « El Azar » où les croisés inventèrent un jeu de dés lors de leurs repos entre deux combats.

Nous dirons qu'une expérience dépend du hasard et nous l'appellerons *expérience aléatoire* – du mot latin *alea* qui signifie jet de dés – si elle produit un résultat que l'on ne connaît pas à l'avance mais qui en revanche appartient à un ensemble connu à l'avance.

Les probabilités et statistiques forment de nos jours un outil d'ingénierie efficace pour la modélisation de plusieurs problèmes d'intérêt commun pour l'ensemble des sciences appliquées et notamment en informatique. L'approche stochastique intervient généralement lorsque la modélisation par les outils mathématiques classiques ne peut aboutir à un modèle de prédiction satisfaisant.

Le besoin de « produire ou reproduire le hasard » par ordinateur est essentiel pour évaluer le comportement des systèmes complexes. Les méthodes de simulation font partie des méthodes sous-jacentes aux applications informatiques en statistique et sciences des données. Résumons les caractères de ces éléments paradoxaux :

- Valider des modèles probabilistes pour des comportements expérimentaux.
- Déterminer les propriétés probabilistes d'une procédure statistique non standard ou sous une loi inconnue (méthode du bootstrap).
- Calculer une valeur approchée d'intégrales/moment pour des lois non-standards (application de la loi des grands nombres).
- Maximiser des fonctions/vraisemblances peu régulières.

Ce type de procédures peut s'avérer intéressant dans plusieurs cas pratiques. Souvent des simulations à l'échelle du monde réel peuvent s'avérer essentielles pour compléter des données manquantes, prendre des décisions, concevoir des projets ou mettre en place des politiques. Ainsi, différents problèmes dans le cadre des applications des TICs et analyse des systèmes complexes trouvent leurs solutions à travers la réalisation de simulations. A titre d'exemples, nous pouvons citer :

- Réseaux informatiques
  - Etude de problèmes de communication à travers un réseau.
  - Validation de protocoles et d'architecture de communication.
  - Evaluation de performance des réseaux et de leurs composants.

- Contrôle des flux pour asservir des débits de bout-en-bout.
- Mise en évidence de goulots d'étranglement dans un réseau de communication.



**Fig. 1 :** Simulation d'un réseau 4G (LTE) à l'aide du simulateur NS-3

- Contrôle de la surcharge dans les réseaux haut-débit.
- Evaluation et contrôle statistique de la sécurité des réseaux.
- Informatique décisionnelle
  - Etude du choix d'investissements ou de lancement de nouvelles lignes de production.
  - Etude des problèmes de gestion des stocks et potentiels.
  - Reconstitution d'échantillons aléatoires pour des sondages d'opinion ou des contrôles de production.
- Génie du logiciel
  - Reconstitution du profile opérationnel de logiciels.
  - Etude de fiabilité de systèmes logiciels.
  - Assurance et certification qualité des logiciels.
- Intelligence artificielle
  - Comprendre l'intelligence naturelle nécessite de comprendre toutes ses influences sur le d'un comportement et ses interactions.
  - Multiplication de scénarii
  - Imiter par une Intelligence Artificielle des comportements d'un système naturel complexe doté d'une « Intelligence Naturelle ».



**Fig. 2 :** Sophia, un robot de type humanoïde basé sur l'intelligence artificielle moderne

- Reproduire des modèles de coévolution du système en réponse d'un environnement changeant.

La simulation de tous ces caractères emploie les nombres aléatoires et les méthodes pour en générer des séquences de longueurs souvent assez grandes. Les techniques de génération de séquences de nombres aléatoires se basent généralement sur l'idée de confection d'échantillons en statistique et donc des techniques de choix au hasard avec remise un « grand nombre de fois ». Mais la question philosophique soulevée par l'usage d'algorithmes implémentés sur des machines déterministes à mémoires de stockage finies ne permet d'aboutir qu'à des séquences de nombres dont la nature est déterministe et la taille est limitée. Donc, des séquences prévisibles avec seulement « un semblant d'aléatoire » et à la limite périodiques à partir d'un certain rang. Ainsi, si nous arrivons à confectionner des générateurs qui dissimulent tout allure de déterminisme et qui ont des périodes assez grandes, nous aurons « gagné notre pari ».

Ce cours est une introduction à ce qui est communément appelé « Simulation Stochastique », dite encore, « Simulation Monte-Carlo » destinée à des élèves ingénieurs de première années de l'ENSIAS. Il décrit l'ensemble des méthodes du calcul scientifique nécessaires au traitement informatique des problèmes requérant l'intromission des nombres aléatoires. Ainsi, il a pour objectif de répondre aux questions de savoir :

1. Comment reproduire l'aléatoire à l'aide d'un ordinateur ?
2. Pourquoi des séquences de données produites à partir de tels algorithmes déterministes peuvent être considérées comme aléatoires, indépendantes et identiquement distribuées ?
3. Comment représenter pour la machine et reproduire des aléas de support discret mais infini ou encore continu ?

Ainsi, en simulation et sous certaines conditions que nous allons élucider nous parlerons plutôt de nombres pseudo-aléatoires que de nombres aléatoires. Nous montrerons que l'approche fondamentale de simulation d'aléas (variables aléatoires) à support infini sera obtenu par la spécification d'une erreur probabiliste qui exprime l'incertitude de croyance en la maîtrise de cette propriété. Nous verrons également comment la notion de distribution uniforme permet d'appréhender la simulation de variables aléatoires continues si certaines conditions sont remplies.

Nous verrons également comment la génération de nombres pseudo-aléatoires est utilisée dans le cadre de plusieurs applications classiques et modernes en vue de munir les machines d'une certaine intelligence « artificielle ».

Avant d'entamer la discussion de ces questions et dans un objectif pédagogique, nous allons donner quelques règles de composition des algorithmes. Elles serviront de conventions pour la représentation des préceptes informatique pour l'écriture en pseudocode des algorithmes de simulation.

A la fin de chaque chapitre de ce manuscrit se trouve un nombre d'exercices qui nécessitent pour la plupart une la confection d'algorithmes en pseudocode puis une implémentation informatique dans un langage de choix. Bien que les programmes correspondant peuvent être développés dans tout langage de programmation destiné à l'ingénierie et sciences des données tels que R, Python, C/C++, Java, ..., il est conseillé aux élèves-ingénieurs de les implémenter dans les langages de programmation scientifique Octave de GNU (en téléchargement libre) ou Matlab de Mathworks (propriétaire). En effet, ces deux langages disposent d'une syntaxe assez puissante orientée mathématiques, très proches de la notation pseudocode utilisée dans ce manuscrit et sont munis de bibliothèques intégrées de visualisation graphique 2D/3D des résultats. Il est possible de télécharger les dernières versions d'Octave sous les termes de la licence GNU General Public License (GPL) à partir de l'un des sites officiels <https://www.gnu.org/software/octave>, <ftp://ftp.gnu.org/gnu/octave>, ou à partir du site miroir <https://ftpmirror.gnu.org/octave> où la version 5.2.0 d'Octave a été publiée et est maintenant disponible au téléchargement pour GNU/Linux et BSD Unix. Pour les plateformes Windows, un programme d'installation binaire pour MS-Windows officiel y est également disponible. Pour l'installation d'Octave pour le système d'exploitation macOS, on pourrait consulter les instructions d'installation à partir du wiki

[https://wiki.octave.org/Octave\\_for\\_macOS](https://wiki.octave.org/Octave_for_macOS). La dernière version d'Octave peut être étendue par des composants (packages) similaires aux boîtes à outils Matlab Toolboxes. Pour retrouver ces différents composants et bibliothèques, on pourrait visiter les sites d'entreposage Sourceforge <https://octave.sourceforge.io/> ou Github <https://gnu-octave.github.io/pkg-index/>.

La documentation nécessaire à l'installation et l'utilisation d'Octave (version 5.2.0) et ses différents composants et bibliothèques pourrait être retrouvée sur l'adresse <https://octave.org/doc/v5.2.0/>.

## Chapitre 1 : Algorithmes et leur codage

### 1 Introduction

La résolution systématique des problèmes en calculs des probabilités, statistiques où encore en simulation des processus stochastiques fait généralement appelle à des algorithmes de calcul scientifique où un savoir-faire lié aux théories traditionnelles des probabilités et statistiques doit être conjugué à une méthode de calcul. Cette méthode doit être décrite sous forme d'une suite d'instructions qui permettra de traduire la marche complète de résolution à suivre pas-à-pas à l'aide d'un moyen de calcul « *le calculateur* ». Ainsi, ce dernier devra opérer de façon systématique la méthode de calcul décrite sous forme d'une suite (ou block) d'instructions.

### 2 Notion de langage

L'échange (ou communication) d'informations entre l'Homme et la Machine pour le calcul se fait par le moyen de messages exprimés dans des langages particuliers. Ainsi, on retrouve des langages primaires où ces messages sont dressés sous forme de suites de 0 et de 1, comme c'est le cas de la programmation au niveau des microprocesseurs. Mais, on retrouve également des langages évolués qui manipulent des suites de lettres, de chiffres ainsi que des symboles de ponctuation et de séparation. Les symboles élémentaires à partir desquels est construit un langage définissent l'alphabet de celui-ci.

#### Définitions

- *On appelle alphabet un ensemble  $\Sigma$  fini et dénombrable.*
- *Les éléments d'un alphabet  $\Sigma$  sont dits des lettres de  $\Sigma$ .*

- La concaténation de suites finies de lettres de  $\Sigma$  donne lieu à des mots sur  $\Sigma$ . L'ensemble de tous les mots possibles sur l'alphabet  $\Sigma$  est noté  $\Sigma^*$ .
- On appelle langage sur un alphabet  $\Sigma$ , toute partie non vide de  $\Sigma^*$ .

Par ailleurs, on peut définir des opérations sur les mots.

### Définitions

- Soit  $w$  un mot de  $\Sigma^*$  ; on appelle longueur de  $w$  et on note  $|w|$  le nombre de symboles figurant dans l'écriture de  $w$ .
- Si  $v$  et  $w$  sont deux mots sur l'alphabet  $\Sigma$ , on appelle produit de concaténation de  $v$  et  $w$  et on note  $v \bullet w$  le mot  $u$  obtenu par écriture les uns à la suite des autres les symboles composants  $v$  suivi de ceux composants  $w$ .
- Dans l'écriture  $u = v \bullet w$ , on dit que  $v$  est facteur gauche de  $u$  et  $w$  est facteur droit de  $u$ .

La concaténation n'est pas la seule opération qu'on peut définir sur un langage.

### Exemple

Le langage machine  $\{0,1\}^n$  construit sur l'alphabet  $\{0,1\}$  construisant des mots de  $n$  bits (0 ou 1) et muni des opérations « + », « - », « \* » et « / ». La définition de ces quatre opérations fait intervenir une représentation polynomiale à coefficients dans  $\mathbb{Z}/2\mathbb{Z}$  des mots.

## 3 Calcul et codage des nombres

La notion de calcul remonte à la préhistoire. Elle est aussi ancienne que l'homme comptait avec ses doigts et des cailloux. Le mot « Calcul » en français vient du mot latin « *Calculi* » qui désigne « Caillou ». En fait, toutes les civilisations qu'elles soient grec, romaine, chinoise, maya, arabe, ... avaient toutes développées des systèmes et des bases de numération ainsi que des modes opératoires pour réaliser des calculs.

### 3.1 Codage en base $B$

On peut définir un codage en base  $B$  sur un alphabet  $\Sigma_B$  à l'aide de  $B$  symboles différents appelés les chiffres de cette base de codage.

#### Définition

*Les lettres de l'alphabet de codage en base  $B$  sont appelées des chiffres.*

#### Exemples

- En base décimale (base de 10) nous avons un alphabet de 10 chiffres

$$\Sigma_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

- En base binaire (base de 2), l'alphabet est composé de deux chiffres

$$\Sigma_2 = \{0, 1\}$$

Dans ce cas, un chiffre correspond à un bit (binary digit).

- Dans la base octale (base de 8),  $\Sigma_8$  contient huit chiffres,

$$\Sigma_8 = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

- En base hexadécimale (base de 16), nous avons seize chiffres, à savoir,

$$\Sigma_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

- Dès les débuts de l'informatique, on avait besoin de coder les caractères du clavier afin d'opérer les échanges d'informations, d'où la naissance du standard ASCII, une norme de codage de caractères en informatique (voir la figure 1.1)

Symbol	Binaire	Octale	Décimale	Hexadécimale	Symbol	Binaire	Octale	Décimale	Hexadécimale	Symbol	Binaire	Octale	Décimale	male
<b>espace</b>	100000	40	32	20	@	1000000	100	64	40	'	1100000	140	96	60
!	100001	41	33	21	A	1000001	101	65	41	a	1100001	141	97	61
"	100010	42	34	22	B	1000010	102	66	42	b	1100010	142	98	62
#	100011	43	35	23	C	1000011	103	67	43	c	1100011	143	99	63
\$	100100	44	36	24	D	1000100	104	68	44	d	1100100	144	100	64
%	100101	45	37	25	E	1000101	105	69	45	e	1100101	145	101	65
&	100110	46	38	26	F	1000110	106	70	46	f	1100110	146	102	66
'	100111	47	39	27	G	1000111	107	71	47	g	1100111	147	103	67
{	101000	50	40	28	H	1001000	110	72	48	h	1101000	150	104	68
}	101001	51	41	29	I	1001001	111	73	49	i	1101001	151	105	69
*	101010	52	42	2A	J	1001010	112	74	4A	j	1101010	152	106	6A
+	101011	53	43	2B	K	1001011	113	75	4B	k	1101011	153	107	6B
-	101100	54	44	2C	L	1001100	114	76	4C	l	1101100	154	108	6C
-	101101	55	45	2D	M	1001101	115	77	4D	m	1101101	155	109	6D
/	101110	56	46	2E	N	1001110	116	78	4E	n	1101110	156	110	6E
0	101111	57	47	2F	O	1001111	117	79	4F	o	1101111	157	111	6F
1	110000	60	48	30	P	1010000	120	80	50	p	1110000	160	112	70
2	110001	61	49	31	Q	1010001	121	81	51	q	1110001	161	113	71
3	110010	62	50	32	R	1010010	122	82	52	r	1110010	162	114	72
4	110011	63	51	33	S	1010011	123	83	53	s	1110011	163	115	73
5	110100	64	52	34	T	1001000	124	84	54	t	1110100	164	116	74
6	110101	65	53	35	U	1010101	125	85	55	u	1110101	165	117	75
7	110110	66	54	36	V	1010110	126	86	56	v	1110110	166	118	76
8	110111	67	55	37	W	1010111	127	87	57	w	1110111	167	119	77
9	111000	70	56	38	X	1011000	130	88	58	x	1111000	170	120	78
:	111001	71	57	39	Y	1011001	131	89	59	y	1111001	171	121	79
,	111010	72	58	3A	Z	1011010	132	90	5A	z	1111010	172	122	7A
,	111011	73	59	3B	\	1011011	133	91	5B	\	1111011	173	123	7B
<	111100	74	60	3C	/	1011100	134	92	5C	/	1111100	174	124	7C
=	111101	75	61	3D	1	1011101	135	93	5D	1	1111101	175	125	7D
>	111110	76	62	3E	A	1011110	136	94	5E	A	1111110	176	126	7E
?	111111	77	63	3F	-	1011111	137	95	5F	Suppr	1111111	177	127	7F

### 3.2 Codage des entiers positifs en base B

Un nombre entier positif  $N$  s'écrit en base  $B$  sous la forme :

$$N \equiv \overline{b_n b_{n-1} b_{n-2} \cdots b_1 b_0}_B$$

où chacun des  $b_i$  ( $i = 0..n$ ) est l'un des  $B$  chiffres de l'alphabet  $\Sigma_B$  défini par la base  $B$ .

$$N \equiv b_n \times B^n + b_{n-1} \times B^{n-1} + b_{n-2} \times B^{n-2} + \cdots + b_1 \times B^1 + b_0 \times B^0$$

#### Exemples (Différents codages du nombre 2257)

- En base décimale le nombre 2257 s'écrit par développement en base de 10,

$$\overline{2257}^{10} = 2 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 \equiv 2257$$

De droite à gauche, on lit le chiffre des *unités*, de *dizaines*, de *centaines*, de *milliers*, de *dizaine de milliers*, ...etc.

- Le nombre 2257 s'écrit en base binaire 100011010001. En effet,

$$\begin{aligned} 2257 &= 1 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\equiv \overline{100011010001}^2 \end{aligned}$$

- Ce même nombre 2257 s'écrit en base octale 4321. Ce qui fait,

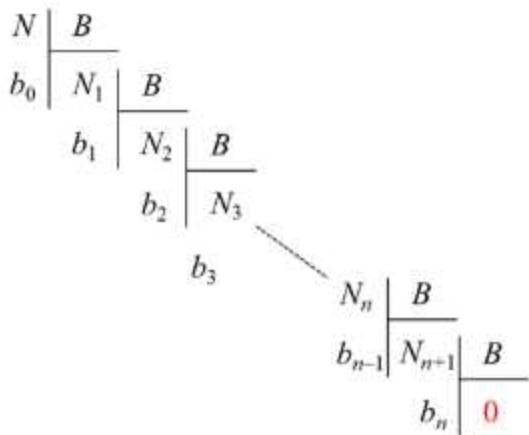
$$2257 = 4 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 \equiv \overline{4321}^8$$

- En base hexadécimale, il s'écrit 8D1. Le calcul se fait de façon aussi aisée,

$$2257 = 8 \times 16^2 + D \times 16^1 + 1 \times 16^0 \equiv \overline{8D1}^{16}$$

### 3.3 Passage du codage en base décimale au codage en base B

Un nombre entier positif  $N$  en base 10 s'écrit en base  $B$  sous la forme :

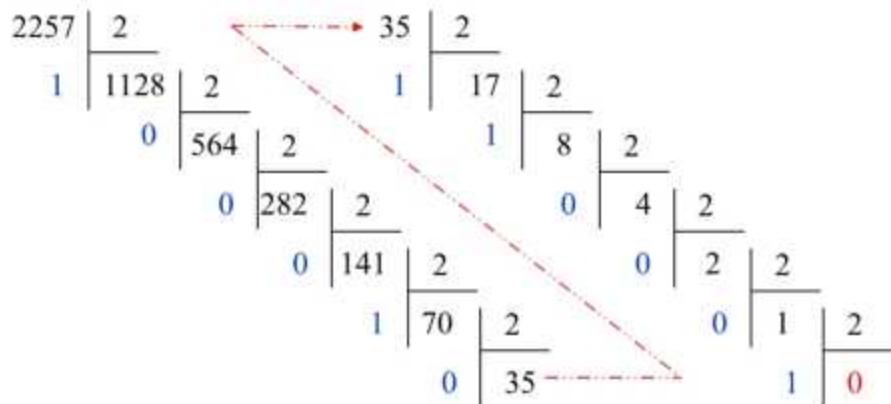


$$N \equiv b_n \times B^n + b_{n-1} \times B^{n-1} + b_{n-2} \times B^{n-2} + \dots + b_1 \times B^1 + b_0 \times B^0 \\ = \overline{b_n \ b_{n-1} \ b_{n-2} \ \dots \ b_1 \ b_0}_B$$

**Fig. 1.1 :** Déroulements des opérations de la représentation en base  $B$

### Exemple

En base binaire la conversion se fait



$$2257 = \overline{1000110100\ 01}^2$$

**Fig. 1.2 :** Déroulements des opérations de la représentation en base 2

### 3.4 Codage des nombres réels positifs en base B

Lors du codage d'un nombre réel positif  $X$ , il suffit de rajouter la partie après une virgule

$$X \equiv \overline{b_n b_{n-1} b_{n-2} \cdots b_1 b_0, v_1 v_2 \cdots v_{m-1} v_m}^B$$

où chacun des  $b_i$  ( $i = 0..n$ ), respectivement  $v_j$  ( $j = 1..m$ ), est l'un des  $B$  chiffres de l'alphabet  $\Sigma_B$  défini par la base  $B$ , c'est-à-dire,

$$X \equiv b_n \times B^n + b_{n-1} \times B^{n-1} + b_{n-2} \times B^{n-2} + \cdots + b_1 \times B^1 + b_0 \times B^0 + v_1 \times B^{-1} + \cdots + v_{m-1} \times B^{-m+1} + v_m \times B^{-m}$$

### Exemples

On considère le codage d'un nombre réel positif  $X \equiv \overline{2257,61}^{10}$

- En base décimale, on considère le codage d'un nombre réel positif. La méthode de calcul de ce nombre en base de 10 consiste à multiplier la partie après la virgule continuellement par 10 jusqu'à ce qu'il n'en reste plus. Ainsi,

$$\overline{2257,61}^{10} \equiv 2 \times 10^3 + 2 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 + 6 \times 10^{-1} + 1 \times 10^{-2}$$

- En base binaire à 8 chiffres significatifs après la virgule, la méthode de calcul de ce nombre en base binaire consiste à multiplier la partie après la virgule continuellement par 2 jusqu'à ce qu'il n'en reste plus, si possible,

$$\begin{array}{ll} 0,61 \times 2 = \underline{1} + 0,22 & 0,76 \times 2 = \underline{1} + 0,52 \\ 0,22 \times 2 = \underline{0} + 0,44 & 0,52 \times 2 = \underline{1} + 0,04 \\ 0,44 \times 2 = \underline{0} + 0,88 & 0,04 \times 2 = \underline{0} + 0,08 \\ 0,88 \times 2 = \underline{1} + 0,76 & 0,08 \times 2 = \underline{0} + 0,16 \\ & \vdots \end{array}$$

Ainsi,

$$\begin{aligned} 2257,61 &\approx \overline{100011010001,10011100}^2 \\ &\equiv 2^{11} + 2^7 + 2^6 + 2^4 + 2^0 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-6} \end{aligned}$$

- De la même manière, en base octale à 8 chiffres significatifs après la virgule

$$2257,61 \approx \overline{4321,47024365}^8$$

$$\equiv 4 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 8^0 + 4 \times 8^{-1} + 7 \times 8^{-2} + 2 \times 8^{-3} +$$

$$4 \times 8^{-4} + 3 \times 8^{-5} + 6 \times 8^{-6} + 5 \times 8^{-7}$$

- La même méthode s'applique en base hexadécimale. A 8 chiffres significatifs après la virgule, nous écrivons

$$2257,61 \approx \overline{8D1,9C28F5C2}^{16}$$

$$\equiv 8 \times 16^2 + D \times 16^1 + 1 + 9 \times 16^{-1} + C \times 16^{-2} + 2 \times 16^{-3} +$$

$$8 \times 16^{-4} + F \times 16^{-5} + 5 \times 16^{-6} + C \times 16^{-7} + 2 \times 16^{-8}$$

### 3.5 Méthode de conversion des nombres réels positifs en base $B$

Pour la conversion des nombres réels positifs en base  $B$ , la méthode consiste à conduire les étapes suivantes :

- Décomposer le nombre réel en la somme de la partie entière et la partie fractionnaire
- Conversion de la partie entière : il faut utiliser la méthode de division entière comme pour les entiers positifs
- Conversion de la partie fractionnaire :
  - Multiplier la partie fractionnaire par  $B$
  - Décomposer ce nombre en une somme de la partie entière et la partie fractionnaire
  - Recommencer cette opération pour la partie fractionnaire du résultat et ainsi de suite
  - Arrêter le calcul lorsqu'on obtient une partie fractionnaire est nulle ou lorsqu'on atteint la précision souhaitée
  - La partie fractionnaire en base  $B$  est obtenu par la concaténation des parties entières obtenues dans l'ordre de leur apparition au terme de ce processus de calcul

#### Remarque

La conversion de la partie fractionnaire n'admet pas nécessairement une *représentation exacte* sous forme d'un nombre fini de chiffres.

### 3.6 Codage des nombres réels en virgule flottante

Le précepte du codage des nombres réels en virgule flottante est de pouvoir représenter un nombre réel avec une virgule flottante et une précision (approximation) limitée. Cette approche permet alors de ne coder que des chiffres significatifs représentant ce nombre. Ainsi, le nombre est présenté sous forme normalisée de manière à déterminer sa mantisse et son exposant.

Dans ces conditions, la forme normalisée du nombre  $X$  codé en base  $B$  s'écrit

$$X \equiv \pm M \times B^E$$

où  $B$  représente base de codage,  $M$  est la mantisse (nombre de  $m$  chiffres) en base  $B$ ,  $E$  est l'exposant (nombre de  $n$  chiffres) en base  $B$ ,  $\pm$  est le symbole qui représente le signe positif ou négatif du nombre  $X$ .

#### Exemple

Le codage du nombre réel  $X = \overline{2257,61}^{10}$  en base de 10 s'écrit

$$X \equiv +0,225761 \times 10^4$$

Dans cet exemple, la base de codage est  $B = 10$ , la mantisse est  $M = 0,225761$ , l'exposant est  $E = 4$  et le signe est positif, donc « + ».

#### Remarque

Dans la forme normalisée,  $\pm 0,xxxxxx \times B^E$ , il n'y a pas de chiffres avant la virgule.

#### Standard IEEE 754

Dans cette norme IEEE, on considère la représentation binaire des nombres réels à virgule flottante avec trois types de précisions possibles :

- Dans la précision simple (32 bits)
  - 1 bit est consacré pour le codage du signe,
  - 8 bits pour le codage de l'exposant,
  - 23 bits pour le codage de la mantisse.

Ainsi, dans ce cas, l'exposant est décalé de  $2^{8-1} - 1 = 127$ . En fait, l'exposant d'un nombre normalisé est compris entre les bornes  $-126$  et  $+127$ ,

$$-126 \leq E \leq +127$$

Ainsi, l'exposant  $-127$  se retrouve décalé vers la valeur  $0$  (réservé à représenter  $0$ ), les nombres sont dénormalisés et l'exposant  $128$  est décalé vers  $255$  qui est réservé pour coder les infinis et les NaN (*Not a Number*).

- En double précision (64 bits)
  - 1 bit est consacré pour le codage du signe,
  - 11 bits pour le codage de l'exposant,
  - 52 bits pour le codage de la mantisse.
- La précision double étendue considère un codage sur 80 bits.
  - 1 bit est consacré pour le codage du signe,
  - 15 bits pour le codage de l'exposant,
  - 64 bits pour le codage de la mantisse

### 3.7 Codage binaire des nombres entiers naturels signés

Il existe trois méthodes pour coder des entiers naturels signés en base de  $2$ , à savoir :

- La méthode utilisant un bit de signe et codant la valeur absolue ;
- La méthode du complément logique ;
- La méthode du complément arithmétique.

Toutes ces méthodes utilisent un bit pour indiquer le signe du nombre entier codé.

#### 3.7.1 Méthode du signe et valeur absolue

##### Définition

*On appelle bit de poids fort, le bit plus à gauche et bit de poids faible celui qui est le plus à droite.*

##### Principe

*Le bit de poids fort code le signe et il vaut :*

- 0 si le nombre entier positif.
- 1 si le nombre entier négatif.

Les autres bits codent le nombre en valeur absolue

### Remarque

Il est essentiel de connaitre sur combien de bits on code les nombres entiers ; cela permet de déterminer le domaine de codage.

#### 3.7.2 Méthode du complément logique (ou complément à 1)

### Principe

*Le complément logique, dit aussi « complément à 1 », est obtenu en remplaçant les 1 par des 0 et inversement.*

Le codage des nombres signés par la méthode du complément logique permet de représenter

- Les nombres positifs : comme dans la représentation des entiers non-signés.
- Les nombres négatifs : comme le complément logique de son opposé positif.
- Le bit de poids le plus fort code le signe : 0 si le nombre est positif et 1 s'il est négatif.

### Exemple

Considérons le codage sur un octet du nombre entier naturel  $\overline{01010011}^2 = 83$ , son complément à 1 est  $\overline{10101100}^2 = -83$  et non pas 172.

L'inconvénient de cette méthode est, comme dans la méthode précédente, 0 est représenté 2 fois avec

$$-2^{p-1} + 1 \leq N \leq 2^{p-1} - 1$$

## 4 Notion de programmation informatique

Généralement un ordinateur est une machine de calcul destinée à effectuer des opérations élémentaires : l'affectation, l'addition, la soustraction et la comparaison.

### Définition

*Un programme informatique est un document permettant de décrire à l'aide de variables l'agencement du déroulement des opérations sur ces variables dans le temps.*

Un programme est donc un « objet » statique qui évolue lors de son exécution de façon dynamique en fonction de l'état de ses variables.

La question à laquelle doit faire face tout programmeur est comment pourrait-il concevoir un programme afin d'avoir une vision aussi claire que possible des situations dynamiques qu'engendre le programme lors de son exécution afin de fournir une démonstration plus ou moins rigoureuse de sa validité ; autrement dit, prouver que le programme fournira les résultats désirés par ses spécifications.

## 5 Algorithmique et modélisation

De manière informelle, nous pouvons dire qu'un algorithme est une « *méthode de calcul bien définie* » qui prend une ou plusieurs valeurs en entrée et produit une ou plusieurs valeurs en sortie. C'est donc une suite « *d'étapes de calcul* » ou « *d'instructions* » qui transforme la ou les entrées en sorties. On pourrait également voir qu'un algorithme est en lui-même un outil pour résoudre un problème de calcul bien déterminé. L'énoncé du problème spécifie en termes généraux la relation entre entrées et sorties désirées. L'algorithme décrit une méthode de calcul spécifique pour réaliser cette relation « *correspondance* ». Ainsi, on dit que l'algorithme définit une « *résolution* » du problème.

### Définition

*Un algorithme est une suite finie d'instructions permettant la résolution systématique d'un problème donné.*

Un algorithme peut être utilisé soit pour décrire par une suite d'instructions ou de procédures qui exposent la marche complète à suivre pour résoudre un problème donné, soit afin d'automatiser une tâche complexe, pour laquelle on sait déjà comment résoudre le problème correspondant ; dans cette approche on cherche à tirer parti de la vitesse des moyens informatiques pour effectuer automatiquement toutes les étapes et tous les calculs intermédiaires qui permettent d'aboutir

rapidement à un résultat, soit chercher la solution d'un problème qu'on ne sait pas a priori résoudre mais tel qu'on puisse tirer parti du système informatisé afin d'explorer l'ensemble des possibilités, et ainsi tenter de trouver la solution, ou du moins une bonne approximation de celle-ci. Dans la mesure où cette résolution est destinée à être exécutée sur un ordinateur, on parle souvent de résolution numérique ou informatique.

Il y a, en général, quatre niveaux de représentations informatiques d'un algorithme qui est destiné à être exécuté sur un calculateur :

- *Niveau représentation machine* décrit manipulation des nombres et des types tels que chacun correspond à une représentation machine binaire.
- *Niveau langage de programmation* est un ensemble d'instructions ou opérations pour décrire les étapes de l'algorithme dans un langage compréhensible par l'Homme et la Machine à la fois.
- *Niveau logiciel* représente un programme écrit dans un langage de programmation permettant ainsi d'opérer des fonctionnalités.
- *Niveau opérationnel* est un profil opérationnel décrivant la manière d'utilisation du logiciel.

En résumé, résoudre informatiquement un problème, consiste à remplacer le problème exprimé dans son contexte théorique par un problème équivalent dont il existe une formulation par une méthode ou algorithme de calcul sous forme d'une suite d'instructions. Cela traduit en quelques sortes une modélisation du problème.

### Exemple

On peut modéliser l'expérience aléatoire qui consiste à lancer un dé non pipé dont les résultats possibles sont les points apposés sur la face supérieure du dé.

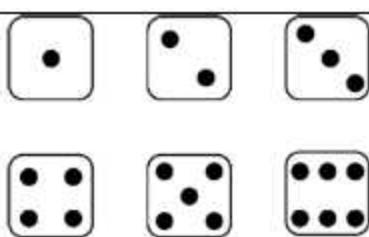


Fig. 1.3 : Ensemble des résultats possibles lors du jet d'un dé à non pipé

par un tirage aléatoire d'un nombre au hasard compris entre « 1 » et « 6 » précisant le nombre de points qui seraient affichés par la face supérieure du dé. L'hypothèse de dés non pipé se traduit par une distribution uniforme sur l'ensemble des valeurs {1,2,3,4,5,6} .

Nous dirons qu'un algorithme est « *correct* » s'il résout convenablement le problème de calcul donné. Autrement dit, un algorithme sera proclamé correct si, pour chaque cas d'entrée, il termine (ou s'arrête) avec une sortie correcte. Contrairement, un algorithme « *incorrect* » pourrait ne pas s'arrêter pour tous les cas d'entrée, ou il pourrait rendre dans certains cas des réponses autres que celles désirées. On dit que pour ces cas d'entrées, ils rendent des résultats incorrects ou divergents. Ce qui permet de définir, par opposition, les « *réponses convergentes* » d'un algorithme incorrect.

### Remarque

A ce niveau, remarquons toutefois que contrairement à ce que pourrait s'attendre un néophyte, les algorithmes incorrects peuvent parfois être très utiles, si leurs taux d'erreur peuvent être contrôlés.

## 6 Notions de programmation informatique

La programmation recouvre l'ensemble des techniques permettant de résoudre des problèmes à l'aide de programmes destiné pour être exécutés sur un ordinateur. Ainsi, il existe plusieurs types de programmation en informatique :

- *Programmation procédurale* : paradigme basé sur le concept d'appel de procédures (exemples : Abal, Ada, ALGOL, BASIC, C, COBOL, Fortran, Perl, ...).
- *Programmation fonctionnelle* : paradigme reposant sur l'évaluation de fonctions mathématiques (Erlang, F#, Haskell, Lisp, ML, OCaml, ...).
- *Programmation orientée objets* : paradigme utilisant des types abstraits de données : objets, classes, classes génériques, héritage (exemples : AS3, C++, C#, Java, Objective C, Eiffel, LOGO, PHP, Python, Ruby, Smalltalk, Vala, ...).
- *Programmation logique* : paradigme utilisant des clauses, unification, instanciation, contrôle (exemples : Absys, Conniver, Ether. Planner, Popler, Prolog, QLISP, ...).
- *Programmation parallèle* : paradigme basé sur des processus, synchronisation, communication (exemples : Java, Modula II, Threaded-C, ...).

- *Langages de spécification* : les programmes sont sous forme de critères basés sur la logique du premier ordre et sur la théorie des ensembles (exemples de formalismes : B, CCS, CSP, VDM, Z, ...).
- *Programmation automatique* : c'est une réalisation de programmes à partir de la description des fonctions à remplir (environnements de génie logiciel) souvent liés souvent à une méthodologie générale de résolution de problème (exemple : SMXCogitor).
- *Programmation impérative* : elle est basée sur la notion d'actions exécutées. Elle regroupe la plupart des formes précédentes de programmation.

### Définition

*La programmation impérative est basée sur la notion d'actions exécutées.*

Un langage est dit impératif lors qu'un programme écrit dans ce langage consiste en une séquence d'instructions données destinées à être exécutées par une machine. L'action exécutée la plus typique est l'affectation de variables. Le concept de variable en programmation est différent de la notion mathématique de variables. En programmation, une variable est une case mémoire de l'ordinateur dans laquelle on peut stocker la valeur d'une donnée (numérique ou autre). La valeur d'une variable doit pouvoir être récupérée et modifiée à volonté.

### 7 Code et pseudocode

On rencontre de très nombreux langages de programmation adaptés pour différents systèmes supports de ces langages, par exemple, des langages sur ordinateurs dont Basic, Fortran, C, C++, Tcl/OTcl, Python, Pearl, Octave (ou Matlab), Java et Lisp, mais aussi des langages spécifiques à des constructeurs de calculatrices dites « *programmables* ».

### Définition

*En programmation impérative, on appelle code l'expression d'un algorithme dans un langage de programmation donné.*

Un processus de développement intégrant la mise en œuvre d'un pseudocode est une approche qui s'avère très utile lorsque l'informaticien souhaite dérouler son projet par raffinements progressifs

selon une approche de structuration de haut vers le bas (Top-Down). Ainsi, il doit exprimer dans un premier temps les différents algorithmes spécifiques à son projet, sous forme de descriptions pseudocodes. A cette étape, il pourra modifier ces algorithmes afin de les faire interagir correctement en vue de leur intégration. Plus tard, il pourra traduire ces descriptions dans le langage de programmation cible afin de les tester sur machine.

L'objectif essentiel d'une approche par pseudocode est d'exprimer d'abord les algorithmes dans un langage proche des langages naturels, par exemple l'Arabe, le Français ou l'Anglais, dans lesquels il est facile d'exprimer les spécifications et d'améliorer l'algorithme indépendamment des contraintes syntaxiques du langage de programmation ainsi que des fonctions prédéfinies spécifiques à ce langage de programmation. Le pseudocode tend ainsi à s'affranchir des détails trop techniques des langages de programmation tout en offrant la possibilité d'être retraduit dans ces langages à une étape avancée du projet.

### Définition

*On entend par pseudocode une description informelle de haut niveau du principe de fonctionnement d'un algorithme.*

Il utilise les conventions de structure du code ; mais, il est destiné à la lecture de l'Homme plutôt que la machine. A ce titre, l'écriture d'un algorithme en pseudocode permet de réduire la difficulté de la formulation informatique de celui-ci. Avant d'écrire une solution à un problème dans un langage de programmation, on le formule d'abord en pseudocode ; ce qui forme une étape intermédiaire qui permet de décomposer la difficulté de l'écriture des programmes. En effet, à l'étape de développement du pseudocode, on se concentre sur l'essentiel des modèles mathématiques utilisés pour résoudre le problème sous forme d'algorithme. Dans une étape suivante, on se préoccupera de traduire le pseudocode dans la syntaxe impérative du langage de programmation souhaité en utilisant ses fonction prédéfinies propres. Cette technique évite ainsi de s'inquiéter simultanément de la modélisation mathématique de la solution et de la difficulté de l'exprimer selon les contraintes syntaxiques du langage utilisé. Le pseudocode essaye de répondre à la normalisation de la pré-écriture de programmes en présence d'une diversité de langages de programmations malgré l'adversité de leurs syntaxes.

## 7.1 Pseudocode et opérations

### Postulat

- L'affectation «  $\leftarrow$  » est une opération prédéfinie dans l'écriture pseudocode.
- Le symbole « ; » définit un séparateur d'instructions.
- Les opérations de lecture « **Lire** » et d'écriture « **Ecrire** » sont supposées prédéfinies.
- Dans une écriture pseudocode toutes les opérations élémentaires auxquelles pourrait faire appel le domaine de définition des objets manipulés sont supposées prédéfinies.

### Exemples

- Les opérations d'addition « + », soustraction « - », multiplication « \* » et modulo « mod » sont supposées être prédéfinies pour le calcul arithmétique.
- L'union «  $\cup$  », l'intersection «  $\cap$  », la différence « \ » et la complétion (ou négation) «  $\neg$  » ensemblistes sont supposées être prédéfinies pour l'algèbre ensembliste.

## 7.2 Structures de contrôle et pseudocode

### Postulat

*Les seules structures de contrôle qui existent en écriture pseudocode sont l'enchaînement, la répétition et le choix.*

### 7.2.1 Enchaînement ou block d'instructions

#### Définition

*Un enchaînement consiste en une succession d'un nombre quelconque d'instructions d'affectation, de lecture et d'écriture.*

La syntaxe d'un enchainement est alors sous la forme :

```
Instruction1 ;  
Instruction2 ;  
...  
InstructionN ;
```

#### Exemple

```
lire(a);  
b←3;  
c←2;  
d←(b+c^a)/91;  
Afficher(d);
```

## 7.2.2 Répétition

### Définition

La répétition d'un ensemble (ou block) d'instructions, appelée aussi boucle, peut prendre l'une des deux formes suivantes :

```
Pour i ← n à m Faire  
    Instruction1 ;  
    Instruction2 ;  
    ...  
Fin_pour ;
```

ou

```
Tant_que Condition Faire  
    Instruction1 ;  
    Instruction2 ;  
    ...  
Fin_tant_que ;
```

Le nombre de répétitions pour la première forme est défini par le nombre d'éléments de l'ensemble  $\llbracket n..m \rrbracket$ ; alors que pour la seconde forme, la répétition se fait à l'infini tant que la clause « Condition » reste vraie.

### Exemples

```
...  
Pour i ← 5 à 17 Faire  
    S[i] ← i^2
```

```
Fin_pour ;
```

```
...
```

ou encore en utilisant une boucle conditionnelle

```
...
i ← 5;
Tant_que i < 18 Faire
    Ecrire(i2)
    i ← i+1;
Fin_tant_que ;
...
```

### 7.2.3 Choix

La notion de choix permet d'opérer différents blocks d'instructions selon la réalisation de certaines conditions particulières.

#### Définition

*Les instructions qui permettent d'opérer un choix parmi différents blocks d'instructions possibles peuvent s'écrire sous la forme suivante :*

```
Si condition1 Alors
    instruction1.1 ;
    instruction1.2 ;
    ...
Sinon Si condition2 Alors
    instruction2.1 ;
    instruction2.2 ;
    ...
Sinon Si condition(N) .Alors
    instruction(N).1 ;
    instruction(N).2 ;
    ...
```

```

Sinon
    instruction(N+1).1 ;
    instruction(N+1).2 ;
    ...
Fin_si ;

```

On exécute le block d'instructions qui suit la première condition si celle-ci est vraie et on sort, sinon, on exécute le deuxième block si c'est la deuxième condition qui est vérifiée et ainsi de suite, on avance profondeur jusqu'à la fin. Si aucune condition n'est vérifiée, on exécute les instructions qui suivent le dernier « **Sinon** ».

### **Exemple**

#### **Algorithme**

```

...
Si i < 6 Alors
    Afficher(i^2) ;
Sinon Si i < 16 Alors
    a ← i^3 ;
    Afficher(a) ;
Sinon
    b ← i mod 12 ;
    Afficher(b+7) ;
Fin_si ;
...

```

### 7.3 Modularité ou structuration hiérarchique

Par opposition à la notion d'algorithmes celle de module permet de structurer (ou décomposer) un algorithme en sous-algorithmes. L'approche par structuration hiérarchique permet de vaincre la complexité des algorithmes trop longs ou dont la formulation est trop difficile.

Il existe deux notions en pseudocode pour déclarer des modules d'un algorithme général : à savoir les formes procédurales et les formes fonctionnelles.

### Définition

*Les formes procédurales qui s'écrivent sous la forme suivante :*

```
Proc Nom_procedure(variables_arguments)
% Déclaration des variables locales
Local{Global} liste_des_variables_locales{globales} ;
% Corps_du_module
Instruction1 ;
Instruction2 ;
...
Fin_proc.
```

### Définition

*Les formes fonctionnelles qui s'écrivent sous la forme :*

```
Fonc Res ← Nom_fonction(variables_arguments)
% Déclaration des variables locales
Local{Global} liste_des_variables_locales{globales} ;
% Corps_du_module
Instruction1 ;
Instruction2 ;
...
Res ← ... ;
Fin_fonc.
```

La différence entre ces deux notions vient du fait qu'une fonction rend nécessairement un résultat évaluable, alors qu'une procédure a pour rôle d'effectuer un ensemble de traitements, pour qu'elle puisse rendre un résultat il faut que certains des instructions du corps du module soient de la forme

**Rendre**(ou une commande équivalente)( ...) ;

Ce type d'instructions devrait produire la restitution d'une certaine « valeur » ou l'écriture de « quelque chose » dans « quelque chose d'autre ».

### Exemples

On pourrait opter soit pour la forme procédurale soit pour la forme fonctionnelle pour écrire en pseudocode un algorithme qui calcul la somme de trois nombres. Ainsi, on pourrait écrire indifféremment

#### Algorithme

```
Proc P_somme3(x,y,z)
% Déclaration des variables locales
Global Res ;
% Corps_du_module
Res ← x+y+z;
Fin_proc.
```

Ou encore

#### Algorithme

```
Fonc Res ← F_somme3(x,y,z)
% Corps_du_module
Res ← x+y+z;
Fin_fonc.
```

Le résultat des deux formes est le même. Remarquez aussi que les instructions « Fin\_proc. » et « Fin\_fonc. » terminent par un « . » (point) et non pas un « ; » (point-virgule).

## 8 Développements en pseudocode

### 8.1 Calcul de la moyenne d'un vecteur de valeurs

Pour écrire en pseudocode l'algorithme qui permet de calculer la moyenne d'un vecteur de n valeurs

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

on peut utiliser essentiellement cinq variables :  $x$  la variable en entrée pour la procédure de calcul de la moyenne ; puis  $i$ ,  $n$ , somme et  $xbar$  quatre variables locales pour retenir respectivement les valeurs des indices des composants de  $x$ , la longueur du vecteur  $x$ , la somme des valeurs des composants de  $x$  et la valeur moyenne à afficher et une fonction supposée prédéfinie `longueur` qui produit le nombre des composants d'un vecteur.

### Algorithme

```
% Déclaration du nom de procédure avec ses arguments
Proc Ma_moyenne (x)

% Déclaration des variables locales %
Local i, n, somme, xbar ;

% Initialisation
somme ← 0;
% Boucle
Pour i ← 1 à n faire
    somme ← somme + x(i)
Fin_pour;
xbar ← somme / n;
% Affichage du résultat
Afficher xbar ;
fin_proc.
```

### 8.2 Problème de jeu de dés

On considère le problème de lancer d'un dé et on s'intéresse plus exactement à compter le nombre de fois que le chiffre 3 est apparu dans la suite de lancés du dé un certain nombre de fois. Pour écrire le pseudocode correspondant à l'algorithme qui permet d'implémenter ce problème on a besoin de trois variables principales :

- `NbreRelances` : le nombre total de lancers que l'on va simuler ;

- Compteur : une variable dont la valeur est initialement nulle et à laquelle, à chaque lancer, on ajoutera 1 si un 3 est obtenu. Ainsi, après les NbreLancers, la valeur de la variable Compteur sera le nombre total de fois que 3 a été obtenu.
- FaceSupDe : une variable dont la valeur sera un chiffre aléatoire entre 1 et 6.

### Algorithme

```
% Déclaration du nom de procédure avec ses arguments %
Proc Nombre_de_trois (NbreRelances)
% Déclaration des variables locales %
Local Compteur ;
% Initialisations %
Compteur ← 0 ;
% Exécution des lancers sous forme de boucle %
Pour i ← 1 à NbreRelances faire
    De ← Nbre_au_hasard_entre(1, 6) ;
    % Nbre_au_hasard_entre est supposée être une fonction prédéfinie %
    Si De = 3 Alors Compteur ← Compteur+1
        Fin_Si
    Fin_Pour ;
    Afficher "Nombre de 3 obtenus = ", Compteur
Fin_proc.
```

Remarquez ici que les séparations par des « ; » à la fin des blocs d'instructions devient optionnelles si elle l'instruction de fin est suivie d'une autre instruction de fin.

### 8.3 Fonction factorielle

Le calcul de la fonction factorielle d'un nombre entier naturel peut s'écrire en pseudocode en introduisant une variable d'incrémentation i et variable Fact qui reçoit le résultat.

### Algorithme

```
Fonc fact ← Factorielle (n)
local i ;
```

```

i ← 0 ;
fact ← 1;
Tant_QUE i < n faire
    i ← i+1 ;
    fact ← fact*i
Fin_Tant_Que
Fin_fonc.

```

#### 8.4 Echange de valeurs dans un tableau

Nous souhaitons produire dans cet exemple un algorithme qui réalise l'échange des valeurs de deux cases données d'un tableau  $A$  de dimension  $n$ .

#### Algorithme

```

Proc Echanger(A,i,j)
Local n,x ;
n ← longueur(A) ;
x ← A[i];
A[i] ← A[j] ;
A[j] ← x
Fin_proc.

```

#### 8.5 Tri par insertion

Dans cet exemple, on se propose de produire un algorithme qui produit le tri par insertion d'un tableau  $A$  de dimension  $n$ .

#### Algorithme

```

Proc Tri_par_insertion(A)
Local i,j,n ;
n ← longueur(A) ;
Pour i = 1 à n - 1 faire
    Pour j = i + 1 à n faire
        Si (A[j] < A[i]) Alors Echanger(A,i,j)

```

```
Fin_si  
Fin_pour  
Fin_pour  
Fin_proc.
```

## 9 Exercices de récapitulation

**E 1.1** Trouvez des méthodes simples pour opérer des conversions du codage en base binaire au codage en base octale et inversement.

**E 1.2** Trouvez des méthodes simples pour opérer des conversions du codage en base binaire au codage en base hexadécimale et vice-versa.

**E 1.3** Ecrire en pseudocode l'algorithme qui insert l'élément à la position  $i$  dans un tableau  $x$  de  $n$  nombres réels.

**E 1.4** Ecrire en pseudocode l'algorithme qui efface l'élément à la position  $i$  dans un tableau  $x$  de  $n$  nombres réels.

**E 1.5** Ecrire en pseudocode l'algorithme qui remplace par 0 l'élément à la position  $i$  dans un tableau  $x$  de  $n$  nombres réels.

**E 1.6** Ecrire en pseudocode l'algorithme de la procédure qui produit le maximum d'un tableau  $x$  de  $n$  nombres réels.

**E 1.7** Ecrire en pseudocode l'algorithme de la procédure qui produit le minimum d'un tableau  $x$  de  $n$  nombres réels.

**E 1.8** Ecrire en pseudocode l'algorithme de la procédure qui produit la moyenne d'un tableau  $x$  de  $n$  nombres réels distribués selon une loi de probabilité de vecteur de probabilités  $p$ .

**E 1.9** Ecrire en pseudocode l'algorithme de la fonction qui produit la variance d'un tableau  $x$  de  $n$  nombres réels.

Do Not Copy. Copyright by Ali Hamlili

---

## Chapitre 2 : Simulation de séquences de nombres au hasard

---

### 1 Notions de simulation

L'usage d'ordinateurs, de plus-en-plus rapides et performants, a rendu possible une large gamme de calculs très complexes. Ainsi, avec le développement de l'informatique et des méthodes de calcul, les ingénieurs, les statisticiens et les chercheurs en sciences des données ont trouvé le moyen de reproduire artificiellement le hasard. Les méthodes Monte-Carlo forment ainsi un moyen pour résoudre des problèmes de simulation et de prédition qui combine à-la-fois le choix de nombres au hasard et l'usage des lois de probabilité.

Les méthodes Monte-Carlo font partie des méthodes sous-jacentes aux applications informatiques en statistique. Elles permettent de reconstituer des réalités souvent complexes où la décision est confrontée à l'incertitude. Ainsi, on est généralement amené à répéter des expériences fictives dans le but de générer à grande vitesse la manifestation du hasard sous forme de séquences de nombres. La méthode Monte-Carlo de base est due à S. Ulam et N. Métropolis. Elle fait référence aux jeux de hasard, une attraction très populaire à Monte-Carlo, Monaco (Metropolis et Ulam, 1949 ; Hoffman, 1998). Les résultats de cette branche de calcul scientifique ont souvent été compilés et réemployés afin de prendre des décisions.



**Fig. 2.1** : Jeux de hasard dans les casinos de Monaco

## 2 Techniques de génération de séquences de nombres au hasard

Par définition la notion de nombres au hasard intervient dans toute simulation Monte Carlo.

### Définition

*Par le terme de simulation (ou méthode Monte-Carlo), on désigne l'ensemble des techniques qui permettent de créer ou reproduire artificiellement des suites de données (nombres) suivant des lois théoriques de probabilité préalablement fixées.*

### 2.1 Méthodes

Il existe deux grandes classes de méthodes pour générer des séquences de nombres au hasard :

- L'utilisation de tables de nombres au hasard, on peut citer :
  - Les tables de Fisher and Yates : Statistical tables for biological, agricultural and medical research.
  - Les tables de Kendall and Babington Smith : 100 000 chiffres obtenus à partir d'un disque tournant divisé en secteurs multiples de 10 éclairé de façon intermittente.
  - Les tables de Rand Corporation 10<sup>6</sup> de chiffres obtenus à partir de l'écrêtage d'un bruit de fonds.

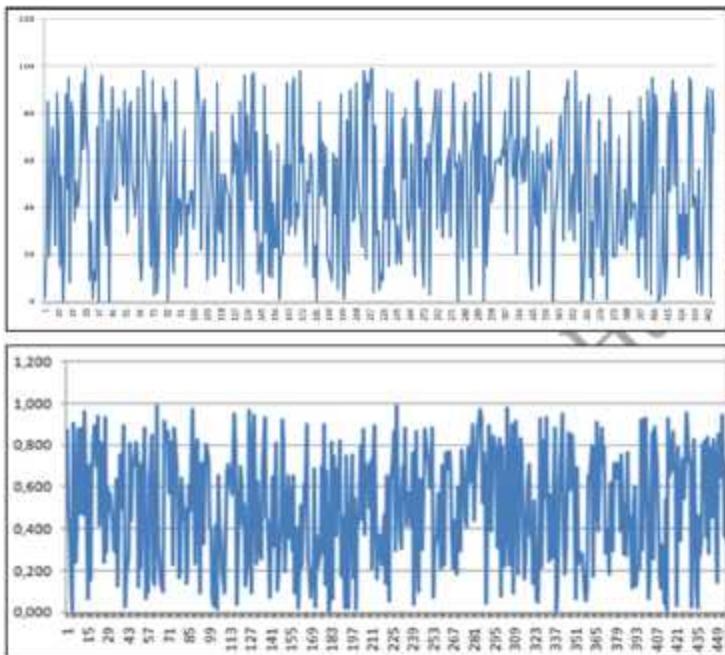
02 22 85 19 48 74 55 24 89 69 15 53 00 20 88 48 95 08
85 76 34 51 40 44 62 93 65 99 72 64 09 34 01 13 09 74
00 88 96 79 38 24 77 00 70 91 47 43 43 82 71 67 49 90
64 29 81 85 50 47 36 50 91 19 09 15 98 75 60 58 33 15
94 03 80 04 21 49 54 91 77 85 00 45 68 23 12 94 23 44
42 28 52 73 06 41 37 47 47 31 52 99 89 82 22 81 86 55
09 27 52 72 49 11 30 93 33 29 54 17 54 48 47 42 04 79
54 68 64 07 85 32 05 96 54 79 57 43 96 97 30 72 12 19
25 04 92 29 71 11 64 10 42 23 23 67 01 19 20 58 35 93
28 58 32 91 95 28 42 36 98 59 66 32 15 51 46 63 57 10
64 35 04 62 24 87 44 85 45 68 41 66 19 17 13 09 63 37
61 05 55 88 25 01 15 77 12 90 69 34 36 93 52 39 36 23
98 93 18 93 86 98 99 04 75 28 30 05 12 09 57 35 90 15
61 89 35 47 16 92 20 16 78 52 82 37 26 33 67 42 11 93
94 40 82 18 06 61 54 67 03 66 76 82 90 31 71 90 39 27
54 38 58 65 27 70 93 57 59 00 63 56 18 79 85 52 21 03
63 70 89 23 76 46 97 70 00 62 15 35 97 42 47 54 60 60
61 58 65 62 81 29 69 71 95 53 53 69 20 95 66 60 50 70
51 68 98 15 05 64 43 32 74 07 44 63 52 38 67 59 56 69
59 25 41 48 64 79 62 26 87 86 94 30 43 54 26 98 61 38
85 00 02 24 67 85 88 10 34 01 54 53 23 77 33 11 19 68
01 46 87 56 19 19 19 43 70 25 24 29 48 22 44 81 35 40
42 41 25 10 87 27 77 28 05 90 73 03 95 46 88 82 25 02
03 57 14 03 17 80 47 85 94 49 89 55 10 37 19 50 20 37
18 95 93 40 45 43 04 56 17 03 34 54 89 91 69 02 90 72

La suite des nombres à deux chiffres de cette table doit être lue nombre après nombre et ligne après ligne.

**Figure 2.2** : Séquence de Nombres extraits de la table de Kendall et Babington Smith

Cette technique demeure très lourde à mettre en œuvre pour générer des suites de nombres aléatoires.

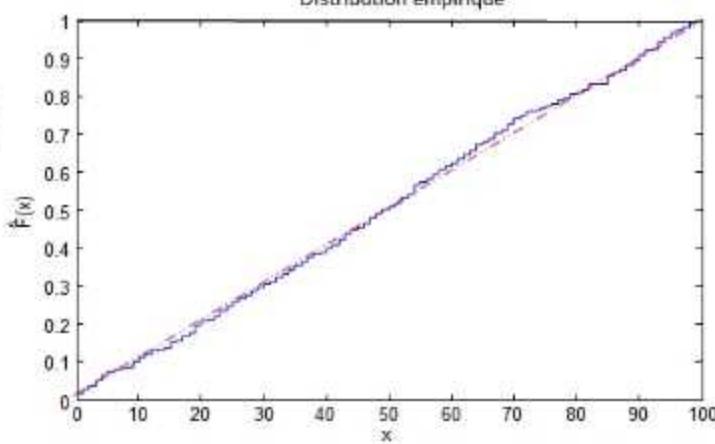
Nombres d'après la table de Kendall & Babington Smith



**Fig. 2.3 :** Comparaison des allures des données issues de tables et d'observation de la loi uniforme  
En notant  $x_1, \dots, x_j, x_{j+1}, \dots, x_n$  la séquence issue de la table de Kendall et Babington Smith, nous obtenons la distribution empirique

$$\hat{F}_{X,n}(x) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{]-\infty, x]}(X_i)$$

Distribution empirique



**Fig. 2.4 :** Comparaison de la distribution empirique des nombres tirés de la table de Kendall et Babington Smith et de la distribution uniforme

Malgré que les séquences de nombres tirés de ces tables peuvent être considérés comme parfaitement aléatoires (figures 2.3 et 2.4), l'emploi de cette approche par lecture directe des valeurs n'est adapté que pour générer des séquences de taille assez réduites. Cette méthode a été progressivement abandonnée avec l'avènement des calculateurs programmables.

- L'utilisation de procédures itératives : les nombres qui en résultent ne sont pas des suites de nombres aléatoires mais plutôt des suites de nombres pseudo-aléatoires. Ces procédures consistent en général en la construction de suites récurrentes appropriées. Le principal inconvénient de ce type de méthodes est qu'il donne malheureusement lieu à des suites périodiques. Leur avantage fondamental réside dans la rapidité du processus par lequel les séquences de nombres sont générées. En effet, comme nous allons le voir un peu plus loin leur structure algorithmique convient mieux au traitement informatique. Autrement dit, ces méthodes sont algorithmiquement reproductibles lorsqu'on connaît un germe et faciles à mettre en œuvre sur machine dans un langage de programmation.

## 2.2 Générateurs procéduraux de nombres pseudo-aléatoires

### 2.2.1 Générateur pseudo-aléatoire

L'élément central des méthodes de simulation stochastique tourne autour d'une transformation déterministe de variables aléatoires réelles uniformément distribuées sur l'intervalle  $]0,1[$ . La définition suivante précise le sens d'un générateur de nombres pseudo-aléatoires. C'est en fait un algorithme déterministe qui décrit comment obtenir les termes successifs de la séquence des nombres générés.

#### Définition

*Un générateur de nombres pseudo-aléatoires est une transformation déterministe  $\mathbf{d}$  de  $]0,1[$  dans  $]0,1[$ , telle que, pour toute valeur initiale  $x_0$  (Seed) et tout  $n$ , la suite :*

$$\pi_n = \{x_0, \mathbf{d}(x_0), \mathbf{d}^2(x_0), \dots, \mathbf{d}^n(x_0)\}$$

*ait le même comportement statistique qu'un  $n$ -échantillon de la loi uniforme  $\mathcal{U}_{]0,1[}$ .*

Ainsi, le traçage graphique des calculs issus de cette procédure permet d'écrire :

On fixe  $x_0$ .

On pose ensuite :

$$x_1 = \mathbf{d}(x_0),$$

$$x_2 = \mathbf{d}(x_1),$$

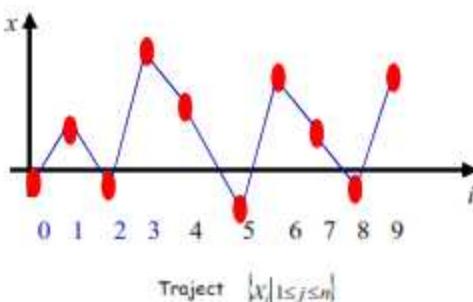
$$x_3 = \mathbf{d}(x_2),$$

...,

$$x_9 = \mathbf{d}(x_8),$$

... etc.

Visualisation



### Exemple

Dans cet exemple, dans l'idée de l'exploration des données simuler nous allons visualiser 500 nombres (Fig. 2.5 ) générés au hasard entre 0 et 1 par une méthode procédurale.

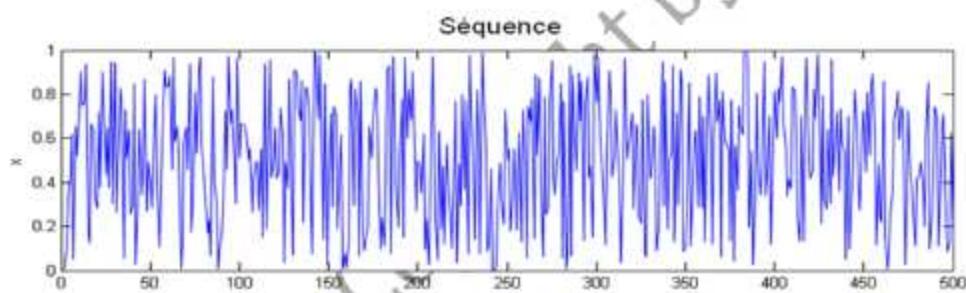


Fig. 2.5 : Suite uniforme de 500 nombres compris entre 0 et 1

Dès que nous avons des valeurs, nous pouvons classer ces données en 100 classes (par exemple) et représenter l'histogramme (Fig. 2.6 ) des classes de valeurs.

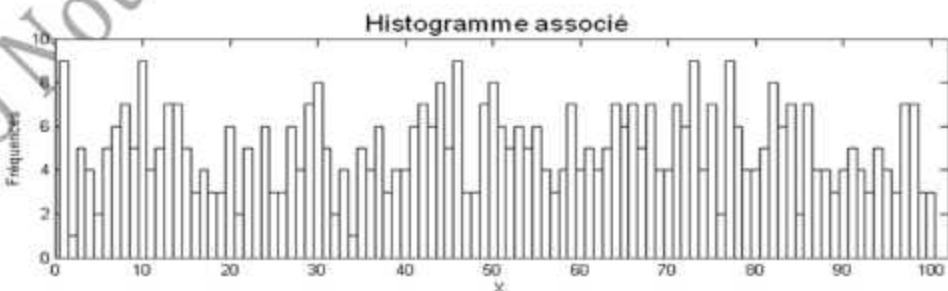


Fig. 2.6 : Histogramme associé des fréquences pour 100 classes de valeurs

## 2.2.2 Méthode du carré médian de Von-Neumann

C'est Von-Neumann qui a mis au point l'une des premières méthodes pour générer des nombres entre 0 et 999. Le principe de son algorithme est simple :

### Algorithme 2.1

- 1) On choisit un nombre  $x_0$
- 2) On élève ce nombre au carré  $x_0^2$
- 3) Les trois chiffres du milieu forment  $x_1$  et on réitère.

### Exemple :

- $x_0 = 131 \Rightarrow x_0^2 = 1 | 716 | 1$
- $x_1 = 716 \Rightarrow x_1^2 = 51 | 265 | 6$
- $x_2 = 131 \Rightarrow x_2^2 = 7 | 022 | 5$
- $x_3 = 22 \Rightarrow x_3^2 = 0 | 48 | 4$
- $x_4 = 48 \Rightarrow x_4^2 = 0 | 230 | 4$
- ... etc.

### Exercice

Ecrivez d'abord en pseudocode l'algorithme puis réaliser dans le langage Octave un programme qui implémente cette méthode. Enfin, visualiser les données pour différentes valeurs du germe (seed) et faites vos conclusions sur la méthode.

En effet, la visualisation des séquences de données engendrées par la méthode du carré médian permet de tirer au clair un certain nombre de remarques.

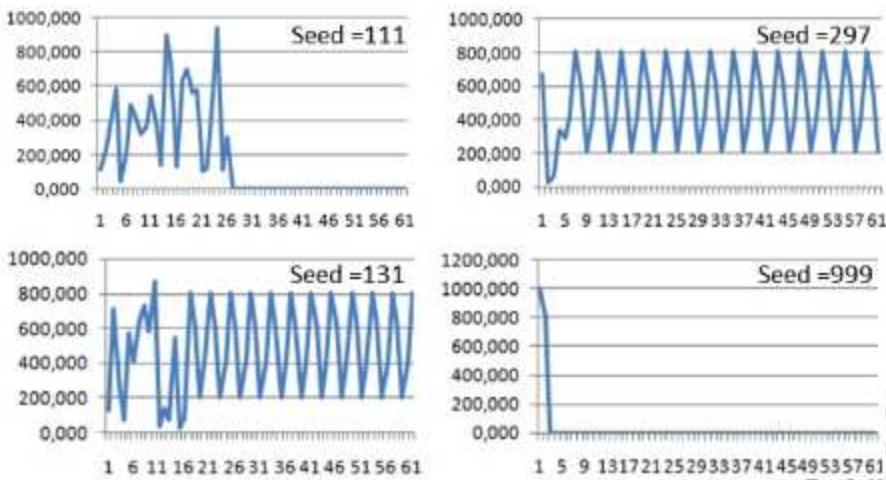


Fig. 2.7 : visualisation de séquences de longueur 60 pour différents choix du germe.

On remarque ainsi que la méthode de von-Neuman est fortement périodique voir même stationnaire pour certaines valeurs du germe.

### 2.2.3 Méthodes congruencielles

La méthode du carré médian de von-Neumann a été remplacée en 1951 par des méthodes congruencielles mises au point en premier par Lehmer. Pour formuler cette méthode, la transformation  $d$  de l'algorithme général (ci-avant) est en fait obtenue à partir d'une succession de transformations.

#### Définition

*On appelle générateur congruenciel, un générateur de nombres pseudo-aléatoires dont la transformation déterministe  $d$  est obtenu à partir de la normalisation de la fonction*

$$D(x) = (ax + b) \bmod m$$

*et on pose*

$$\begin{cases} x = [m \times u] \\ d(u) = \frac{D([m \times u])}{m} \in ]0,1[ \text{, pour } u \in ]0,1[ \end{cases}$$

Ainsi, la terminologie suivante est d'usage.

## Terminologie

$a$  : multiplicateur,

$b$  : incrément,

$m$  : base modulo (ou modulus),

$x_0$  : valeur initiale, germe, racine ou encore seed.

## Exemples

- La relation modulo associe à un nombre entier, une classe modulo dont le représentant est le reste de la division de ce nombre par la base modulo. Ainsi, à titre d'exemples,

$$17 \bmod 5 = 2 ; 81 \bmod 2 = 1 ; 251 \bmod 3 = 0$$

- On considère le générateur congruenciel de paramètres  $a = 1103515245$ ,  $b = 12345$ ,  $m = 2^{32}$  et  $x_0 = 4294967291$ . Ainsi, les premiers éléments calculés par le générateur sont :

$$x_1 = 3072370712$$

$$x_2 = 112136817$$

$$x_3 = 809803991$$

$$x_4 = 1938042308$$

$$x_5 = 2664522925$$

:

## Questions d'implémentation et de réflexion

- Représentez en pseudocode l'algorithme du schéma congruenciel considéré dans l'exemple 2 (précédent).
- Ecrivez le programme correspondant en Octave et produisez la visualisation des données générées pour différentes tailles  $N$  des séquences (par exemple : 50, 100, 500, 1000, ...).
- Représentez enfin les histogrammes des différentes séquences chaque fois pour un nombre  $m$  différent des classes (10, 50, 100, 500, ...).
- Que remarquez-vous lorsque la taille  $N$  de la séquence est « très grande ».
- Que remarquez-vous pour de « petites valeurs » du nombre  $m$  de classes de l'histogramme.

6) Même question dans le cas où le rapport du nombre  $m$  de classes par le nombre  $N$  des données générées est assez grand devant 0.1 (i.e.  $m/N \gg 1/10$ ).

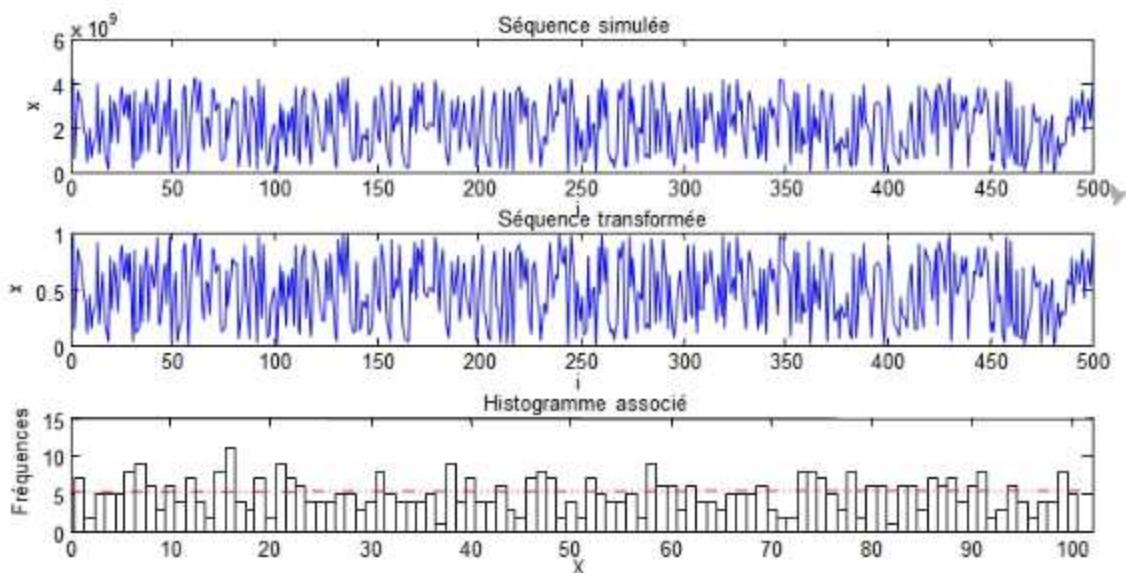


Fig. 2.8 : Exemples d'affichages pour l'exercice précédent

### Définitions

i. On appelle période d'une suite  $(x_n)_{n \in \mathbb{N}}$  le plus petit entier  $T$  tel que

$$x_{n+kT} = x_n; \forall n, k \in \mathbb{N}$$

ii. On appelle période  $T$  d'une méthode congruencielle, la période de la suite  $(x_n)_{n \in \mathbb{N}}$  qu'elle définit.

#### 2.2.4 Types de générateurs congruenciel

La terminologie exprimée par les définitions suivantes est d'usage pour classer les différents générateurs primitifs.

### Définitions

On distingue trois types de générateurs congruenciel :

Générateurs additifs :

$$x_n = x_{n-1} + x_{n-k} \bmod m, \text{ avec } k > 0.$$

### Générateurs multiplicatifs :

$$x_n = a x_{n-1} \bmod m, \text{ où } a \text{ et } m \text{ sont premiers entre eux.}$$

### Générateurs mixtes :

$$x_n = a x_{n-1} + b \bmod m, \text{ où } a \text{ et } m \text{ sont premiers entre eux.}$$

### **Remarques**

- 1) En base modulo  $m$ , il y a au plus  $m$  nombres différents modulo  $m$ . Ainsi, toutes les méthodes congruencielles ont une période inférieure ou égale à leurs modulus.
- 2) Les méthodes additives sont peu utilisées puisque les deux autres méthodes mènent à de meilleures propriétés statistiques.
- 3) Si on démarre avec un germe  $x_0$  ( $0 < x_0 < m$ ), la condition «  $a$  et  $m$  premiers entre eux » garantit pour la méthode multiplicative associée

$$\forall n \in \mathbb{N}^* : x_n = a^n x_0 \neq 0 \bmod m$$

- 4) De manière générale, pour pouvoir générer des séquences de très grande taille, il faut choisir une grande valeur du modulus  $m$ .

## 3 Critères d'existence d'une période maximale

Un bon générateur doit permettre de générer des séquences les plus longues possibles. Les théorèmes qui suivent permettent d'obtenir une séquence de longueur maximale.

### 3.1 Cas d'une base modulo générale

Hull et Dobell (1962) ont montré que la période des séquences générées peut atteindre la base modulo  $m$  (résultats admis).

#### **Théorème 1** (Admis)

Le générateur  $x_i = a x_{i-1} + b \bmod m$  a une période égale à  $m$ ssi,

- 1)  $b$  et  $m$  sont premiers entre eux,
- 2) Pour tout facteur premier  $p$  de  $m$ ,  $a - 1$  est un multiple de  $p$ ,
- 3) Si  $m$  est un multiple de 4, alors  $a - 1$  est un multiple de 4.

## Remarques

Ce théorème constitue en soi un critère de construction de générateurs congruenciel aditifs de périodes maximales. En plus, si  $a = 5 \bmod 8$  et  $b = x_0 \bmod 4$ ; alors en posant  $y_i = (x_i - b)/4$ , la suite obtenue à partir du générateur  $y_i = a y_{i-1} + b(a-1)/4 \bmod (m/4)$  vérifie les conditions d'applicabilité du théorème 1.

Par ailleurs, on remarque que ce théorème est directement applicable dans le cas des générateurs multiplicatifs, si on pose  $b = 0$ .

### 3.2 Cas d'une base modulo un nombre premier

Le théorème suivant donne des conditions nécessaires et suffisantes de maximisation de la période si la base modulo est un nombre premier.

#### Théorème 2 (Admis)

- 1) La période  $T$  de la suite  $x_{n+1} = a x_n \bmod m$  (un entier premier) est un diviseur de  $m-1$ .
- 2) La période  $T = m-1$  si et seulement si  $a$  est une racine primitive de  $m-1$ . i.e.,  $a \neq 0$  et pour tout facteur premier  $p$  de  $m-1$ , nous avons  $a^{\left(\frac{m-1}{p}\right)} \neq 1 \bmod m$ .
- 3) Si  $a$  est une racine primitive de  $m-1$  et si les entiers  $k$  et  $m-1$  sont premiers entre eux, alors,  $a^k \bmod m$  est également une racine primitive de  $m-1$ .

## Remarques

- 1) La taille de codage sur machine détermine une limite supérieure pratique pour le choix de la base modulo  $m$ , i.e.  $m < 2^\alpha$ .
- 2) Cela suggère de prendre  $m$  comme le plus grand nombre premier de la forme  $m = 2^\alpha - 1$ . Ces nombres sont appelés « Nombres de Mersenne ».
- 3) Par exemple, sur une machine 32 bit, on doit coder les nombres entiers positifs, leurs opposés et le nombre 0. Ainsi,

$$m = 2^{31} - 1 \text{ i.e. } \alpha = 31$$

### Exemple 1 (Générateur Minimal Standard)

Le générateur multiplicatif défini par  $m = 2^{31} - 1$  (nombre de Mersenne) premier et  $a = 7^5 = 16807$  (racine primitive de  $m - 1$ ) possède d'après le théorème 2 une période maximale, égale à,

$$\begin{aligned} T &= m - 1 = 2^{31} - 2 \\ &= 2\,147\,483\,646 \\ &\approx 2,15 \times 10^9 \end{aligned}$$

### Exemple 2 (Générateur de Lehmer)

C'est une méthode congruentielle multiplicative avec  $m = 2^{31} - 1$  et  $a = 23$ . La période  $T$  de cette méthode est supérieure à  $m/4$ . En effet,

$$T = \frac{m-1}{2} = 1\,073\,741\,824 \text{ alors que } \frac{m}{4} = 5,3687 \text{ e+08}$$

Ce qui est conforme à la proposition 1 du théorème 2.

### 3.3 Cas d'une base modulo une puissance de 2

Le théorème suivant donne des conditions nécessaires et suffisantes de maximisation de la période si la base modulo est une puissance de 2.

#### Théorème 3 (Admis)

Le générateur  $x_i = a x_{i-1} \bmod m$  avec  $m = 2^\alpha$  (avec  $\alpha \geq 4$ ) a une période maximale  $T = m/4$ , si et seulement si,

- 1)  $x_0$  et  $m$  sont premiers entre eux,
- 2)  $a \bmod 8 \in \{3, 5\}$

## Remarques

- 1) Dans ce cas la base modulo n'est pas un nombre premier.
- 2) Dans ces conditions la base modulo du générateur multiplicatif est de la forme  $m = 2^\alpha$  et la période maximale est égale à  $T = \frac{m}{4} = 2^{\alpha-2}$ .

## 4 Critères d'ordre statistique et discrépance

Ici nous utilisons le terme de discrépance avec deux dimensions : une dimension philosophique qui révèle la divergence et la discordance entre deux perceptions et une dimension mathématique qui traduit le fait de remplir ou pas l'espace dans toutes les directions.

### 4.1 Mesures de discrépance

On s'intéresse à l'étude de l'uniformité de distribution de  $n$ -points dans l'hyper-cube unité, de dimension  $d$ , noté  $]0,1[^d$ .

#### Notations et définitions

On s'intéresse à la décomposition de la séquence générée en des groupes de  $k$  valeurs consécutives.  
La terminologie suivante est d'usage en la matière :

- On note  $u = (u^{(1)}, u^{(2)}, \dots, u^{(d)}) \in ]0,1[^d$  ( coordonnées d'un vecteur)
- $\forall u, v \in ]0,1[^d : u \leq v \Leftrightarrow u^{(i)} \leq v^{(i)}$  pour  $i = 1, 2, \dots, d$  (ordre partiel sur les vecteurs)
- $\forall u \in ]0,1[^d : [0, u] = \{w \in [0,1]^d \mid w \leq u\}$  (intervalle vectoriel)
- $\forall u \in ]0,1[^d : \text{vol}[0, u] = \prod_{i=1}^d u^{(i)}$  (volume d'un intervalle vectoriel)

Ainsi, dans le cas où  $d = 1$ , on s'attend à  $n$  points équidistants les uns des autres dans l'intervalle  $]0,1[$ .



Fig. 2.9 : Points uniformément répartis en dimension  $d = 1$ .

Dans le cas où  $d = 2$ , on considère un ensemble  $P$  de  $n$  points uniformément répartis (figure 2.10), tels que, on s'attend à ce que le nombre  $n_R$  de points de  $P$  situés dans le rectangle de la forme  $R = [a_1, b_1] \times [a_2, b_2]$  soit approximativement  $n_R = n \times \text{vol}(R)$ .

### Définition

*Nous dirons que :*

- $D(P, R) = |n \times \text{vol}(R) - |P \cap R||$  est l'écart de discrépance.
- $D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \{D(P, R)\}$  est la discrépance par rapport aux rectangles.
- $\delta(n, \mathcal{R}) = \inf_{\substack{P \subset ]0,1[^2 \\ |P|=n}} \{D(P, \mathcal{R})\}$  est la plus petite discrépance à  $n$  points.
- $\Delta(P)$  représente la finesse de recouvrement de discrépance (voir figure).

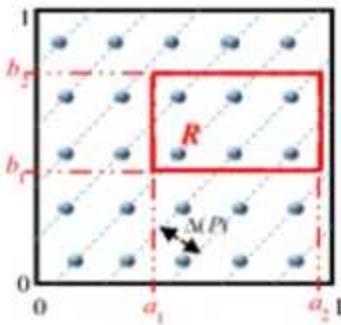


Fig. 2.10 : Représentation de la mesure de discrépance pour des points uniformément répartis en dimension  $d = 2$ .

Ainsi, la distance  $\Delta(P)$  entre les droites révélées par la structure de discrépance représente la finesse de recouvrement de discrépance.

### 4.2 Uniformité et calcul d'intégrales

#### Définition

Soit  $n \in \mathbb{N}^*$ , on dit que la séquence  $\{u_i \in ]0,1[ \mid 1 \leq i \leq n\}$  est uniformément répartie sur  $]0,1[$ , si pour tout sous-intervalle  $[a,b] \subset ]0,1[$ , on a,

$$\lim_{n \rightarrow +\infty} \left[ \frac{1}{n} \mid \{u_1, u_2, \dots, u_n\} \cap [a, b] \mid \right] = b - a$$

Les séquences uniformément réparties ont une propriété encore plus forte (qui n'est en fait pas difficile à prouver à partir de la définition précédente).

### Propriété (admise)

*Soient  $f$  une fonction de  $]0,1[ \rightarrow \mathbb{R}$  intégrable au sens de Riemann (bornée et presque sûrement continue pour la mesure de Lebesgue) et  $\{u_i \in ]0,1[ | 1 \leq i \leq n\}$  une séquence uniformément répartie sur  $]0,1[$ . Alors,*

$$\lim_{n \rightarrow +\infty} \frac{1}{n} \sum_{k=1}^n f(u_k) = \int_0^1 f(x) dx$$

## 4.3 Discrépance étoile

### Définition

*Soit  $n \in \mathbb{N}^*$ , on appelle mesure de discrépance étoile d'une séquence  $\{u_i \in ]0,1[^d | 1 \leq i \leq n\}$  la mesure définie par*

$$D_n^*(u_1, u_2, \dots, u_n) = \sup_{v \in [0,1]^d} \left| \frac{1}{n} \sum_{i=1}^n \mathbf{1}_{[0,v]}(u_i) - \text{vol}([0,v]) \right|$$

La mesure de discrépance est une mesure de non-uniformité d'une séquence de points placés dans l'hypercube  $]0,1[^d$ .

### Définition

*On dit que la suite  $\{u_n | n \in \mathbb{N}^*\}$  est uniformément répartie sur  $]0,1[^d$ , si sa mesure de discrépance étoile est asymptotiquement nulle, i.e.*

$$\lim_{n \rightarrow +\infty} D_n^*(u_1, u_2, \dots, u_n) = 0$$

Idéalement une séquence obtenue à partir d'un générateur de nombres pseudo-aléatoires devrait être distribuée uniformément dans l'intervalle  $]0,1[$ . Si cette propriété est réellement vérifiée, les paires  $(u_i, u_{i+1})$  devraient être uniformément réparties dans le carré  $]0,1[^2$ . Nous pouvons ainsi, proposer l'algorithme suivant pour la représentation graphique de la structure de discrépance.

### **Algorithme**

```
Proc Discrep_plan(u)
local x,y,N ;
N ← Longueur(u);
Pour i ← 1 à N – 1 Faire
    x[i] ← u[i];
    y[i]← u[i+1];
Fin_pour ;
Dessiner((x, y))
Fin_proc.
```

### **Remarques**

1. Cela n'est pas toujours le cas, il apparaît souvent des structures de discrépance.
2. Ce constat constitue le point faible principal de la méthode de congruence simple.

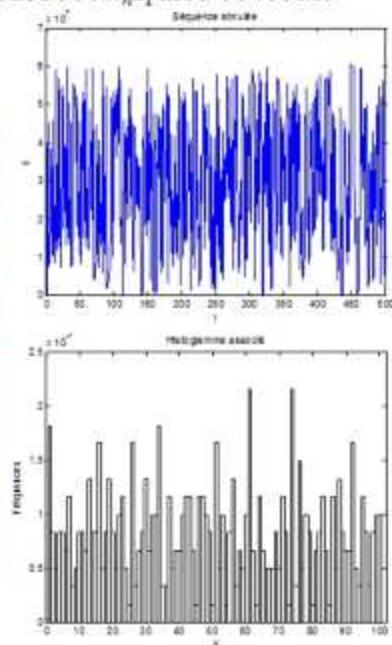
#### **4.4 Etude d'exemples**

Dans les exemples suivants, les indices statistiques de position et de variation sont calculés pour les séquences générées et différentes analyses d'adéquation visuelle sont menées. Le but étant d'analyser l'uniformité de la séquence. Ainsi, on représentera les trajectoires des séquences, puis on remarquera visuellement des classes plus ou moins uniformément remplies, ainsi que des structures de discrépances plus ou moins apparentes.

- 1) Générateur avec une forte structure de discrépance grossière :

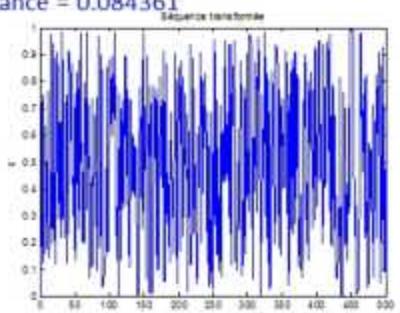
$$x_n = 30233086 x_{n-1} \bmod 60466169$$

Faible périodicité  
Forté discrépance  
Très faible finesse de recouvrement



Moyenne = 0.48933

Variance = 0.084361

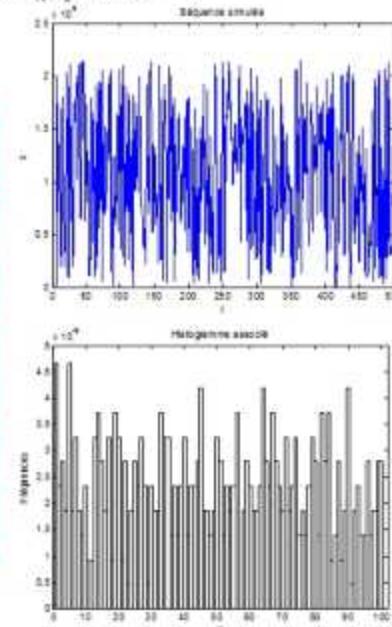


Finesse du  
recouvrement de  
discrépance

## 2) Générateur avec une forte structure de discrépance fine

$$x_n = 23 x_{n-1} \bmod 2^{31}-1$$

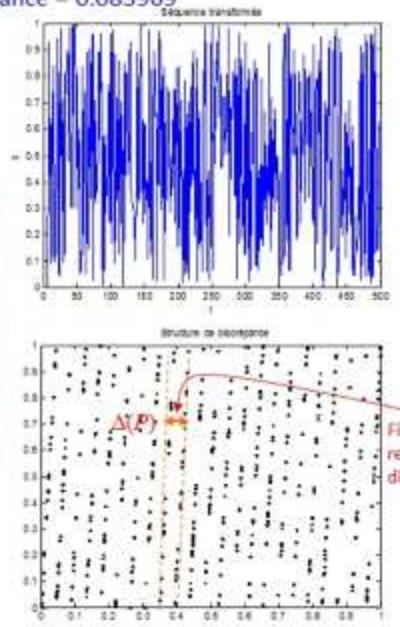
Faible périodicité  
Discrépance modérée  
Assez bonne finesse de recouvrement



Moyenne = 0.4891

Variance = 0.083969

Générateur de Lehmer

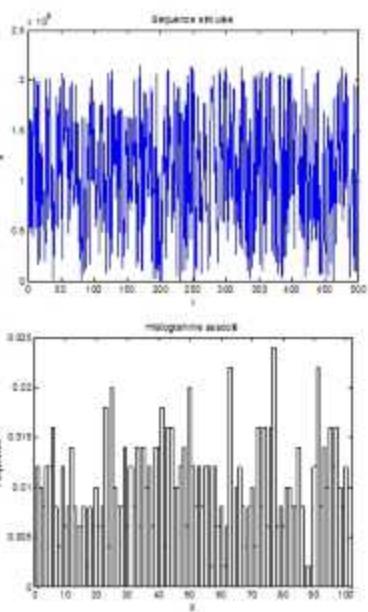


Finesse du  
recouvrement de  
discrépance

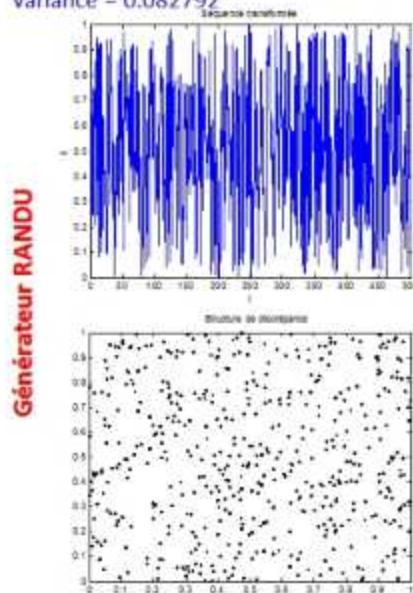
## 3) Générateur où la structure de discrépance n'est pas évidente (ou très apparente)

$$x_n = 65539x_{n-1} \bmod 2^{31}$$

Faible périodicité  
Discrépance modérée  
Assez bonne finesse de recouvrement



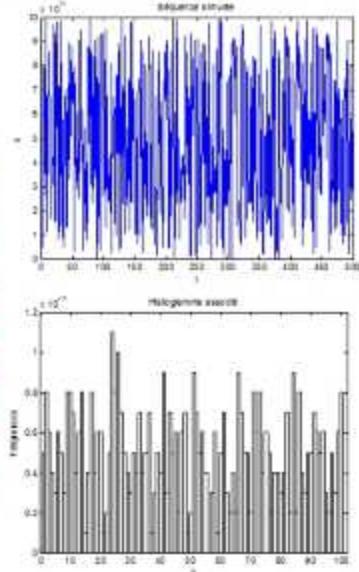
Moyenne = 0.5065  
Variance = 0.082792



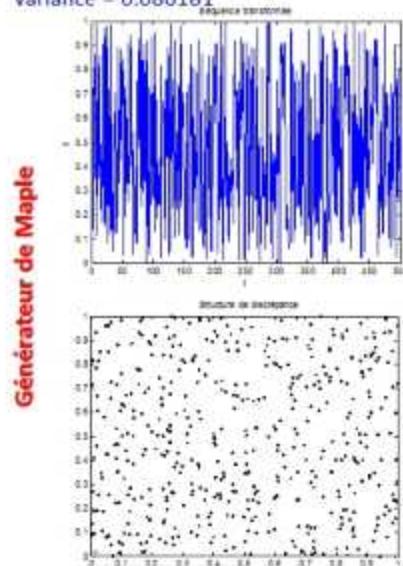
#### 4) Générateur où la structure de discrépance n'est pas très évidente

$$x_n = 427419669081 x_{n-1} \bmod 10^{12} - 11$$

Faible périodicité  
Faible discrépance  
Bonne finesse de recouvrement



Moyenne = 0.48621  
Variance = 0.086181



## 5 Amélioration des générateurs congruenciers

### 5.1 Méthode du registre à décalage avec rétroaction linéaire

La méthode du registre à décalage avec rétroaction linéaire (*Linear Feedback Shift Register Method*) est une généralisation des méthodes congruencielles classiques.

$$\begin{cases} x_n = a_0 + a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k} \bmod m \\ u_n = x_n / m \end{cases}$$

Pour calculer une occurrence à l'étape  $n$ , la mémoire doit se rappeler des  $k$  étapes précédentes  $(x_n; x_{n-1}, x_{n-2}, \dots, x_{n-k})$ . Comme tous les modèles congruenciel, toute séquence qui découle de ce modèle devient forcément périodique à partir d'un certain rang. Ainsi, remarquons que si  $k = 1$ , on retrouve une méthode congruencelle additive classique. Dans le cas général, l'espace des états est  $(\mathbf{Z}/m\mathbf{Z})^k$ , avec  $|(\mathbf{Z}/m\mathbf{Z})^k| = m^k$ . Ces modèles sont utilisés en cryptographie pour concevoir des séquences de nombres pseudo-aléatoires. Les paramètres du modèle doivent être calibrés de manière à ce que l'allure de la séquence qui en découle exhibe la plus grande période possible tout en maintenant des coûts de calcul et de stockage raisonnables.

#### Exemple

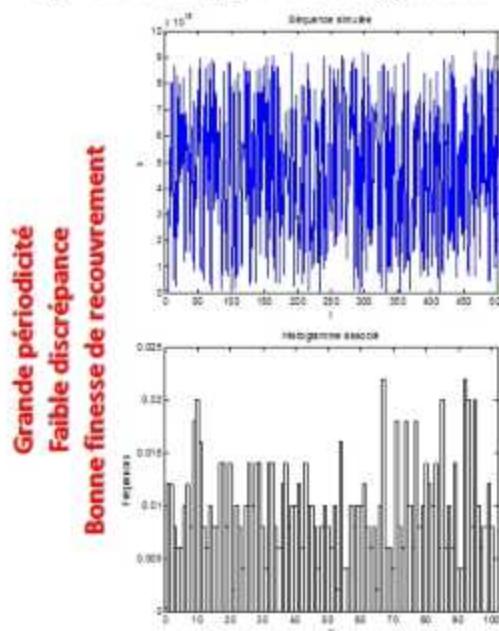
Le modèle (de **Hamlili**) suivant a de bonnes propriétés statistiques. Comme nous pouvons le remarquer son coût (en termes de nombre d'opérations et de stockage mémoire) est acceptable et visiblement il ne montre pas de structure de discrépance 2D

$$\begin{cases} x_n = 9648451 x_{n-5} + 747391 x_{n-3} \bmod 2^{31} - 1 \\ u_n = x_n / (2^{31} - 1) \end{cases}$$

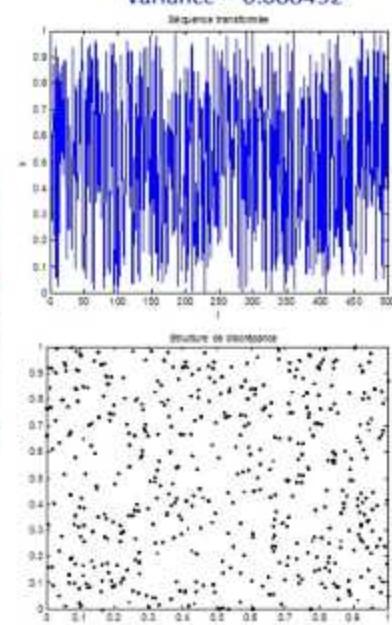
En tant qu'instanciation du modèle de registre à décalage avec rétroaction linéaire, les paramètres non-nuls de ce modèle sont

$$\begin{cases} a_3 = 747391 \\ a_5 = 9648451 \end{cases} \text{ et } m = 2^{31} - 1$$

$$x_n = 9648451x_{n-5} + 747391x_{n-3} \bmod 2^{31} - 1$$



Moyenne = 0.50437  
Variance = 0.088452



L'intérêt de ce type d'amélioration est qu'elle reste très performante de point de vue temps de calcul.

## 5.2 Méthodes congruencielles vectorielles

L'amélioration des générateurs de nombres pseudo-aléatoire est souvent utilisée dans le sens de prolonger leurs périodes afin de pouvoir générer des séquences « indépendantes » toujours plus grandes.

L'approche que nous considérons ici consiste à coupler des générateurs congruencielles différents

$$\begin{cases} \mathbf{X}_n = \mathbf{A}\mathbf{X}_{n-1} \bmod \mathbf{m} \\ U_n = \mathbf{C}\mathbf{X}_{n-1} \bmod 1 \end{cases}$$

où  $\mathbf{X}_k$ ,  $\mathbf{m}$  et  $\mathbf{C}$  sont des vecteurs et  $\mathbf{A}$  est une matrice. Ainsi, plusieurs générateurs évoluent en parallèle puis ils sont composés. Ce type de méthodes conduit à une très grande famille de générateurs qui peut être utilisée pour générer des vecteurs uniformes de dimensions quelconques et combinant différents générateurs. En effet, l'espace des états de tels générateurs est

$\prod_{i=1}^k (\mathbf{Z}/m_i \mathbf{Z})$  ; donc, avec  $\left| \prod_{i=1}^k (\mathbf{Z}/m_i \mathbf{Z}) \right| = \prod_{i=1}^k m_i$  états. Ainsi, si les paramètres du modèle sont

soigneusement choisis, en prenant les bases modulos  $m_i$  (pour  $i = 1..k$ ) très grandes, on peut espérer aboutir à des séquences de très grandes périodes.

Les générateurs vectoriels sont basés sur le calcul matriciel linéaire et donc très adaptés au calcul graphique (GPU) et aux ordinateurs vectoriels et multiprocesseurs.

### Exemple

A titre d'exemple le générateur vectoriel proposé par L'Écuyer a de bonnes propriétés

$$\begin{cases} x_n = 40014 x_{n-1} \bmod 2^{31} - 85 \\ y_n = 40692 y_{n-1} \bmod 2^{31} - 249 \\ z_n = (x_{n-1} - y_{n-1}) \bmod 2^{31} - 86 \end{cases}$$

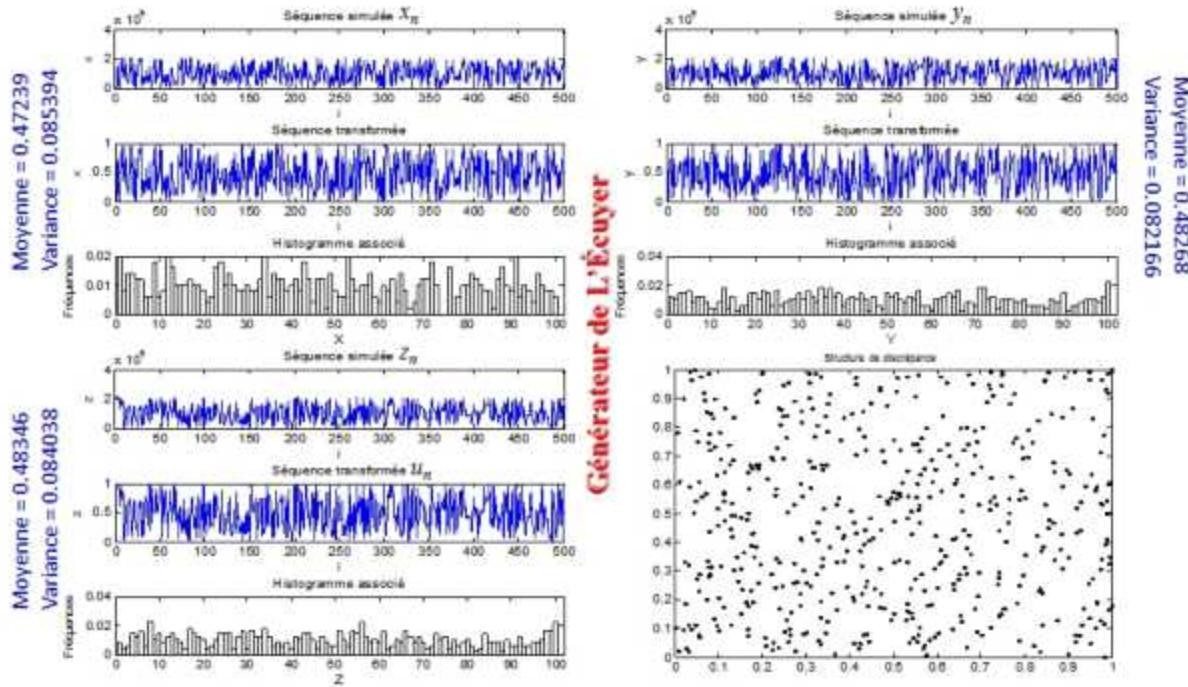
Dans ce cas,

$$\mathbf{X}_n = \begin{pmatrix} X_n \\ Y_n \end{pmatrix}, \quad \mathbf{m} = \begin{pmatrix} 2^{31} - 85 \\ 2^{31} - 249 \end{pmatrix} \text{ et } \mathbf{C} = \frac{1}{2^{31} - 86} \times (1, -1).$$

A l'étape  $n$ , la mémoire devra se rappeler de  $(x_n, y_n, u_n; x_{n-1}, y_{n-1})$ . L'espace des états est alors  $(\mathbf{Z}/m_1 \mathbf{Z}) \times (\mathbf{Z}/m_2 \mathbf{Z})$ , avec  $|(\mathbf{Z}/m_1 \mathbf{Z}) \times (\mathbf{Z}/m_2 \mathbf{Z})| = m_1 \times m_2$  états. La période de cette méthode

$$T = \frac{(m_1 - 1)(m_2 - 1)}{2} \cong 2.205 \times 10^{18}$$

de l'ordre de 2 milliards de milliards.



Pour ce type de générateurs plus la taille des groupes mélangés augmente, plus la structure de discrépance disparaît (i.e. les paires successives obéissent à une distribution uniforme). Par contre, le temps de calcul est beaucoup plus important que dans le cadre de l'amélioration par la méthode du registre à décalage avec rétroaction linéaire.

## 6 Approche Monte-Carlo pour l'évaluation d'intégrales

Une des applications classiques de la génération des nombres au hasard consiste au calcul d'intégrales lorsque le calcul numérique de leurs valeurs par des méthodes usuelles pose des problèmes. Ce type d'applications est communément connu sous le nom de méthodes d'intégration Monte-Carlo. Elles sont surtout utilisées pour l'évaluation d'intégrales multidimensionnelles complexes en calcul bayésien.

Pour faciliter l'exposé des méthodes d'intégration **Monte-Carlo**, nous proposons de faire les calculs dans le cas unidimensionnel, puis en déduire le calcul multidimensionnel par les voies classiques.

## 6.1 Forme canonique

Soit  $g$  une fonction de variable réelle dans  $[0,1]$ , déterminons le calcul de l'intégrale

$$I = \int_0^1 g(x) dx$$

Pour calculer la valeur numérique de  $I$ , la méthode Monte-Carlo consiste à prendre une variable aléatoire  $U$  uniformément distribuée sur  $[0,1]$  et d'établir que

$$I = \int_{\mathbb{R}} g(x) \mathbf{1}_{[0,1]}(x) dx = E[g(U)]$$

Si  $U_1, \dots, U_k$  est un  $k$ -échantillon de  $U$ , il découle des résultats établis en probabilité que les variables  $g(U_1), \dots, g(U_k)$  sont indépendantes, identiquement distribuées et ayant pour moyenne commune  $I$ . De même moyenne égale à l'intégrale  $I$  et de même variance  $\sigma^2 = \text{var}[g(U)]$ .

Ainsi, par la loi faible des grands nombres, nous pouvons écrire

$$\sum_{i=1}^k \frac{g(U_i)}{k} \xrightarrow[k \rightarrow +\infty]{\text{P}} E[g(U)] = I$$

Autrement dit, lorsque  $k$  est assez grand,  $\hat{I}_k$  constitue un estimateur (approximation) de  $I$  au sens de la convergence en probabilité de la suite des  $(\hat{I}_k)_{k \in \mathbb{N}}$  vers  $I$ . i.e.

$$\forall \varepsilon > 0 : \lim_{k \rightarrow +\infty} \Pr\left(\left|\hat{I}_k - I\right| \geq \varepsilon\right) = 0$$

Ainsi, en générant un assez grand nombre  $k$  de réalisations pseudo-aléatoires  $u_1, \dots, u_k$  dans l'intervalle  $[0,1]$ , on est pratiquement sûre que  $I$  est la moyenne des  $g(u_1), \dots, g(u_k)$ .

Malheureusement, à moins que la taille de l'échantillon ne soit effectivement très grande, il est très difficile de savoir si la théorie asymptotique est suffisamment précise pour nous permettre d'interpréter nos résultats en toute confiance.

### Algorithme

```
Proc Integrale_simple(k)
local i,s,u ;
s ← 0 ;
Pour i ← 1 à k Faire
    u ← RND() ;
    s ← s + g(u)
Fin_pour
Retourner(s/k)
Fin_proc.
```

### Question d'implémentation

Réaliser dans Octave un programme qui implémente cette méthode.

#### 6.2 Cas général des deux bornes finies

Dans le cas général des bornes finies, pour le calcul de l'intégrale

$$I = \int_a^b g(x) dx \text{ avec } -\infty < a < b < +\infty$$

nous aurons besoin d'utiliser le changement de variables

$$u = \frac{x-a}{b-a}$$

soit  $du = dx / (b-a)$ . Ainsi, on obtient

$$\begin{aligned} I &= \int_0^1 (b-a) g[a + (b-a)u] du = \int_0^1 h(u) du \\ &= E[h(U)] \end{aligned}$$

où  $h(u) = (b-a) g[a + (b-a) u]$

Ainsi, on se ramène au cas de la forme canonique du paragraphe 5.1.

### Question d'implémentation

Ecrivez en pseudocode l'algorithme qui permet le calcul Monte-Carlo de l'intégrale  $I$  dans ce cas précis.

### 6.3 Autres cas unidimensionnels

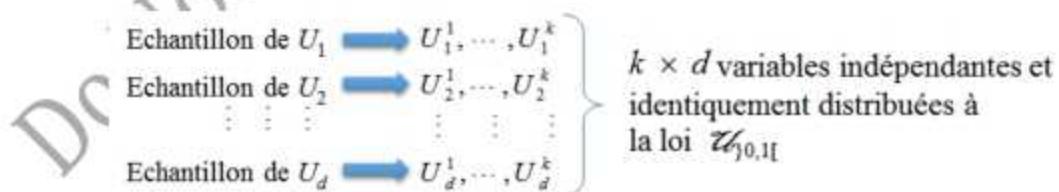
Des changements de variables adéquats permettent de ramener les calculs au cas canonique du paragraphe 5.1 lorsque l'une ou même les deux bornes de l'intégrale sont infinies. Cette question fait l'objet de l'exercice E 2.7 ci-après.

### 6.4 Calcul des intégrales multiples

On se propose dans ce paragraphe d'examiner le calcul d'intégrales multidimensionnelles par la méthode Monte-Carlo. Soit  $g : \mathbb{R}^d \rightarrow \mathbb{R}$ , comme dans le cas d'intégration simple, on peut écrire

$$\begin{aligned} I &= \int_0^1 \int_0^1 \cdots \int_0^1 g(x_1, x_2, \dots, x_d) dx_1 dx_2 \cdots dx_d \\ &= E[g(U_1, U_2, \dots, U_d)] \end{aligned}$$

où  $(U_1, U_2, \dots, U_d)$  est un vecteur de  $d$  variables aléatoires i.i.d. de loi uniforme sur  $[0, 1]$ . Pour chacune de ces variables on génère un échantillon de taille  $k$ , ce qui permet d'obtenir



Comme ces variables sont toutes i.i.d., l'intégrale multiple  $I$  peut être estimée par application de la loi faible des grands nombres en utilisant la statistique

$$\hat{I}_k = \frac{1}{k} \sum_{i=1}^k g(U_1^i, \dots, U_d^i)$$

### Question d'implémentation

Ecrivez en pseudocode l'algorithme qui permet le calcul Monte-Carlo de l'intégrale  $I$  dans le cas multidimensionnel.

### 6.5 Autours de la vitesse de convergence de l'intégration Monte-Carlo

D'abord, la vitesse de convergence de l'intégration Monte-Carlo est en  $1/\sqrt{n}$  quel que soit la dimension du domaine d'intégration. Ainsi, si on considère le cas de l'intégration en dimension 1 ( $d = 1$ ), la méthode de calcul Monte-Carlo est beaucoup plus lente que les méthodes d'intégration déterministe classiques (exemple : la méthode des trapèzes).

Par contre, dans le cas multidimensionnel ( $d > 1$ ), la vitesse de convergence de l'intégration Monte-Carlo est en  $1/\sqrt{n}$  alors que pour les méthodes déterministes, la vitesse de convergence est de

l'ordre de  $1/n^{(s/d)}$  où  $d$  désigne la dimension d'intégration et  $s$  représente l'ordre de régularité de la fonction (i.e., tel que les dérivées d'ordre inférieurs ou égaux à  $s$  sont toutes bornées). Ainsi, la méthode Monte-Carlo devient compétitive aux méthodes d'intégration déterministes dès que la dimension d'intégration dépasse le seuil  $d > 2s$ . La méthode Monte-Carlo utilisant une fonction d'importance est intéressante pour le calcul des intégrales multiples d'ordres élevés.

### 7 Simulation d'un vecteur de variables aléatoires

Soit  $X = (X_1, X_2, \dots, X_k)$  un vecteur aléatoire à valeurs dans  $\mathbb{R}^k$  de densité conjointe  $f(x_1, x_2, \dots, x_k)$ . Si on peut construire des simulateurs pour les densités marginales conditionnelles  $f_{i|x_1, x_2, \dots, x_{i-1}}(x_i)$  des composants (variables aléatoires)  $X_i$  connaissant  $X_1 = x_1, X_2 = x_2, \dots, X_{i-1} = x_{i-1}$ :

$$f_{i|x_1, x_2, \dots, x_{i-1}}(x_i) = \frac{f_{\{i\}}(x_1, x_2, \dots, x_{i-1}, x_i)}{f_{\{i-1\}}(x_1, x_2, \dots, x_{i-1})}$$

où  $f_{(i)}$  est la fonction densité conjointe du vecteur  $(X_1, X_2, \dots, X_i)$ .

Ainsi, si des simulateurs des lois conditionnelles sont disponibles ou peuvent être déterminées, alors, d'ipso-facto, nous pouvons simuler alternativement les composantes du vecteur aléatoire  $X$ . Par ailleurs, cette méthode peut incontestablement être utilisée en considérant un ordre quelconque sur les composants du vecteur  $X$ .

## 8 Méthode d'intégration à l'aide d'une fonction d'importance

On cherche à calculer une intégrale multiple

$$I = \int_{\mathbb{R}^k} g(x) dx$$

### Définition

On appelle fonction d'importance  $f$  une densité de probabilité sur  $\mathbb{R}^k$  telle que  $\text{Supp}(g) \subseteq \text{Supp}(f)$ .

Autrement dit, si pour tout  $x \in \mathbb{R}^k$ ,  $f(x) = 0$ , alors également  $g(x) = 0$ . On peut écrire

$$I = \int_{\mathbb{R}^k} \frac{g(x)}{f(x)} f(x) dx = E_f \left( \frac{g(x)}{f(x)} \right)$$

Notons maintenant  $(X_1, X_2, \dots, X_n)$  une suite de vecteurs aléatoires indépendants et identiquement distribués à la loi de densité  $f$  ( $n$ -échantillon du caractère  $X$ ) et considérons la statistique

$$J_n = \frac{1}{n} \sum_{i=1}^n \frac{g(X_i)}{f(X_i)}$$

D'après la loi forte des grands nombres, on sait que la suite des variables aléatoires  $(J_n)_{n \in \mathbb{N}}$  converge presque sûrement vers  $I$ . Ainsi, lorsque  $n$  est assez grand,  $J_n$  fournit un estimateur de l'intégrale  $I$ . Il est possible de mesurer la précision de cette estimation à l'aide du théorème de

théorème de la limite centrale (si la variance de la variable aléatoire  $\xi(X) = \frac{g(X)}{f(X)}$  est finie),

autrement dit,

$$\Pr\left(\frac{|J_n - I|}{\sigma_{\xi(X)}/\sqrt{n}} < a\right) = \frac{1}{\sqrt{2\pi}} \int_{-a}^a \exp\left(-\frac{x^2}{2}\right) dx$$

où  $\sigma_{\xi(X)}$  est l'écart type de la variable  $\xi(X) = \frac{g(X)}{f(X)}$ .

La densité  $f$  utilisée étant largement arbitraire, la seule contrainte impliquant que le support de  $f$  contienne le support de la fonction  $g$  qu'on souhaite intégrer est imposée. La convergence est d'autant meilleure que l'écart-type  $\sigma_{\xi(X)}$  est petit. On peut donc chercher la densité  $p$ , qui rende  $\sigma_{\xi(X)}$  le plus petit possible.

### Proposition

La variance  $\sigma_{\xi(X)}$  de  $\xi(X) = \frac{g(X)}{f(X)}$  est minimale pour la densité

$$f^*(x) = \frac{\|g(x)\|}{\int_{\mathbb{R}^k} \|g(x)\| dx}$$

### Preuve

En effet,

$$\begin{aligned}\sigma_{\xi(X)}^2 &= \int_{\mathbb{R}^k} \xi(x)^2 f(x) dx - I^2 \\ &= \int_{\mathbb{R}^k} \frac{g(x)^2}{f(x)} dx - I^2\end{aligned}$$

Pour minimiser l'écart-type  $\sigma_{\xi(x)}$ , il faut rendre minimum la quantité  $\int_{\mathbb{R}^k} \left( \frac{g(x)}{f(x)} \right)^2 f(x) dx$ , or d'après l'inégalité de Cauchy-Schwartz, pour le produit scalaire  $\langle g_1, g_2 \rangle = \int_{\mathbb{R}^k} g_1(x) g_2(x) ds$  sur  $L^2(\mathbb{R}^k)$  (espace vectoriel des fonctions de carré intégrable), nous avons

$$\begin{aligned} \int_{\mathbb{R}^k} \frac{\|g(x)\|}{f(x)} dx &= \int_{\mathbb{R}^k} \left( \frac{g(x)}{f(x)} \right)^2 f(x) dx \\ &\leq \left( \int_{\mathbb{R}^k} \left| \frac{g(x)}{f(x)} \right|^2 dx \right)^{1/2} \left( \int_{\mathbb{R}^k} |f(x)|^2 dx \right)^{1/2} \end{aligned} \quad (*)$$

Or, si  $f(x) = \frac{\|g(x)\|}{\int_{\mathbb{R}^k} \|g(s)\| ds}$  nous avons, d'une part,

$$\int_{\mathbb{R}^k} \frac{\|g(x)\|^2}{f(x)} dx = \left( \int_{\mathbb{R}^k} \|g(x)\| dx \right)^2$$

et

$$\begin{cases} \int_{\mathbb{R}^k} |f(x)|^2 dx = 1 \\ \int_{\mathbb{R}^k} \left\| \frac{g(x)}{f(x)} \right\|^2 dx = \left( \int_{\mathbb{R}^k} \|g(x)\| dx \right)^2 \end{cases}$$

d'autre part. Ainsi, (\*) devient

$$\int_{\mathbb{R}^k} \left\| \frac{g(x)}{f(x)} \right\|^2 f(x) dx = \left( \int_{\mathbb{R}^k} \|g(x)\| dx \right)^2$$

Pour cette densité la borne minimale est atteinte, ce qui démontre la proposition. Ce résultat n'est pas directement utilisable, car pour pouvoir le faire, il faudrait d'abord connaître la valeur exacte

de  $L$ . Il montre cependant que l'écart-type est d'autant plus petit que la densité  $f(x)$  ressemble à la fonction  $f(x)$ .

## 9 Exercices de récapitulation

**E 2.1** Soient le générateur multiplicatif défini par  $x_0 = 7$  et  $x_n = 7x_{n-1} \bmod 138$ .

- Simuler à l'aide d'Octave les 500 premières valeurs de tel générateur.
- Tracez la trajectoire de cette séquence de donnée.
- Visualisez la distribution des valeurs à l'aide de l'histogramme des fréquences relatives à 50 classes de même portée.
- Mettez en évidence la structure de discrépance de ce générateur.

**E 2.2** On considère le générateur congruenciel défini par  $u_{n+1} = 25u_n + 16 \bmod 256$ .

- Ecrire en Octave (ou Matlab) une fonction qui implémente ce générateur.
- Que se passe-t-il dans chacun des cas lorsque  $u_0 = 10$  ?  $u_0 = 11$  ?  $u_0 = 12$  ?  $u_0 = 111$  ?
- Représentez graphiquement la visualisation de la structure de discrépance correspondante.

**E 2.3** Implémentez en Octave (ou Matlab) et étudiez graphiquement (à l'image de ce qui a été produit dans les exemples des pages 7 et 8) les générateurs congruencIELS multiplicatifs très célèbres suivants de paramètres respectifs  $m$  et  $a$  :

- Générateur Minimal Standard (introduit par Lewis en 1969) :  $m = 2^{31} - 1$  et  $a = 75$
- Générateur de SAS :  $u_0 = 1$   $m = 2^{31} - 1$  et  $a = 397204094$ .
- Générateurs de Texas Instrument :
  - $m = 2^{31} - 85$  et  $a = 40014$
  - $m = 2^{31} - 249$  et  $a = 40692$ .
- Faites vos remarques, compte-tenu du cours, sur des séquences de longueurs 500 pour chacun de ces générateurs et pour des valeurs initiales  $u_0$  de votre choix entre 1 et  $m - 1$ .

**E 2.4** Nous considérons le générateur congruencIEL définit par  $u_{n+1} = 31415821u_n + 1 \bmod 108$  et  $u_0 = 1$ .

- Ecrivez en Octave (ou Matlab) la fonction qui implémente ce générateur.

- b) Générer la séquence des 20 premiers termes de ce générateur.
- c) Que remarquez-vous à propos du dernier chiffre des nombres générés ?
- d) Expliquez ce qui se passe.

**E 2.5** On définit les générateurs congruenciel multiplicatifs par

$$\begin{cases} x_n = 157 x_{n-1} \bmod 32363 \\ y_n = 146 y_{n-1} \bmod 31727 \\ z_n = 142 z_{n-1} \bmod 31657 \end{cases}$$

et la suite croisée de terme général

$$w_n = x_n + y_n + z_n \bmod 323621$$

- a) Déterminez la matrice **A**, le vecteur **C** et l'expression de la suite  $U_n$  qui formalisent ce modèle vectoriel sous la forme

$$\begin{cases} \mathbf{X}_n = \mathbf{A} \mathbf{X}_{n-1} \bmod \mathbf{m} \\ U_n = \mathbf{C} \mathbf{X}_{n-1} \bmod 1 \end{cases}$$

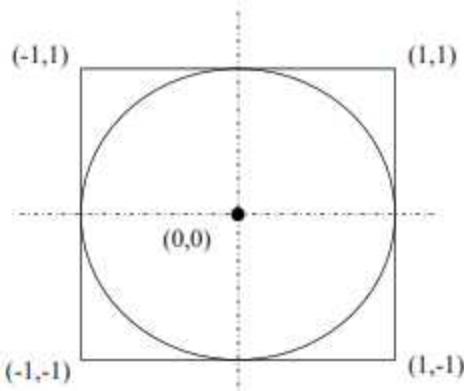
avec les vecteurs

$$\mathbf{X}_n \equiv \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix}; \quad \mathbf{m} \equiv \begin{pmatrix} 32363 \\ 31727 \\ 31657 \end{pmatrix}$$

- b) Implémentez en Octave (ou Matlab) et étudiez les propriétés du générateur vectoriel ainsi construit (à l'image de ce qui a été fait pour l'analyse du générateur de L'Écuyer défini dans ce chapitre).

**E 2.6** (Estimation du nombre  $\pi$ )

Soit  $(X, Y)$  un couple de variables aléatoires indépendantes et uniformément distribuées à l'intérieur du carré **Car** représenté dans la figure I.1 et considérons la probabilité que le point  $(X, Y)$  soit restreint à varier à l'intérieur du disque **Dis** inscrit de rayon 1.



**Figure 2.10 :** Cercle inscrit dans un carré

Sur la base de ces éléments de modélisation donner un algorithme qui permet de calculer une valeur approchée de  $\pi$ .

**E 2.7** Ecrire en pseudocode les algorithmes qui permettent de calculer les intégrales suivantes et trouver dans chacun des cas la fonction  $h$  qui permet de ramener l'intégrale considérée sous une forme canonique :

- a)  $I = \int_a^b g(x) dx$  avec  $-\infty < a < b < +\infty$
- b)  $I = \int_0^{+\infty} g(x) dx$
- c)  $I = \int_0^1 \int_0^1 \cdots \int_0^1 g(x_1, x_2, \dots, x_n) dx_1 dx_2 \cdots dx_n$

**E 2.8** Dans chacun des trois cas suivant, calculez la valeur exacte de l'intégrale  $I$  puis comparez cette valeur à la valeur obtenue par simulation Monte-Carlo. En fait grâce à une visualisation graphique, il faut montrer comment la valeur calculée par simulation converge vers la valeur exacte de l'intégrale lorsqu'on fait varier la taille de la séquence des nombres aléatoires ayant servi au calcul Monte-Carlo de cette intégrale :

- a)  $I = \int_0^1 e^x dx$
- b)  $I = \int_0^1 \exp(e^x) dx.$
- c)  $I = \int_0^{+\infty} \int_0^x \exp[-(x+y)] dy dx.$

**E 2.9** Montrez comment pourrait-on calculer à l'aide de la méthode Monte-Carlo l'intégrale

$$I = \int_0^1 \log(1 + x^2) \exp(-x^2) dx \text{ (où la fonction « log » indique le logarithme népérien)}$$

Ensuite, proposez en pseudocode un algorithme qui fait ce calcul, puis produisez un programme en Octave pour évaluer la valeur de cette intégrale.

Do Not Copy - Copyright by Ali Hamlili

## Chapitre 3 : Simulation de variables aléatoires réelles discrètes

### 1 Notions de simulation de variables aléatoires réelles discrètes

On peut partager les variables aléatoires réelles de différentes façons. Une des manières possibles pour ce faire est de les différencier à partir de l'ensemble de leurs valeurs possibles. Autrement dit, de l'ensemble des valeurs qu'elles peuvent charger, c'est-à-dire, les valeurs pour lesquels la fonction de probabilité induite par la variable aléatoire n'est pas nul. Un tel ensemble est appelé support de la variable aléatoire. Ainsi, on peut discriminer des variables aléatoires comme étant discrètes si leur support est discret et des variables aléatoires comme étant continues si leur support est continu.

De plus, en se basant sur ce même type de critère de discrimination, on peut distinguer parmi les variables aléatoires réelles discrètes deux sous classes, à savoir, celles dont le support est fini et celles dont le support est infini. En effet, ce qui nous pousse vers une telle distinction, c'est que de point de vue implémentation informatique, le cas d'une variable aléatoire discrète à support fini présente un intérêt évident compte tenu du stockage des valeurs du support. Pour le cas de variables aléatoires discrètes réelles à support discret infini, nous allons montrer qu'en tolérant une certaine erreur en probabilité, on peut encore ramener l'implémentation informatique de ce cas au cas d'une v.a.r. discrète à support fini.

### 2 Méthode de transformation inverse

Cette méthode repose sur l'inversion de la fonction de répartition de la v.a.r. qu'on souhaite simuler.

## Définition

Soit  $X$  une variable aléatoire discrète de fonction de probabilité induite

$$p_X(x) = \Pr(X = x) = \Pr[X^{-1}\{x\}]$$

Nous appelons support de  $X$  l'ensemble des valeurs chargées par la loi de probabilité  $p_X$ , i.e.

$$\text{Supp}(X) = \{x \in \mathbb{R} \mid p_X(x) \neq 0\}$$

### 2.1 Principe de construction

Nous souhaitons simuler le comportement probabiliste d'une variable aléatoire réelle discrète  $X$  qui prend ses valeurs dans l'ensemble  $\chi = \{x_i \mid i \in \mathbb{N}\}$ , i.e.  $\text{Supp}(X) \subseteq \chi$ .

La loi de probabilité induite par  $X$  étant caractérisée par :

$$(1) \quad p_i = \Pr(X = x_i) \text{ pour } i \in \mathbb{N} \text{ et } \sum_{i \geq 0} p_i = 1$$

pour des raisons d'implémentation algorithmique nous allons construire un ordre «  $\prec$  » sur  $\chi = \{x_i \mid i \in \mathbb{N}\}$  défini par

$$(2) \quad x_i \prec x_j \Leftrightarrow p_i \leq p_j$$

Maintenant, nous considérons une variable aléatoire  $U$  (continue) uniformément distribuée sur son support  $]0,1[$  qu'on note  $U \sim \mathcal{U}_{]0,1[}$  et on définit la variable aléatoire  $\tilde{X}$  par

$$(3) \quad \tilde{X} = \begin{cases} x_0 & \text{si } U \leq p_0 \\ x_1 & \text{si } p_0 < U \leq p_0 + p_1 \\ \vdots & \\ x_j & \text{si } \sum_{i=0}^{j-1} p_i = F(x_{j-1}) < U \leq F(x_j) = \sum_{i=0}^j p_i \\ \vdots & \end{cases}$$

La simulation de  $X$  par la méthode de transformation inverse repose sur le résultat suivant.

**Proposition 3.1**

*Les variables aléatoires  $X$  et  $\tilde{X}$  ont la même loi de probabilité.*

**Preuve** (En exercice !)

Ainsi pour simuler  $X$ , il suffit de simuler  $\tilde{X}$  à partir de l'algorithme (3) qui a servi pour sa construction.

2.2 Implémentation, en pseudo-code, d'un simulateur d'une v.a.r. discrète à domaine fini  
2.2.1 Générateur de la variable

**Algorithme**

```
Proc SimulDisFini(p,x)
Local i,u,F;
n ← length(x);
i ← 1;
F ← p[1];
u ← Rnd();
Tant_que (F < u) ∧ (i < n) faire
    i ← i+1 ;
    F ← F+p[i];
Fin_tant_que;
Retourner(x[i])
Fin_proc.
```

2.2.2 Générateur d'une séquence de réalisations

**Algorithme**

```
Proc DisFini_sequence(N,p,x)
local i , seq;
```

```

Pour i  $\leftarrow$  1 à N Faire
    seq[i]  $\leftarrow$  SimulDisFini(p,x)
Fin_pour ;
Afficher(seq) ;
Fin_proc.

```

### 2.3 Exemple d'implémentation d'une v.a.r. discrète à support fini

$X$  représente le résultat d'une expérience de réussite/échec avec  $\Pr\{X = 1\} = p$  est la probabilité de réussite et  $\Pr\{X = 0\} = 1 - p$  est la probabilité d'échec.

#### 2.3.1 Algorithme du générateur

Comme dans une loi de Bernoulli la réussite constitue l'événement préféré l'algorithme de simulation d'une telle loi va s'écrire comme suit.

##### Algorithme

**Proc** Ber(p)

**Local** x,u:

u  $\leftarrow$  Rnd();

**Si** u < p **Alors** x  $\leftarrow$  1

**Sinon** x  $\leftarrow$  0

**Fin\_si**;

**Retourner**(x)

**Fin\_proc.**

#### 2.3.2 Algorithme de génération de la séquence

##### Algorithme

**Proc** DisFini\_sequence(N,p,x)

**local** i , seq ;

**Pour** i  $\leftarrow$  1 à N **Faire**

```

seq[i] ← SimulDisFini(p,x)

Fin_pour ;

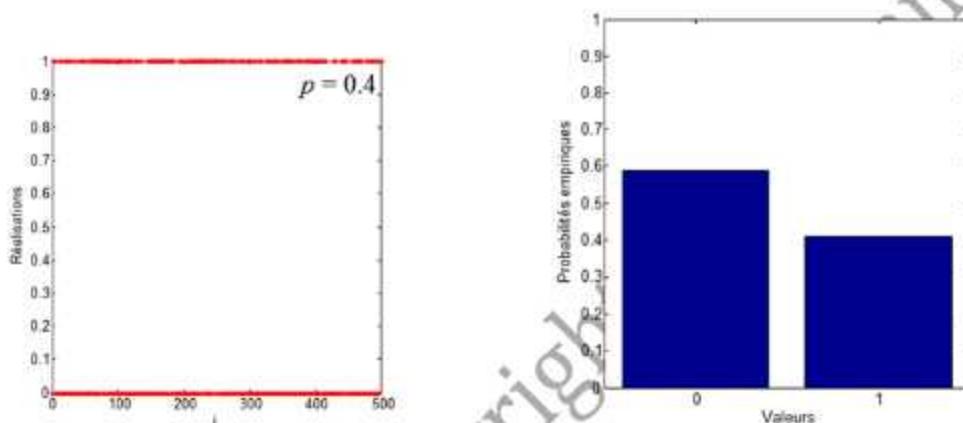
Afficher(seq)

Fin_proc.

```

### 2.3.3 Application

Pour  $p = 0.4$ , la mise-en-œuvre de cette procédure permet de visualiser les résultats correspondants.



**Fig. 3.1 :** Visualisation des résultats de simulation d'une variable aléatoire de Bernoulli

### 2.4 Implémentation d'un générateur pour une v.a. discrète à domaine infini

Dans ce cas, l'implémentation informatique repose sur le résultat suivant.

#### Proposition

Soit  $X$  une variable aléatoire discrète à domaine infini. Alors, elle admet un simulateur  $\tilde{X}$  à domaine fini.

#### Preuve

Comme la fonction de répartition  $F_X$  de  $X$  tend vers 1 à l'infini, i.e.

$$\lim_{n \rightarrow +\infty} \Pr(X \leq n) = \lim_{n \rightarrow +\infty} F_X(n) = 1$$

i.e. Pour tout  $\varepsilon > 0$  assez petit, il existe entier naturel  $N_\varepsilon$  tel que

$$\Pr(X > N_\varepsilon) < \varepsilon$$

Il suffit de déterminer le plus petit entier naturel  $N_0$  vérifiant cette relation. De fait, pour cette valeur  $N_\varepsilon$ , l'événement  $\{X > N_0\}$  est pratiquement impossible. i.e. Pour  $\varepsilon > 0$  assez petit,  $X$  ne peut pratiquement jamais atteindre des valeurs supérieures à  $N_0$ . Donc, il suffit de considérer le simulateur de la restriction de la v.a.  $X$  au domaine fini  $\{0, 1, 2, \dots, N_0\}$ .

#### 2.4.1 Algorithme de recherche de $N_0$

Donnons les algorithmes dans le cadre de simulation d'une loi de support égal à  $N$ . Pour simuler une v.a.  $X$  qui suit une loi de Poisson de paramètre  $\lambda > 0$ , on pose

$$\forall k \in \mathbf{N} : \Pr(X = k) = e^{-\lambda} \frac{\lambda^k}{k!}$$

A des fins d'optimisation du code, nous introduisons une implémentation récursive. Ainsi, il suffit de montrer par récurrence que

$$\forall k \in \mathbf{N} : \Pr(X = k+1) = \frac{\lambda}{k+1} \times \Pr(X = k)$$

#### Algorithme

```
Proc Simul_Poisson(lam,eps)
Local i,F,pr,x,p;
pr ← exp(- lam); i ← 0; p[1] ← pr; F ← pr;
Tant_que F < 1-eps Faire
    i ← i+1; pr ← pr*lam/i; p[i] ← pr;
    x[i] ← i; F ← F+pr
Fin_tant_que;
N0 ← i;
SimulDisFini(N0,p,x)
Fin_proc.
```

#### 2.4.2 Algorithme de simulation d'une séquence

Toujours, nous considérons le cadre de simulation de la loi de Poisson.

##### Algorithme

```
Proc Poisson_sequence(N,lambda,epsilon)
local i, seq;
Pour i ← 1 à N Faire
    seq[i] ← Simul_Poisson(lambda,epsilon)
Fin_pour;
Afficher(seq)
Fin_proc.
```

Pour  $\lambda = 2$  et un échantillon de taille  $N = 500$ , après implémentation des deux derniers algorithmes en Matlab, les résultats de la simulation peuvent être visualisés.

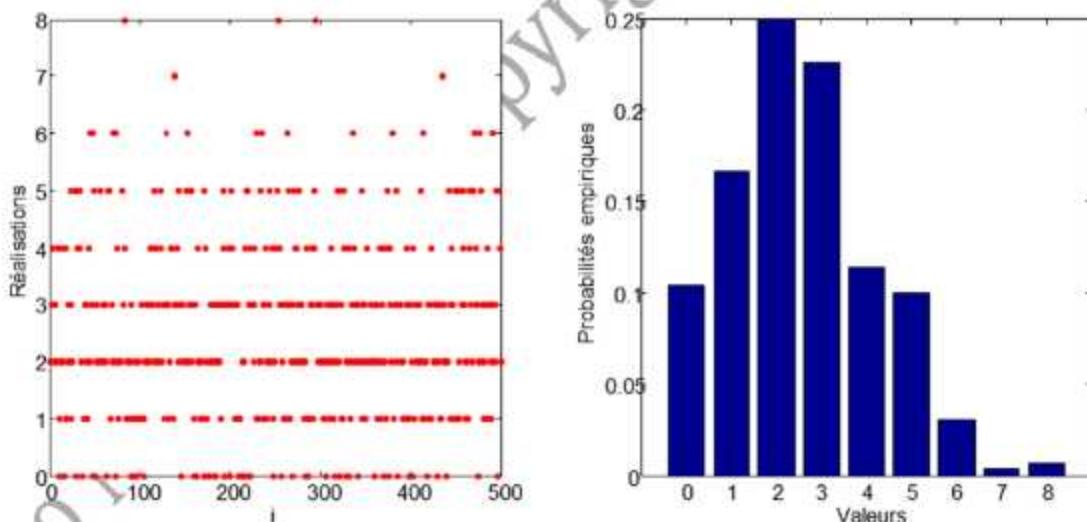
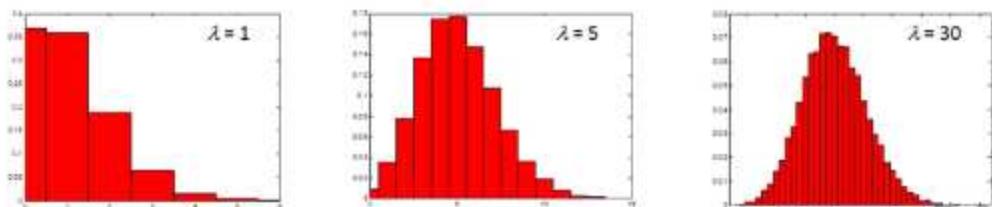


Fig. 3.2 Visualisation des résultats de simulation d'une variable aléatoire de Poisson

##### Remarque

Une observation intéressante dans le cas des modèles de Poisson est que lorsque la moyenne augmente, les propriétés de la distribution de Poisson sont proches de ceux de la distribution normale.

En conséquence pour des distributions de Poisson avec des moyennes supérieures à 30 et sous réserve de la précision requise, il est possible d'utiliser la distribution normale comme une approximation de la distribution de Poisson (Th. De la Limite Centrale).



### 3 Méthode de rejet/acceptation

Malheureusement la simulation par la méthode de transformation inverse n'est pas toujours le bon choix. Si pour une raison ou une autre nous n'arrivons pas à inverser la fonction de répartition de la variable que nous souhaitons simuler l'application de cette méthode se trouve compromise. Ce qui justifie le recours à d'autres mécanismes pour s'affranchir de cette contrainte et par la suite l'impossibilité d'arriver à un simulateur de la loi en question. La méthode de rejet/acceptation propose de simuler indirectement une v.a.r. discrète  $X$  à partir d'une autre v.a.r.  $Y$  discrète de contrôle qui peut être fixée par l'utilisateur pour atteindre les objectifs de simulation. Ce type de méthodes a été mis au point au début pour la reproduction de comportement d'automatismes.

Pour l'application de la méthode de simulation par rejet et acceptation, il faut toujours inspecter si les quatre hypothèses d'applicabilité de cette méthode sont vérifiées :

- $X$  et  $Y$  sont indépendantes, mais n'ont pas la même loi ;
- $Y$  admet le même ensemble des valeurs que  $X$  ;
- Il existe un simulateur simple de  $Y$  que nous noterons  $\tilde{Y}$  ;
- Il existe une constante  $c > 0$  telle que :  $\forall k \in \mathbb{N} : \Pr(X = k) \leq c \Pr(Y = k)$

Dans de tels conditions, en posant

$$\begin{cases} p_k = \Pr(X = k) \\ q_k = \Pr(Y = k) \end{cases}$$

Si  $U \sim \mathcal{U}_{[0,1]}$  est une variable aléatoire uniformément distribuée dans  $[0,1]$  et indépendante des variables  $X$  et  $Y$ , la méthode de rejet/acceptation propose un algorithme du générateur de la variable aléatoire  $X$  utilisant le simulateur  $\tilde{Y}$  de la variable  $Y$  qui s'écrit en trois étapes constitutives.

### Algorithme

**Etape 1 :** Générer un nombre aléatoire  $k \leftarrow \tilde{Y}$  ;

**Etape 2 :** Générer un nombre aléatoire  $u \leftarrow \tilde{U}$  ;

**Etape 3 :** Si  $u \leq \frac{p_k}{c q_k}$  Alors  $\tilde{X} \leftarrow k$  et Arrêter

Sinon Revenir à l'étape 1 .

### Proposition

Le simulateur  $\tilde{X}$  issu de la méthode de rejet/acceptation (algorithme précédent) a la même loi que  $X$ .

### Preuve

Soit  $U \sim \mathcal{U}_{[0,1]}$  une variable aléatoire uniformément distribuée sur  $[0,1]$  indépendante des variables  $X$  et  $Y$ . Pour tout entier naturel  $k$

$$\begin{aligned}\Pr(\tilde{X} = k) &= \Pr(Y = k \mid \text{Acceptation}) = \frac{\Pr(\{Y = k\} \cap \text{Acceptation})}{\Pr(\text{Acceptation})} = \frac{\Pr\left[\{Y = k\} \cap \left(U \leq \frac{p_k}{c q_k}\right)\right]}{\Pr(\text{Acceptation})} \\ &= \frac{\Pr(Y = k) \Pr\left(U \leq \frac{p_k}{c q_k}\right)}{\Pr(\text{Acceptation})} = \frac{q_k p_k}{\Pr(\text{Acceptation}) c q_k} = \frac{p_k}{c \Pr(\text{Acceptation})}\end{aligned}$$

Par ailleurs, d'après le théorème des probabilités totales

$$1 = \sum_k \Pr(\tilde{X} = k) = \frac{\sum_k p_k}{c \Pr(\text{Acceptation})} = \frac{1}{c \Pr(\text{Acceptation})}$$

Ainsi,  $\Pr(\tilde{X} = k) = p_k$ .

De là nous pouvons tirer l'écriture pseudocode finale de l'algorithme de la méthode de rejet/acceptation dans le cadre discret.

### Algorithme

```

Proc Simul_X(p, q)
Local k, u, x :
k  $\leftarrow$  Simul_Y(q);
u  $\leftarrow$  Rnd();
Tant que (((p[k]/q[k])/c) < u) Faire
    x  $\leftarrow$  k ;
    y  $\leftarrow$  Simul_Y(q);
    u  $\leftarrow$  Rnd();
Fin_tant que;
Retourner(x)
Fin_proc.
```

Pour générer une séquence de réalisations de loi de distribution de  $X$ , il suffit d'écrire la procédure suivante :

### Algorithme

```

Proc X_sequence(N,p,q)
local i , seq ;
Pour i  $\leftarrow$  1 à N Faire
    seq[i]  $\leftarrow$  Simul_X(p, q);
Fin_pour ;
```

Afficher(seq)

Fin\_poc.

### Exemple

Soient deux variables  $X$  et  $Y$  dont les lois sont complètement définies par

$k$	1	2	3	4
$p_k = \Pr(X=k)$	0.25	0.15	0.30	0.30

$k$	1	2	3	4
$q_k = \Pr(Y=k)$	0.20	0.30	0.25	0.25

On suppose qu'il n'est pas donné de simulateur de  $Y$  et pourtant on souhaite implémenter un simulateur par la méthode de rejet/acceptation pour simuler  $X$ . En effet, vu la complexité équivalente du problème de simulation des variables  $X$  et  $Y$ , au lieu d'implémenter d'abord un simulateur pour  $Y$  puis d'essayer d'implémenter  $X$  à partir de  $Y$  est plus difficile que de simuler  $X$  directement à partir de la méthode de transformation inverse. Comme il est demandé de simuler  $Y$  à partir de  $X$ , on va générer les valeurs de  $Y$  à partir d'un générateur qui ne suit pas nécessairement sa loi (celle de  $Y$ ).

### Algorithme

**Etape 1 :** Générer un nombre aléatoire  $u_1 \leftarrow \tilde{U}$  ;

**Etape 2 :** Poser  $k \leftarrow [4*u_1] + 1$  et  $\tilde{Y} \leftarrow k$  ;

**Etape 3 :** Générer un nombre aléatoire  $u_2 \leftarrow \tilde{U}$  ;

**Etape 4 :** Si  $u_2 \leq \frac{p_k}{c q_k}$  , Alors  $\tilde{X} \leftarrow k$  et Arrêter

Sinon revenir à l'étape 1.

### Remarque

Il faut remarquer que la vérification de la condition  $u_2 \leq \frac{p_k}{c q_k}$ , suppose que l'on ait accepté la

valeur  $k$  pour  $Y$ . Mais, du fait que dans la présente solution proposée  $k$  n'a pas été généré à partir d'un générateur de  $Y$ . L'impact de d'un tel choix, c'est que ce dernier va ralentir d'avantage la convergence de l'algorithme de simulation de  $X$  à partir de  $Y$ .

C'est pourquoi en informatique, on peut parfois accepter des choix qui ne représentent pas exactement l'idéal mathématique à condition qu'il existe un intérêt qui le justifie et que nous jugeons acceptables ou pouvons maîtriser (ou contourner) par un moyen ou un autre les impacts négatifs de nos choix.

Contrairement à ce que pourrait croire un néophyte, un bon informaticien doit être un bon mathématicien pour qu'il comprenne d'un côté les impacts négatifs inférés par des choix contraires aux idéaux théoriques et d'un autre côté pour qu'il sache contourner ces impacts s'il y a des avantages pratiques qui justifient de tels choix ! C'est l'affirmation que nous tentons de défendre dans cet exercice et que nous essaierons d'étayer davantage dans l'exercice E 3.4 ! Mais, cette idée ne devrait en aucun cas justifier toutes les maladresses qu'on pourrait rencontrer chez certains algorithmiciens et programmeurs !

## 4 Méthode de composition

Soient  $X_1$  et  $X_2$  deux v.a.r. discrètes données, le caractère  $X$  mélange des variables  $X_1$  et  $X_2$  s'écrit

$$X = \begin{cases} X_1 & \text{avec la probabilité } \alpha \\ X_2 & \text{avec la probabilité } 1 - \alpha \end{cases}$$

où  $\alpha \in ]0,1[$ . La simulation du caractère  $X$  par la méthode de composition des simulateurs  $\tilde{X}_1$  et  $\tilde{X}_2$  de  $X_1$  et  $X_2$  respectivement. Ainsi,

$$\tilde{X} = \begin{cases} \tilde{X}_1 & \text{avec la probabilité } \alpha \\ \tilde{X}_2 & \text{avec la probabilité } 1 - \alpha \end{cases}$$

Autrement dit, a priori nous pouvons décomposer une telle simulation en quatre temps.

#### Algorithme

**Etape 1 :** Simuler la v.a. aléatoire  $x_1 \leftarrow \tilde{X}_1$  ;

**Etape 2 :** Simuler la v.a. aléatoire  $x_2 \leftarrow \tilde{X}_2$  ;

**Etape 3 :** Générer un nombre aléatoire  $u \leftarrow U \sim \mathcal{U}_{[0,1]}$  ;

**Etape 4 :** Si  $u < \alpha$  Alors  $x_1$

Sinon  $x_2$  .

Soit en pseudocode

#### Algorithme

**Proc** Simul\_comp(alpha)

**Local** u,x:

**u**  $\leftarrow$  Rnd ();

**Si** ( $u < \alpha$ ) **Alors**

**x**  $\leftarrow$  Simul\_X1()

**Sinon**

**x**  $\leftarrow$  Simul\_X2()

**Fin\_si;**

**Retourner(x)**

**Fin\_proc.**

Pour générer une séquence, nous écrivons

#### Algorithme

**Proc** Comp\_sequence(N,alpha)

**local** i , seq ;

**Pour** i  $\leftarrow 1$  à N **Faire**

```

seq[i] ← Simul_comp(alpha)
Fin_pour ;
Afficher(seq)
Fin_proc.

```

### **Proposition**

*Le simulateur  $\tilde{X}$ , ainsi construit, a la même loi que  $X$ .*

### **Preuve**

En effet, d'après le théorème des probabilités totales,

$$\Pr(\tilde{X} = k) = \alpha \Pr(\tilde{X}_1 = k) + (1 - \alpha) \Pr(\tilde{X}_2 = k)$$

Comme  $\tilde{X}_1$  a la même loi que  $X_1$  et  $\tilde{X}_2$  a la même loi que  $X_2$ , alors,

$$\begin{aligned} \Pr(\tilde{X} = k) &= \alpha \Pr(X_1 = k) + (1 - \alpha) \Pr(X_2 = k) \\ &= \Pr(X = k) \end{aligned}$$

Ce qui montre le résultat recherché.

## 5 Exercices de récapitulation

**E 3.1** On considère la loi binomiale négative de paramètres  $n$  et  $p$  de fonction probabilité

$$\Pr(X = k) = \binom{k+n-1}{k} (1-p)^k p^n$$

- Produisez en pseudocode un simulateur pour une séquence de longueur  $m$  suivant la loi binomiale négative de paramètres  $n$  et  $p$ .
- Réécrivez ce programme en Octave.
- Visualisez l'histogramme de ce simulateur pour  $m = 1000$ ,  $n = 13$  et  $p = 0.3$ .

**E 3.2** On considère la loi hypergéométrique de paramètres  $a, b$  et  $n$  de fonction probabilité

$$\Pr(X = k) = \frac{\binom{a}{k} \binom{b}{n-k}}{\binom{a+b}{n}}$$

- a) Produisez en pseudocode un simulateur pour une séquence de langueur  $m$  suivant la loi binomiale négative de paramètres  $n$  et  $p$ .
- b) Ecrivez ce programme en Octave.
- c) Produisez l'histogramme de ce simulateur pour  $m = 1000$ ,  $a = 7$ ,  $b = 3$  et  $n = 13$ .

**E 3.3** Ecrivez en pseudo code puis en Octave une procédure qui génère une loi composée des deux lois vues dans les exercices 1 et 2 avec les coefficients de mixture respectivement associés à ces deux lois tels que  $\alpha = 0.3$  et  $1 - \alpha = 0.7$ .

**E 3.4** Soient  $X$  et  $Y$  deux variables aléatoires indépendantes qui prennent leurs valeurs dans  $\{1, 2, 3, 4\}$ . On pose :

$k$	1	2	3	4
$p_k = \Pr(X=k)$	0.25	0.15	0.30	0.30

Ecrivez un simulateur de la variable aléatoire  $X$  à partir d'un simulateur de  $Y$ , dans les deux cas de figure suivants :

- a) En utilisant la distribution de  $Y$

$k$	1	2	3	4
$q_k = \Pr(Y=k)$	0.20	0.30	0.25	0.25

- b) Puis en utilisant la distribution de  $Y$

$k$	1	2	3	4

$q_k = \Pr(Y=k)$	0.25	0.25	0.25	0.25
------------------	------	------	------	------

- c) Que remarquez-vous après l'implémentation et la mise en œuvre, sous Octave sur votre machine, de ces deux versions de la simulation par rejet et acceptation de la variable  $X$  ?

**E 3.5** On souhaite généraliser la méthode de simulation par composition pour la simulation d'un mélange de  $n$  variables aléatoires  $n \in \mathbb{N}^* \setminus \{1, 2\}$ . Ainsi, on considère  $n$  variables aléatoires discrètes  $X_1, X_2, \dots, X_n$  indépendantes,  $n$  nombres réels positifs  $\alpha_1, \alpha_2, \dots, \alpha_n$  tels que

$$\begin{cases} \alpha_k \in ]0, 1[ \text{ pour } k = 1..n \\ \sum_{k=1}^n \alpha_k = 1 \end{cases}$$

et on définit le mélange  $X$  par

$$X = \begin{cases} X_1 \text{ avec la proportion } \alpha_1 \\ X_2 \text{ avec la proportion } \alpha_2 \\ \vdots \\ X_n \text{ avec la proportion } \alpha_n \end{cases}$$

on suppose qu'il existe des simulateurs  $\tilde{X}_1, \tilde{X}_2, \dots, \tilde{X}_n$  pour les variables aléatoires  $X_1, X_2, \dots, X_n$  respectivement. On construit la variable aléatoire

$$\tilde{X} = \begin{cases} \tilde{X}_1 & \text{si } U \leq \alpha_1 \\ \tilde{X}_2 & \text{si } \alpha_1 < U \leq \alpha_1 + \alpha_2 \\ \vdots & \\ \tilde{X}_n & \text{si } \sum_{k=1}^{n-1} \alpha_i < U \leq 1 \end{cases}$$

- Montrez que  $\tilde{X}$  est un simulateur pour le mélange  $X$ .
- Traduisez en pseudocode cet algorithme.
- Que remarquez-vous ?

Do Not Copy. Copyright by Ali Hamlili

## Chapitre 4 : Simulation de variables aléatoires continues

### 1 Introduction

Dire que l'on simule une variable aléatoire revient à dire que l'on génère des observations de cette variable aléatoire. Jusqu'à maintenant, nous avons montré comment générer artificiellement des occurrences et des séquences de valeurs d'une variable aléatoire discrète. A chacune des techniques utilisées pour la génération de variables aléatoires discrètes correspond une méthode analogue pour la génération de variables aléatoires continues. L'objet de ce chapitre est de faire le tour de cette question dans le cadre de variables aléatoires continues.

### 2 Méthode de transformation inverse

#### 2.1 Principe de construction

Considérons une variable aléatoire continue  $X$  de fonction densité de probabilité  $f$  et de fonction de répartition  $F$ .

#### Définition

Soit  $X$  une variable aléatoire continu de fonction densité de probabilité  $f$ . Nous appelons support de  $X$  l'ensemble des valeurs chargés par la loi de probabilité induite par  $X$ . i.e.

$$\text{Supp}(X) = \{x \in \mathbb{R} | f(x) \neq 0\}$$

Etant donnée les propriétés de continuité à gauche et de monotonie d'une fonction de répartition  $F$  d'une variable aléatoire réelle continue, il existe un inverse à gauche de  $F$  en tout point du support. La définition suivante détermine implicitement cet inverse.

### Définition

Pour tout  $u \in [0,1]$ , nous définissons l'inverse à gauche de  $u$  et on note  $F^{-1}(u)$ , la plus petite valeur de  $x$  telle que  $F(x) = u$ . i.e.

$$F^{-1}(u) = \{x \in \text{Supp}(X) \mid F(x) = u\}$$

La méthode de transformation inverse pour la simulation de variable aléatoires continues se base sur la connaissance de la forme analytique de l'inverse  $F^{-1}$ .

### Proposition 3.1

Soit  $U$  une variable aléatoire uniformément distribuée sur l'intervalle  $[0,1]$ . Pour toute variable aléatoire continue  $X$  de fonction de répartition  $F$ , on définit la variable aléatoire

$$\tilde{X} = F^{-1}(U)$$

Alors, les variables aléatoires  $X$  et  $\tilde{X}$  ont la même loi de probabilité.

### Preuve

Notons  $F_{\tilde{X}}$  la fonction de répartition du simulateur  $\tilde{X}$ . Alors,

$$F_{\tilde{X}}(x) = \Pr(\tilde{X} \leq x) = \Pr[F_{\tilde{X}}^{-1}(U) \leq x] = \Pr[U \leq F_X(x)] = F_X(x)$$

Ce qui montre le résultat.

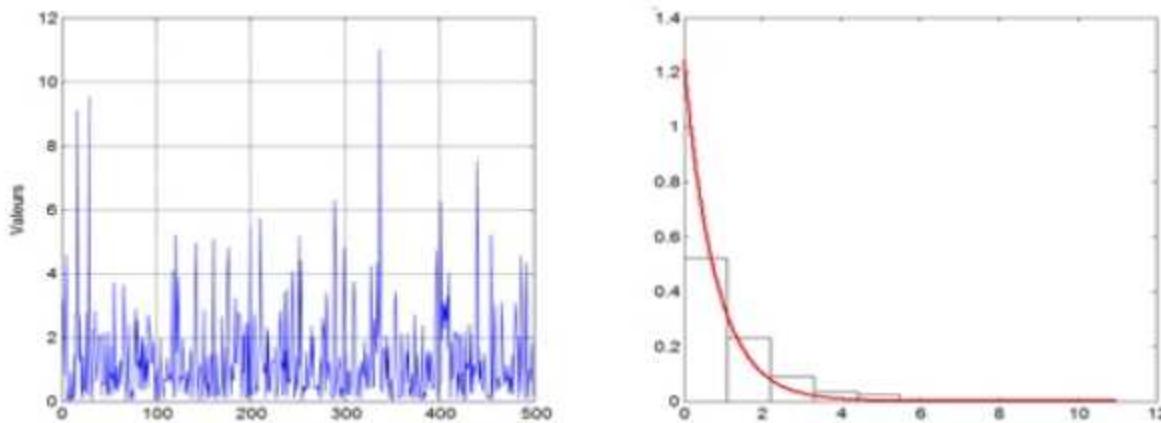
## 2.2 Exemples d'application

### Questions d'implémentation

- Déterminez la forme analytique de l'inverse de la fonction de répartition d'une loi exponentielle de paramètre  $\lambda > 0$ .
- Donnez en pseudocode l'algorithme qui permet de les simuler.

- c. Implémentez en Octave le programme qui permet de visualiser les trajectoires des séquences générées et l'adéquation des distributions empiriques des séquences à la distribution théorique de la loi de  $X$ .

A titre d'exemple la figure 3.1 donne la visualisation d'une séquence simulant une variable aléatoire exponentielle de paramètre 2 et comparaison des distributions empirique et théorique de cette loi.



**Fig. 3.1 :** Séquence de 500 nombres selon une loi exponentielle de paramètre  $\lambda = 2$  comparaison des distributions empirique et théorique de cette loi

### 3 Méthode de rejet/acceptation (variable de contrôle)

Nous supposons que nous connaissons une méthode pour générer une v.a.  $Y$  de densité  $g$ . Nous pouvons utiliser cette connaissance pour générer toute v.a.  $X$  de densité  $f$  telle que :

$$\exists c > 0, \forall y : \frac{f(y)}{g(y)} \leq c$$

Nous disons alors que la v.a.  $X$  est contrôlée par la v.a.  $Y$ . La simulation de  $X$  à partir de  $Y$  se fait selon l'algorithme suivant.

#### Algorithme 3.1

**Etape 1 :** Générer  $Y = y$  ayant pour densité  $g$ .

**Etape 2 :** Générer un nombre aléatoire  $U \leftarrow u$ .

**Etape 3 :** Si  $u \leq \frac{f(y)}{c g(y)}$ , prendre  $\tilde{X} \leftarrow y$ . Sinon revenir à l'étape 1.

### Proposition 3.2

- a. La variable aléatoire  $\tilde{X}$  construite par la méthode de rejet est de même loi que  $X$ .
- b. Le nombre d'itérations nécessaires pour la génération de la v.a.  $\tilde{X}$  est une v.a. géométrique de paramètre  $c$ .

#### Preuve

- a. Soit  $N$  la v.a. qui indique que la valeur  $y$  générée par  $\tilde{Y}$  est acceptable pour  $\tilde{X}$ . i.e.

$$N = \begin{cases} 1 & \text{si la valeur } y \leftarrow \tilde{Y} \text{ est acceptée pour } \tilde{X} \\ 0 & \text{sinon} \end{cases}$$

Notons  $U$  une variable aléatoire uniformément distribuée sur  $]0,1[$  indépendante des variables  $X$  et  $Y$ . Pour tout  $y \in \text{Supp}(X)$

$$\begin{aligned} f_{Y|N=1}(y) &= \int_0^{\frac{f(y)}{c g(y)}} f_{Y,U|N=1}(y, u) du \\ &= f_{Y|N=1}(y) \int_0^{\frac{f(y)}{c g(y)}} f_U(u) du \\ &= f_{Y|N=1}(y) \Pr\left(U \leq \frac{f(y)}{c g(y)}\right) \\ &= f_{Y|N=1}(y) \frac{f(y)}{c g(y)} \\ &= \frac{g(y)}{\Pr(N=1)} \frac{f(y)}{c g(y)} \\ &= \frac{f(y)}{c \Pr(N=1)} \end{aligned} \tag{*}$$

Par ailleurs, d'après le théorème des probabilités totales

$$\Pr(N=1) = \int_{\mathbb{R}} \Pr(N=1 | \tilde{Y}=y) g(y) dy \text{ avec } \Pr(N=1 | \tilde{Y}=y) = \Pr\left(U \leq \frac{f(y)}{c g(y)}\right) = \frac{f(y)}{c g(y)}$$

$$\text{d'où, } \Pr(N=1) = \int_{\mathbb{R}} \frac{f(y)}{c g(y)} g(y) dy = \frac{1}{c} \int_{\mathbb{R}} f(y) dy = \frac{1}{c}$$

Ainsi, l'équation (\*) devient  $f_{\tilde{X}}(y) = f(y)$ .

#### 4 Méthode de composition

La méthode de composition, comme dans le cas des discrets, permet de simuler la loi des mélanges. Ainsi, la simulation d'un mélange  $X$  de deux caractères aléatoires  $X_1$  et  $X_2$  indépendants dans des proportions  $\alpha$  et  $1-\alpha$  avec  $\alpha \in ]0,1[$  s'écrit

$$X = \begin{cases} X_1 & \text{avec la probabilité } \alpha \\ X_2 & \text{avec la probabilité } 1-\alpha \end{cases}$$

Si des simulateurs  $\tilde{X}_1$  et  $\tilde{X}_2$  de  $X_1$  et  $X_2$  respectivement sont connus, l'algorithme menant à la simulation de  $X$  à partir de  $\tilde{X}_1$  et  $\tilde{X}_2$  s'écrit alors

$$\tilde{X} = \begin{cases} \tilde{X}_1 & \text{avec la probabilité } \alpha \\ \tilde{X}_2 & \text{avec la probabilité } 1-\alpha \end{cases}$$

#### 5 Exercices de récapitulation

**E 4.1** Soit une variable aléatoire  $X$  qui suit une loi de fonction de répartition

$$F(x) = x^n \mathbf{1}_{]0,1]}(x) + \mathbf{1}_{[1,+\infty[}(x)$$

- a) Déterminez le support de  $X$ .
- b) Exprimez la forme analytique de l'inverse de cette fonction sur le support de  $X$ .
- c) Donnez en pseudocode l'algorithme qui permet de simuler  $X$ .

- d) Implémentez en Octave le programme qui permet de visualiser les trajectoires des séquences générées et l'adéquation des distributions empiriques des séquences à la distribution théorique de la loi de  $X$ .

**E 4.2** Soit  $X$  suivre une loi de  $\gamma(\lambda, n)$ , où  $\lambda \in \mathbb{R}_+^*$  et  $n \in \mathbb{N}^*$ . Sa fonction de densité peut s'écrire sous la forme

$$f(x) = \frac{\lambda e^{-\lambda x} (\lambda x)^{n-1}}{(n-1)!} \mathbf{1}_{\mathbb{R}_+}(x)$$

- a) Déterminez la fonction de répartition  $F$  associée à cette loi de probabilité.
- b) Est-ce que cette fonction de répartition est inversible ?
- c) Peut-on exprimer explicitement l'inverse de cette fonction ?
- d) Montrez que les propriétés mathématiques de la loi gamma permettent d'appliquer la méthode de transformation inverse.
- e) Ecrivez en pseudocode une procédure permettant de simuler cette variable aléatoire à l'aide de la méthode de transformation inverse.

**E 4.3** On souhaite simuler une v.a.  $X$  distribuée selon la loi  $\gamma\left(\frac{3}{2}, 1\right)$ .

- a) Ecrivez la fonction densité de cette loi.
- b) Calculez son moment d'ordre 1.
- c) Proposez une variable de contrôle adéquate pour la simulation de cette v.a.
- d) Ecrivez en pseudocode un algorithme adéquat à la simulation de cette v.a.
- e) Produisez en Octave un programme qui visualise les principales propriétés des séquences de variables aléatoire ainsi générées.

**E 4.4** On souhaite simuler la v.a.r.  $X$  ayant pour fonction de densité de probabilité

$$f(x) = C x (1-x)^3 \mathbf{1}_{[0,1]}(x)$$

par la méthode de rejet/acceptation.

- a) Déterminez le support de  $X$ .

- b) Déterminez la constante  $C$ .
- c) En utilisant comme variable de contrôle une v.a. uniforme adéquate, écrivez en pseudocode un algorithme pour la simulation la v.a.  $X$ .
- d) Ecrivez en Octave un programme qui permet de visualiser les séries générées.

**E 4.5** Nous voulons simuler une v.a. normale centrée et réduite  $X$  en utilisant une variable de contrôle  $Y$  qui suit une loi de Laplace de densité

$$g(x) = C e^{-|x|}$$

- a) Déterminez le support de  $X$ .
- b) Déterminez la constante  $C$ .
- c) Montrez que la méthode de rejet/acceptation est applicable.
- d) Ecrivez en pseudocode une procédure pour générer une séquence de taille  $N$  de la loi de  $X$ .
- e) Comment utiliseriez-vous cet algorithme pour simuler une loi normale  $\mathcal{N}(m, \sigma)$  avec  $m \neq 0$  et  $\sigma > 0$ .
- f) Ecrivez en Octave un programme qui permet de visualiser la séquence générée et comparer visuellement les distributions empiriques à la distribution des séquences des  $X$  et des  $Y$  utilisées aux distributions de Gauss centrée réduite et de Laplace théoriques proposées dans le cadre de ce problème.

**E 4.5** On considère deux caractères aléatoires  $X_1$  et  $X_2$  qui ne soient pas nécessairement indépendants.

- a) Formulez le problème de simulation du mélange  $X$  de ces deux caractères.
- b) Ecrivez en pseudocode un algorithme que permet de simuler une telle variable aléatoire.