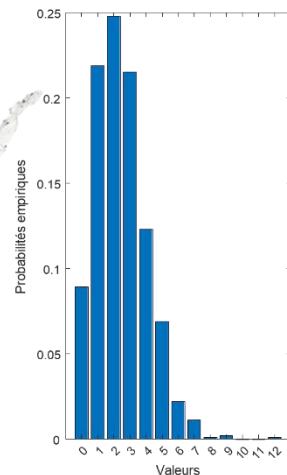
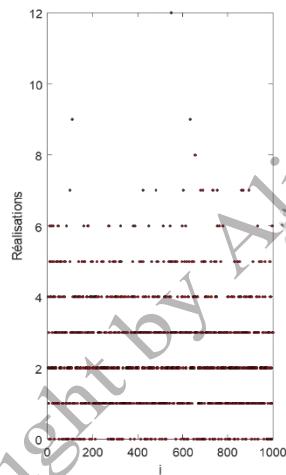


Simulation Stochastique

Prof. Ali HAMILI



Copyright by Ali HAMILI - Do not copy

Table des matières

Introduction	7
Chapitre 1 : Algorithmes et leur codage.....	11
1 Introduction	11
1.1 Formulation du Problème	11
1.2 Modélisation.....	11
1.3 Choix de la Méthode	12
1.4 Développement de l’Algorithme.....	12
1.5 Implémentation	12
1.6 Validation et Vérification.....	12
1.7 Interprétation et Présentation des Résultats	12
2 Notion de langage.....	12
2.1 Alphabet et Langage	12
2.2 Opérations sur les Mots.....	13
2.2.1 Concaténation	14
2.2.2 Etoile de Kleene.....	14
2.2.3 Union	14
2.2.4 Intersection	14
2.2.5 Complémentation.....	14
2.2.6 Inversion	14
2.2.7 Troncature	14
2.2.8 Remplacement	15
2.3 Langage Machine	15
3 Calcul et codage des nombres	15
3.1 Codage en base B	16
3.2 Codage des entiers positifs en base B	18
3.3 Passage du codage en base décimale au codage en base B	18
3.4 Codage des nombres réels positifs en base B.....	20

3.5 Méthode de conversion des nombres réels positifs en base B	21
3.6 Codage des nombres réels en virgule flottante	22
3.7 Codage binaire des nombre entiers naturels signés.....	23
3.7.1 Méthode du bit de signe et la valeur absolue.....	23
3.7.2 Méthode du complément logique	24
3.7.3 Méthode du complément arithmétique	24
3.7.2 Méthode du complément logique (ou complément à 1)	24
4 Notion de programmation informatique.....	25
5 Algorithmique et modélisation	26
5.1. Représentation d'un algorithme	26
5.2. Validité et exactitude d'un algorithme	27
6 Notions de programmation informatique	28
6.1 Programmation Procédurale	28
6.2 Programmation Fonctionnelle	28
6.3 Programmation Orientée Objet	28
6.4 Programmation Logique.....	29
6.5 Programmation Parallèle.....	29
6.6 Langages de Spécification.....	29
6.7. Programmation Automatique	29
6.8 Programmation Impérative.....	29
7 Code et pseudocode.....	30
7.1 Pseudocode et opérations	31
7.2 Structures de contrôle et pseudocode	33
7.2.1 Enchaînement ou bloc d'instructions.....	34
7.2.2 Répétition.....	34
7.2.3 Choix.....	36
7.3 Modularité ou structuration hiérarchique.....	37
8 Développements en pseudocode	39
8.1 Calcul de la moyenne d'un vecteur de valeurs.....	39
8.2 Problème de jeu de dés.....	40
8.3 Fonction factorielle	41
8.4 Echange de valeurs dans un tableau.....	41
8.5 Tri par insertion.....	42
9 Amélioration de lisibilité et optimisation des algorithmes.....	42

9.1 Evaluation de la complexité temporelle et spatiale de l'algorithme	43
9.2 Simplification de la syntaxe et commentaires.....	43
9. 3 Amélioration des conventions de nommage	43
10 Exercices de récapitulation.....	43
Chapitre 2 : Simulation de séquences de nombres au hasard.....	45
1 Notions de simulation.....	45
2 Techniques de génération de séquences de nombres au hasard	46
2.1 Méthodes	46
2.1.1 Tables de Nombres au Hasard	46
2.2 Générateurs procéduraux de nombres pseudo-aléatoires	50
2.2.1 Générateur pseudo-aléatoire	50
2.2.2 Méthode du carré médian de Von-Neumann.....	51
2.2.3 Méthodes congruentielles	53
2.2.4 Types de générateurs congruentiels	55
3 Critères d'existence d'une période maximale	56
3.1 Cas d'une base modulo générale.....	56
3.2 Cas d'une base modulo un nombre premier.....	57
3.3 Cas d'une base modulo une puissance de 2	58
4 Critères d'ordre statistique et discrépance	59
4.1 Mesures de discrépance.....	59
4.2 Uniformité et calcul d'intégrales.....	61
4.3 Discrépance étoile	61
4.4 Application.....	62
4.5 Etude d'exemples	63
5 Amélioration de la qualité des générateurs congruentiels	65
5.1 Méthode du registre à décalage avec rétroaction linéaire	66
5.2 Méthodes congruentielles vectorielle.....	69
6 Approche Monte-Carlo pour l'évaluation d'intégrales	72
6.1 Forme canonique	72
6.2 Cas général des deux bornes finies	75
6.3 Autres cas unidimensionnels	76
6.4 Calcul des intégrales multiples.....	76
6.5 Autours de la vitesse de convergence de l'intégration Monte-Carlo	77

7 Simulation d'un vecteur de variables aléatoires	77
8 Méthode d'intégration à l'aide d'une fonction d'importance	78
9 Exercices de récapitulation.....	81
Chapitre 3 : Simulation de variables aléatoires réelles discrètes.....	85
1 Notions de simulation de variables aléatoires réelles discrètes.....	85
2 Méthode de transformation inverse.....	86
2.1 Principe de construction.....	86
2.2 Implémentation, en pseudo-code, d'un simulateur d'une v.a.r. discrète à domaine fini	87
2.2.1 Générateur de la variable	87
2.2.2 Générateur d'une séquence de réalisations.....	88
2.3 Exemple d'implémentation d'une v.a.r. discrète à support fini	88
2.3.1 Algorithme du générateur	88
2.3.2 Algorithme de génération de la séquence	89
2.3.3 Application	89
2.4 Implémentation d'un générateur pour une v.a. discrète à domaine infini	89
2.4.1 Algorithme de recherche de N0.....	90
2.4.2 Algorithme de simulation d'une séquence	91
3 Méthode de rejet/acceptation	92
4 Méthode de composition	96
5 Exercices de récapitulation.....	99
Chapitre 4 : Simulation de variables aléatoires continues.....	102
1 Introduction	102
2 Méthode de transformation inverse.....	102
2.1 Principe de construction.....	102
2.2 Exemples d'application.....	104
3 Méthode de rejet/acceptation (variable de contrôle)	105
4 Méthode de composition	107
5 Exercices de récapitulation.....	107
Références bibliographiques	110

Introduction

Les ingénieurs et chercheurs travaillent dans le domaine de l'informatique sont souvent confrontés à des comportements et phénomènes incertains et à des résultats expérimentaux divergents. Cela peut être dû à de nombreux facteurs, tels que la complexité des systèmes informatiques, les interactions imprévues entre leurs composants, ou même la présence des erreurs de conception dans des logiciels et programmes informatiques. Dans ces situations, l'incertitude peut être attribuée au hasard, ce qui signifie essentiellement que les variations inexplicables dans les résultats sont le résultat de facteurs inconnus ou imprévisibles. Cette façon de voir les choses met en évidence le besoin d'approches méthodiques et de techniques sophistiquées pour comprendre, anticiper, contrôler et rejouer de manière artificielle ces variations. Ainsi, les ingénieurs informatiques utilisent souvent des méthodes telles que la modélisation probabiliste, la simulation stochastique et l'analyse de sensibilité pour évaluer les incertitudes et comprendre les limites de leurs systèmes.

De plus, le développement de technologies comme l'apprentissage automatique (machine learning) et l'intelligence artificielle (IA) permet aux chercheurs de créer des algorithmes capables de reproduire des scénarios non observés afin d'anticiper le hasard et réagir aux variations imprévues de manière plus efficace. Dans ce contexte, les probabilités et les statistiques sont considérés comme des outils d'ingénierie essentiels pour modéliser de nombreux problèmes pertinents dans divers domaines des sciences appliquées. L'approche stochastique est souvent employée lorsque les méthodes de modélisation traditionnelles ne peuvent pas produire un modèle de comportement déterministe satisfaisant à partir des outils mathématiques conventionnels.

Pour les étudiants en ingénierie, acquérir une solide compréhension des concepts de probabilité constitue un prérequis essentiel avant d'aborder la simulation Monte Carlo. Cette méthode repose fondamentalement sur les principes probabilistes, la statistique et l'algorithmique, nécessitant ainsi une maîtrise préalable de ces domaines pour une application précise et efficace. En assimilant ces fondements, les étudiants peuvent développer des compétences avancées en simulation Monte Carlo, renforçant ainsi leur capacité à résoudre des problèmes complexes dans divers domaines de l'ingénierie. Ces concepts sont largement utilisés dans des domaines tels que l'analyse de risques, la fiabilité des systèmes, la modélisation des processus aléatoires et l'optimisation stochastique, ce qui en fait des outils indispensables dans la prise de décision et la résolution de problèmes réels.

Le besoin pour « produire ou reproduire le hasard » par ordinateur est essentiel pour évaluer le comportement des systèmes complexes dans des scénarios non observés encore. Ainsi, les méthodes de simulation font partie de ces méthodes sous-jacentes aux applications informatiques permettant à une machine d'imiter les comportements aléatoires.

Ce cours, destiné à des élèves ingénieurs, se focalise sur la nature des nombres générés des algorithmes de simulation de variables aléatoires dans un contexte informatique. Nous verrons que les méthodes de simulation stochastique pour modéliser et analyser des phénomènes aléatoires reposent sur l'usage de nombres pseudo-aléatoires. Dans le cadre de ce cours, des séquences de nombres seront générées par des algorithmes déterministes appelés générateurs de nombres au hasard ou plutôt de « nombres pseudo-aléatoires ». En effet, puisque ces algorithmes sont déterministes, une fois que la valeur initiale (graine) est fixée, le générateur produira toujours la même séquence de nombres pseudo-aléatoires. Cette propriété est importante pour assurer la reproductibilité des simulations. Les nombres pseudo-aléatoires sont généralement supposés suivre une distribution uniforme entre 0 et 1, et ils sont indépendants. Ils peuvent ensuite être utilisés dans le cadre de calculs complexes tels que le calcul d'intégrales simples ou multiples. De plus, ces nombres peuvent être transformés pour suivre d'autres distributions probabilistes discrètes (telles que la loi de Bernoulli, binomiale, géométrique, etc.) ou continues (comme la distribution normale, exponentielle, bêta, etc.) à l'aide de techniques telles que la méthode de transformation inverse, la méthode de Box-Muller pour la distribution normale, ou encore la méthode de rejet et acceptation.

L'utilisation de nombres pseudo-aléatoires dans les simulations stochastiques permet aux chercheurs, ingénieurs et scientifiques de modéliser des phénomènes complexes et incertains dans

divers domaines tels que la finance, l'ingénierie, les sciences sociales, la météorologie et bien d'autres. Ces méthodes sont essentielles pour évaluer le comportement des systèmes complexes dans des conditions aléatoires et prendre des décisions éclairées basées sur ces évaluations. Par exemple, en évaluation de la performance des réseaux informatiques, la simulation stochastique permet d'évaluer les différents attributs de performance sous diverses contraintes de dimensionnement. Dans le domaine de la finance, elle facilite l'estimation des risques et des opportunités liés aux fluctuations des valeurs boursières. Pour les planificateurs et les décideurs, la simulation de scénarios offre les éléments nécessaires pour prendre des décisions éclairées.

Dans une perspective éducative, à la fin de chaque chapitre de ce manuscrit, vous trouverez plusieurs exercices. Ces exercices impliquent principalement la conception d'algorithmes en pseudocode. Ces algorithmes peuvent ensuite être implémentés dans un langage informatique de votre choix, tels que R, Python, C/C++, ou Java. Pour des raisons de richesse en bibliothèques mathématiques et graphiques, ainsi que pour des questions de disponibilité des outils, nous recommandons aux élèves ingénieurs d'implémenter leurs programmes dans les langages de programmation scientifique Octave de GNU (disponible en téléchargement gratuit) ou Matlab de Mathworks (propriétaire). Pour Matlab, nous disposons de quelques licences à l'ENSIAS (veuillez vous référer aux techniciens du centre de calcul pour plus d'informations). Ces deux langages disposent d'une syntaxe similaire à la notation pseudocode utilisée dans ce cours, et ils sont équipés de bibliothèques intégrées permettant la visualisation graphique en 2D/3D des résultats. Pour télécharger les dernières versions d'Octave sous les termes de la licence [GNU General Public License](#) (GPL), vous pouvez le faire à partir de l'un des sites officiels suivants : [<https://www.gnu.org/software/octave>], [<ftp://ftp.gnu.org/gnu/octave>], ou 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 utilisateurs de Windows, un programme d'installation binaire officiel pour MS-Windows est également disponible. Pour installer Octave sur le système d'exploitation macOS, veuillez consulter les instructions d'installation disponibles sur le wiki prévu pour les utilisateurs de ce système d'exploitation: [https://wiki.octave.org/Octave_for_macOS]. La dernière version d'Octave peut être étendue avec des composants similaires aux boîtes à outils Matlab Toolboxes. Pour les retrouver, veuillez examiner les sites de stockage tels que Sourceforge à l'adresse [<https://octave.sourceforge.io/>] ou Github à [<https://gnu-octave.github.io/pkg-index>].

Pour plus de précision concernant documentation nécessaire à l'installation et à l'utilisation d'Octave (version 5.2.0) ainsi que de ses différents composants et bibliothèques, veuillez-vous rapporter au lien : [<https://octave.org/doc/v5.2.0/>].

Copyright by Ali HAMILILI - Do not copy

Chapitre 1 : Algorithmes et leur codage

1 Introduction

La résolution méthodique des problèmes en probabilités, statistiques et simulation stochastique repose sur des algorithmes de calcul scientifique, offrant une solution précise et efficace aux problèmes les plus complexes. Ces algorithmes doivent être élaborés avec soin en combinant des méthodes analytiques et algébriques pures, ainsi que les principes fondamentaux des théories classiques en probabilités et statistiques. Ils doivent intégrer également des techniques de calcul numérique avancées pour résoudre efficacement des problèmes complexes. Ce chapitre présente les principes de l'interaction Homme-Machine et pose les fondations du calcul analytique et algébrique essentielles pour le codage et l'exécution de calculs numériques sur machine. Les concepts liés à la notation pseudocode seront également présentés et détaillés dans ce chapitre.

1.1 Formulation du Problème

Tout d'abord, il est essentiel de définir le problème de manière claire et précise. Cela implique de comprendre la question posée et de déterminer les variables, les données disponibles et les objectifs de l'analyse.

1.2 Modélisation

Ensuite, le problème est traduit en un modèle mathématique ou probabiliste. Cela peut impliquer l'utilisation de distributions probabilistes, d'équations statistiques ou de modèles stochastiques pour représenter le phénomène en question.

1.3 Choix de la Méthode

En fonction du problème et du modèle, il est nécessaire de choisir la méthode appropriée. Cela peut inclure des méthodes d'inférence statistique, des simulations de Monte Carlo, des méthodes d'optimisation ou d'autres techniques spécifiques aux probabilités et aux statistiques.

1.4 Développement de l'Algorithme

L'algorithme de résolution est développé en décrivant de manière détaillée les étapes à suivre pour résoudre le problème. Ces étapes sont généralement décrites sous forme d'une suite d'instructions, qui doivent être codées dans un langage de programmation informatique. Il peut s'agir de scripts dans des langages de programmation interprétés tels que Python, MATLAB, etc., ou de langages évolués comme C, C++, Java, etc.

1.5 Implémentation

Les instructions de l'algorithme sont traduites en code informatique et exécutées à l'aide d'un logiciel ou d'un calculateur. L'implémentation correcte de l'algorithme est cruciale pour obtenir des résultats précis.

1.6 Validation et Vérification

Les résultats obtenus sont vérifiés pour s'assurer qu'ils sont cohérents avec le problème posé et le modèle utilisé. Des techniques de validation croisée, des tests statistiques et d'autres méthodes peuvent être utilisés pour évaluer la validité des résultats.

1.7 Interprétation et Présentation des Résultats

Les résultats sont interprétés à la lumière du problème initial. Cela peut impliquer des analyses statistiques, la visualisation sous forme de sorties graphiques, et des conclusions basées sur les méthodes théoriques et les résultats obtenus à partir de l'implémentation informatique des algorithmes.

2 Notion de langage

2.1 Alphabet et Langage

L'échange d'informations entre l'Homme et la Machine dans le contexte du calcul s'effectue à travers des messages exprimés dans des langages spécifiques, structurant ainsi la communication

entre ces deux entités. Ces langages se déclinent en deux formes principales : les langages primaires et les langages évolués, chacun ayant ses caractéristiques distinctives et son domaine d’application particulier.

Les langages primaires reposent sur l’utilisation de suites binaires, constituées de 0 et de 1, pour représenter les données. C’est notamment le cas dans la programmation au niveau des microprocesseurs, où les instructions sont codées en langage binaire, formant ainsi le langage de base de la machine elle-même. En revanche, les langages évolués manipulent des suites complexes de lettres, de chiffres et de symboles de ponctuation et de séparation. Des langages tels que Python, Matlab, C++, Java, entre autres, sont des exemples de langages évolués. Ils sont utilisés dans le développement de logiciels et d’applications complexes, permettant une interaction plus intuitive et flexible entre l’Homme et la Machine. Les symboles élémentaires à partir desquels est construit un langage définissent l’alphabet de celui-ci. Les expressions construites sur ce langage sont interprétées par les ordinateurs pour exécuter les tâches spécifiées par les programmeurs.

Définitions

- *En informatique théorique, un alphabet est défini comme un ensemble Σ fini et dénombrable.*
- *Les éléments d’un alphabet Σ sont dits des lettres de Σ .*
- *La concaténation de suites finies de lettres de Σ donne lieu à des mots sur Σ . L’ensemble de tous les mots possibles sur l’alphabet Σ est noté Σ^* .*
- *On appelle langage sur un alphabet Σ , toute partie non-vide de Σ^* .*

Un alphabet est la base à partir de laquelle sont construits les mots et les langages formels. Les symboles de l’alphabet sont utilisés pour former des chaînes de caractères, qui peuvent être interprétées ou traitées par des systèmes informatiques, des automates ou d’autres outils d’analyse formelle. Cette notion d’alphabet est fondamentale dans la théorie des langages formels et de l’automate, et elle constitue un élément essentiel dans de nombreux domaines de l’informatique théorique. Par ailleurs, on peut définir des opérations sur les mots.

2.2 Opérations sur les Mots

En effet, dans le contexte des langages formels et de la théorie des automates, il est possible de définir diverses opérations sur les mots, qui sont des chaînes de caractères composées à partir d’un

alphabet donné. Ces opérations permettent de manipuler, de combiner ou de transformer des mots, ce qui est essentiel dans l'étude des langages formels et des automates. Les opérations sur les mots sont importantes en analyse formelle des langages et dans la conception d'automates et d'expressions régulières pour reconnaître et manipuler des motifs dans les chaînes de caractères. Nous donnons dans la suite les opérations couramment définies sur les mots et les langages.

2.2.1 Concaténation

L'opération de concaténation consiste à joindre deux mots pour former un nouveau mot. Si u et v sont deux mots, leur concaténation est notée $u \bullet v$ ou simplement uv .

2.2.2 Etoile de Kleene

L'étoile de Kleene d'un mot u , notée u^* , représente l'ensemble de tous les mots obtenus en concaténant zéro ou plusieurs copies de u .

2.2.3 Union

L'union de deux langages L_1 et L_2 consiste à former un nouveau langage composé de tous les mots qui appartiennent à L_1 ou à L_2 .

2.2.4 Intersection

L'intersection de deux langages L_1 et L_2 consiste à former un nouveau langage composé de tous les mots qui appartiennent à la fois à L_1 et à L_2 .

2.2.5 Complémentation

La complémentation d'un langage L consiste à former un nouveau langage composé de tous les mots qui n'appartiennent pas à L .

2.2.6 Inversion

L'inversion d'un mot u consiste à écrire le mot à l'envers. Par exemple, si $u = abc$ (u est composé de la concaténation des trois lettres a , b , et c), son inversion est cba .

2.2.7 Troncature

La troncature d'un mot u consiste à retirer un certain nombre de symboles du début ou de la fin du mot.

2.2.8 Remplacement

Le remplacement d'un sous-mot par un autre mot dans un mot donné est une opération courante dans le traitement du langage naturel et la manipulation de chaînes de caractères en informatique.

2.3 Langage Machine

Le terme « langage machine » se réfère au langage basique utilisé par les ordinateurs pour exécuter des instructions. Les ordinateurs ne comprennent que le langage machine, qui est composé de séquences binaires de 0 et de 1, appelées code binaire. Chaque instruction en langage machine est codée sous une forme spécifique de bits qui est interprétée par le processeur de l'ordinateur. Le langage machine $\{0,1\}^n$ construit sur l'alphabet $\{0,1\}$ construisant des mots de n bits (0 ou 1). Pour mettre en place des calculs à partir de cette représentation, il est requis de pourvoir des opérations arithmétiques « + », « - », « * » et « / » sur $\{0,1\}^n$. La définition de ces quatre opérations fait intervenir une représentation polynomiale des mots à coefficients dans $\mathbb{Z}/2\mathbb{Z}$. En fait, il existe d'autres codages possibles pour établir des calculs sur les nombres, en dehors du codage binaire.

3 Calcul et codage des nombres

Les techniques de calcul primitives ont constitué les fondations des systèmes mathématiques complexes que nous connaissons actuellement. L'origine du mot français *Calcul* remonte au latin *Calculi*, qui signifie *Caillou*. Cette étymologie témoigne de l'utilisation des cailloux en tant qu'instruments de calcul dans les premières civilisations humaines. En effet, à travers l'histoire, de grandes civilisations telles que les Grecs, les Romains, les Chinois, les Mayas et les Arabes, ainsi que bien d'autres, ont développé des systèmes de numération et des techniques de calcul variés. Ces systèmes étaient diversifiés selon les cultures, mais tous avaient pour objectif commun de résoudre des problèmes mathématiques et de faciliter les échanges commerciaux, la construction, l'astronomie et d'autres aspects de la vie civilisée. Au fil des siècles, les méthodes de calcul ont progressé et se sont sophistiquées, aboutissant l'automatisation des calculs et à des avancées significatives dans les domaines des mathématiques et de l'informatique. Aujourd'hui, les ordinateurs et autres dispositifs électroniques de calcul sont capables d'effectuer des calculs extrêmement complexes en des temps records.

3.1 Codage en base B

Un codage en base B sur un alphabet Σ_B peut être défini en utilisant B symboles distincts, appelés les chiffres de cette base de codage. Dans ce contexte, B représente la base du système de numération, c'est-à-dire le nombre de chiffres différents disponibles pour représenter les valeurs.

Définition

Les lettres de l'alphabet Σ_B de codage en base B sont appelés des chiffres.

Ces chiffres sont simplement les symboles de l'alphabet Σ_B . Le choix d'une base de codage des nombres dépend du contexte d'application et des exigences spécifiques du problème analysé.

Exemples

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

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

- 2) 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).

- 3) Dans la base octale (base de 8), Σ_8 contient huit chiffres,

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

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

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



Fig. 1.1 : Code ASCII (American Standard Code for Information Interchange)

Symbol	Binaire	Octale	Décimale	Hexadécimale	Symbol	Binaire	Octale	Décimale	Hexadécimale	Symbol	Binaire	Octale	Décimale	male
espace	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
/	101111	57	47	2F	O	1001111	117	79	4F	o	1101111	157	111	6F
0	110000	60	48	30	P	1010000	120	80	50	p	1110000	160	112	70
1	110001	61	49	31	Q	1010001	121	81	51	q	1110001	161	113	71
2	110010	62	50	32	R	1010010	122	82	52	r	1110010	162	114	72
3	110011	63	51	33	S	1010011	123	83	53	s	1110011	163	115	73
4	110100	64	52	34	T	1010100	124	84	54	t	1110100	164	116	74
5	110101	65	53	35	U	1010101	125	85	55	u	1110101	165	117	75
6	110110	66	54	36	V	1010110	126	86	56	v	1110110	166	118	76
7	110111	67	55	37	W	1010111	127	87	57	w	1110111	167	119	77
8	111000	70	56	38	X	1011000	130	88	58	x	1111000	170	120	78
9	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]	1011101	135	93	5D]	1111101	175	125	7D
>	111110	76	62	3E	^	1011110	136	94	5E	^	1111110	176	126	7E
?	111111	77	63	3F	-	1011111	137	95	5F	suppl	1111111	177	127	7F

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)

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 Σ_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)

- 1) 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.

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

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

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

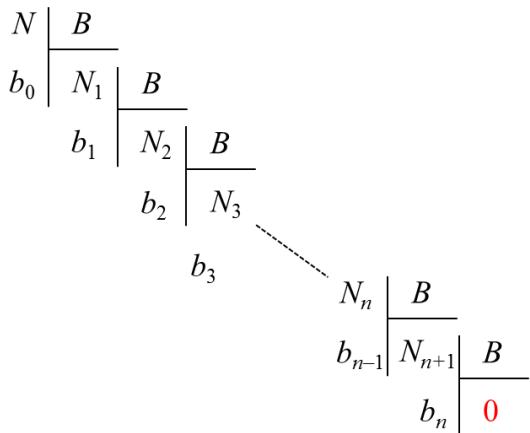
$$2257 = 4 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 1 \times 8^0 \equiv \overline{4321}^8$$

- 4) 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 :



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

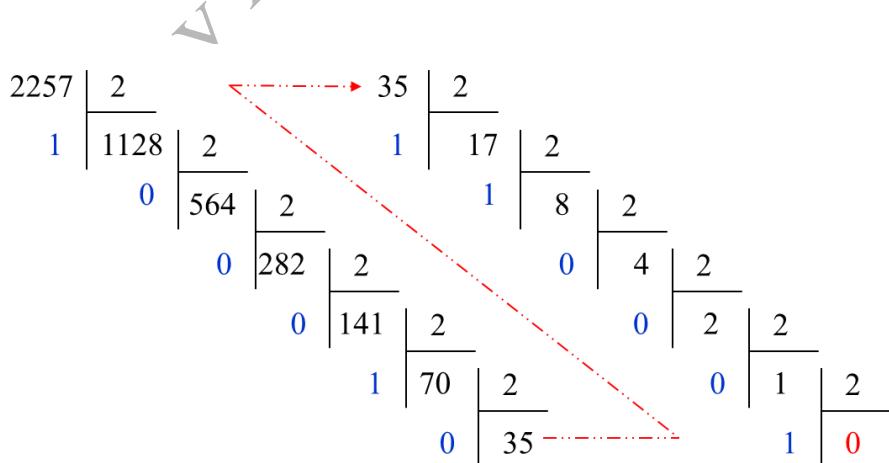
$$= \overline{b_n \ b_{n-1} \ b_{n-2} \cdots \ b_1 \ b_0}_B$$

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

Le processus de représentation en base B implique la conversion des chiffres en valeurs de position, leur multiplication par les puissances appropriées de B , l'addition des valeurs de position pour former le nombre final et, dans le cas des nombres réels, la prise en compte de la partie décimale et des nombres négatifs selon les règles spécifiques du système de représentation des nombres dans la base B utilisée.

Exemple

En base binaire la conversion se fait



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

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

Ainsi, la représentation en base 2 implique des séquences de chiffres binaires 0 et 1, la conversion entre les systèmes décimal et binaire, les opérations arithmétiques de base et la représentation des nombres à virgule flottante en utilisant une notation spécifique.

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 Σ_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 $X \equiv \overline{2257,61}^{10}$:

- 1) 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,

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

- 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,

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

Ainsi,

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

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

$$\begin{aligned}2257,61 &\approx \overline{4321,47024365}^8 \\&\equiv 4 \times 8^3 + 3 \times 8^2 + 2 \times 8^1 + 8^0 + 4 \times 8^{-1} + 7 \times 8^{-2} + 2 \times 8^{-4} + \\&\quad 4 \times 8^{-5} + 3 \times 8^{-6} + 6 \times 8^{-7} + 5 \times 8^{-8}\end{aligned}$$

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

$$\begin{aligned}2257,61 &\approx \overline{8D1,9C28F5C2}^{16} \\&\equiv 8 \times 16^2 + D \times 16^1 + 1 + 9 \times 16^{-1} + C \times 16^{-2} + 2 \times 16^{-3} + \\&\quad 8 \times 16^{-4} + F \times 16^{-5} + 5 \times 16^{-6} + C \times 16^{-7} + 2 \times 16^{-8}\end{aligned}$$

3.5 Méthode de conversion des nombres réels positifs en base B

Pour convertir des nombres réels positifs en base B , le processus implique les étapes suivantes :

- Décomposez le nombre réel en somme de la partie entière et de la partie fractionnaire.
- Pour la conversion de la partie entière, utilisez la méthode de la division entière, similaire à celle utilisée pour les entiers positifs.
- Conversion de la partie fractionnaire :
 - Multipliez la partie fractionnaire par B .
 - Décomposez ce nombre en une somme de la partie entière et de la nouvelle partie fractionnaire.
 - Répétez cette opération pour la nouvelle partie fractionnaire et ainsi de suite.
 - Arrêtez le calcul lorsque la nouvelle partie fractionnaire est nulle ou lorsque vous atteignez la précision souhaitée.
 - La partie fractionnaire en base B est obtenue en concaténant les parties entières obtenues dans l'ordre de leur apparition à la fin de ce processus de calcul.

Il faut se rappeler que la conversion de la partie fractionnaire ne permet pas toujours d'obtenir une représentation exacte sous forme d'un nombre fini de chiffres.

3.6 Codage des nombres réels en virgule flottante

La règle fondamentale du codage des nombres réels en virgule flottante est de pouvoir représenter un nombre réel avec une virgule flottante et une précision limitée (approximative). Cette méthode permet de coder uniquement les chiffres significatifs représentant ce nombre. Par conséquent, le nombre est présenté sous forme normalisée, ce qui permet de déterminer sa mantisse et son exposant.

Dans ces conditions, la forme normalisée d'un 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 $X \equiv \overline{2257,61}^{10}$ 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, $\pm 0,xxxxxxxx \times B^E$, 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 y a trois techniques couramment utilisées pour coder des entiers naturels signés en base 2. L'objectif de ce paragraphe est de présenter ces trois techniques de codage des entiers naturels signés en base 2, de les présenter en détail et de mettre en évidence leurs avantages et inconvénients respectifs. Comprendre ces techniques est essentiel dans le domaine de l'informatique et de l'algorithme, car elles sont largement utilisées dans la représentation des nombres dans les systèmes informatiques modernes.

3.7.1 Méthode du bit de signe et la valeur absolue

Cette méthode utilise un bit pour indiquer le signe du nombre (0 pour positif, 1 pour négatif) et les bits restants codent la valeur absolue du nombre.

3.7.2 Méthode du complément logique

Dans cette méthode, le bit de signe est également utilisé, mais pour les nombres négatifs, les bits sont inversés (0 devient 1 et 1 devient 0) pour obtenir le complément logique du nombre.

3.7.3 Méthode du complément arithmétique

Cette méthode utilise également un bit pour le signe. Pour les nombres négatifs, elle utilise le complément arithmétique du nombre, qui est obtenu en inversant tous les bits du nombre et en ajoutant 1 au résultat.

Chacune de ces méthodes utilise un seul bit pour indiquer si le nombre est positif ou négatif, permettant ainsi de représenter des entiers naturels signés en utilisant la base 2.3.7.1 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.

En fait, dans un nombre binaire, le bit de poids fort (ou *Most Significant Bit* - MSB en anglais) est le bit le plus à gauche, et le bit de poids faible (ou *Least Significant Bit* - LSB en anglais) est le bit le plus à droite. Cette désignation est importante lorsqu'on manipule des nombres binaires, car elle détermine l'ordre de valeur des bits dans le nombre.

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 connaître 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 $\overline{01010011}^2 \equiv 83$, son complément à 1 est $\overline{10101100}^2 \equiv -83$ 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 informatique est essentiellement un artefact statique qui, lors de son exécution, évolue de manière dynamique en réaction à l'état changeant de ses variables. La tâche cruciale qui incombe à tout programmeur est de conceptualiser le programme de manière à comprendre de façon approfondie les situations dynamiques qu'il engendre pendant son exécution. Cette compréhension est indispensable pour fournir une démonstration aussi précise que possible de la validité du programme, c'est-à-dire pour prouver de manière rigoureuse que le programme produira les résultats souhaités conformément à ses spécifications.

5 Algorithmique et modélisation

5.1. Représentation d'un algorithme

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 doit présenter divers avantages qui ne sont pas nécessairement exclusifs. Il doit contenir les instructions indispensables pour résoudre des problèmes spécifiques ou automatiser des tâches complexes en exploitant la rapidité des machines pour effectuer les calculs requis. Ainsi, il doit orienter et gérer les étapes ainsi que les calculs intermédiaires afin d'atteindre le résultat recherché. De plus, les algorithmes sont fréquemment utilisés pour explorer toutes les possibilités d'un problème complexe, permettant ainsi de rechercher une solution ou une bonne approximation parmi toutes les options disponibles. Ce processus, souvent désigné sous le nom de résolution numérique ou informatique, s'effectue sur un ordinateur.

Il existe plusieurs façons de représenter un algorithme pour une machine. Chacune d'elles a ses avantages et inconvénients. Le choix de l'une d'entre elles dépend du contexte dans lequel l'algorithme est développé et de la communauté à laquelle il est destiné :

- a) **Pseudocode** : Le pseudocode est une méthode informelle pour planifier un algorithme. C'est un mélange de langage naturel et de structures de contrôle de base empruntées à la programmation. Il est utilisé pour décrire l'algorithme de manière compréhensible sans se soucier des détails syntaxiques d'un langage de programmation spécifique.

- b) **Diagrammes de flux** : Les diagrammes de flux considèrent des formes géométriques pour représenter différentes étapes d'un algorithme. Les flèches indiquent le flux de contrôle d'une étape à l'autre. Ils sont particulièrement utiles pour visualiser la séquence des étapes d'un algorithme.
- c) **Diagrammes de Nassi-Shneiderman** : Ces diagrammes sont utilisés pour représenter graphiquement la logique d'un algorithme. Ils utilisent des boîtes pour représenter les structures de contrôle comme les séquences, les boucles et les branches, et sont très utiles pour visualiser la logique complexe des algorithmes.
- d) **Langages de modélisation formelle** : Certains langages et notations comme UML (*Unified Modeling Language*) peuvent être utilisés pour modéliser des algorithmes de manière formelle. Ils proposent ainsi divers diagrammes de structuration et de fonctionnement d'un programme. Ces outils permettent aux développeurs de visualiser et de comprendre plus facilement la logique des algorithmes complexes, facilitant ainsi le processus de conception et de développement des logiciels.
- e) **Langages de programmation** : Enfin, un algorithme peut également être représenté dans un langage de programmation spécifique. Cela implique d'écrire le code dans un langage comme C++, Java, Matlab, Python, R, etc. Le code source ainsi créé peut être directement exécuté par un ordinateur.

5.2. Validité et exactitude d'un algorithme

La validité d'un algorithme se réfère à la capacité de ce dernier à résoudre le problème pour lequel il a été conçu. Un algorithme est considéré comme valide s'il produit la sortie correcte pour toutes les entrées possibles conformément aux spécifications du problème.

Ainsi, nous pouvons affirmer qu'un algorithme est « correct » s'il résout adéquatement le problème de calcul donné. En d'autres termes, un algorithme sera considéré comme correct s'il produit une sortie correcte pour chaque cas d'entrée et termine son exécution. En revanche, un algorithme sera qualifié d'« incorrect » s'il ne parvient pas à s'arrêter pour tous les cas d'entrée ou s'il génère des réponses différentes de celles attendues dans certains cas. Dans ces situations, on dit que l'algorithme produit des résultats incorrects ou divergents. Cela permet de définir, en opposition, les « réponses convergentes » d'un algorithme correct.

A ce niveau, remarquons toutefois que, contrairement à ce que pourrait s'attendre un néophyte, les algorithmes incorrects peuvent parfois s'avérer très utiles, si leurs erreurs sont identifiées peuvent être contrôlées.

6 Notions de programmation informatique

La programmation est l'art de résoudre des problèmes à l'aide de programmes conçus pour être exécutés sur un ordinateur. Dans le domaine de l'informatique, divers types de programmation coexistent, chacun reposant sur un paradigme spécifique. On peut envisager la programmation d'une manière différente en la considérant comme un univers vaste et créatif. Chaque méthode est un langage unique parlé par les développeurs, un peu comme des dialectes, chacun ayant sa propre grammaire et ses règles. Ces paradigmes et langages offrent une variété d'outils pour résoudre des problèmes informatiques, permettant aux développeurs de choisir l'approche qui convient le mieux à leur projet en fonction de ses besoins spécifiques.

6.1 Programmation Procédurale

Ce paradigme se base sur le concept d'appel de procédures, où le programme est structuré en modules autonomes qui effectuent des tâches spécifiques. Imaginez cette manière de programmer comme une recette de cuisine. Le chef suit des étapes précises, chaque étape étant une procédure, pour créer un plat délicieux. Exemples de langages : Ada, C, COBOL, Fortran, Pascal.

6.2 Programmation Fonctionnelle

Ce paradigme repose sur l'évaluation de fonctions mathématiques pures, sans état partagé ni effets de bord. C'est comme les mathématiques. Vous avez des fonctions pures, comme des équations, qui prennent des arguments en entrée, les transforment, et produisent un résultat sans jamais altérer les arguments d'origine. Exemples de langages : Haskell, Lisp, OCaml, Scala.

6.3 Programmation Orientée Objet

Ce paradigme utilise des types abstraits de données, tels que des objets, des classes, et l'héritage pour organiser le code en entités réutilisables. C'est comme une usine de fabrication. Vous avez des objets (comme des pièces dans une usine), chacun avec ses caractéristiques uniques, qui peuvent effectuer des actions et interagir avec d'autres objets. Exemples de langages : Java, C++, Python, Ruby, Swift.

6.4 Programmation Logique

Dans ce paradigme, le programme est basé sur des clauses logiques, l'unification, et l'instanciation. Il procède comme dans un jeu de détection. Vous posez des questions logiques et le programme essaie de trouver les réponses en vérifiant des faits et des règles. Exemples de langages : Prolog, Logic Programming, Mercury.

6.5 Programmation Parallèle

Ce paradigme repose sur l'exécution simultanée de plusieurs tâches ou processus, avec une synchronisation et une communication entre eux. Il ressemble à un orchestre. Chaque musicien joue une partie différente simultanément, mais ils doivent se synchroniser pour créer une belle mélodie. Exemples de langages : Java (pour la programmation multi-threading), CUDA, OpenMP.

6.6 Langages de Spécification

Ces langages sont utilisés pour définir formellement les propriétés et les spécifications d'un système logiciel, souvent en utilisant la logique du premier ordre et la théorie des ensembles. Ils fonctionnent comme les plans détaillés d'architecte. Ils décrivent précisément ce que le logiciel doit faire et comment il doit le faire. Exemples : VDM, Z, Alloy.

6.7. Programmation Automatique

Il s'agit ici de génération automatique du code à partir de spécifications, souvent dans le contexte des environnements de génie logiciel. C'est comme si vous aviez un assistant intelligent. Vous lui dictez ce que vous voulez, et il génère le code pour vous. Exemple de langage : SMX-Cogitor, Xtext.

6.8 Programmation Impérative

Ce paradigme est basé sur la notion d'actions exécutées séquentiellement. Il englobe la plupart des formes précédentes de programmation et est le plus couramment utilisé dans de nombreux langages de programmation. Il ressemble à écrire une histoire. Vous écrivez les étapes les unes après les autres, et le programme suit ces instructions séquentiellement. Exemples de langages : C, Java, Python, Ruby, JavaScript.

Définition

La programmation impérative repose sur l'exécution d'actions.

Un langage de programmation est qualifié d'impératif lorsqu'un programme écrit dans ce langage se compose d'une série d'instructions destinées à être exécutées par une machine. L'action principale dans ce contexte est généralement l'affectation de variables. Il est essentiel de noter que le concept de variable en programmation diffère de sa signification en mathématiques. En programmation, une variable représente un emplacement mémoire de l'ordinateur où diverses données, qu'elles soient numériques ou autres, peuvent être stockées. La valeur d'une variable peut être récupérée et modifiée à tout moment pendant l'exécution du programme.

7 Code et pseudocode

Il existe une grande variété de langages de programmation conçus pour différents systèmes, tels que les ordinateurs. Parmi eux, on trouve des langages plus ou moins populaires tels que Basic, Fortran, C, C++, Tcl/OTcl, Python, Perl, Octave (ou Matlab), Ada, Java, Smalltalk, Simula et Lisp. En outre, il existe des langages spécifiques développés pour les constructeurs de calculatrices dites « programmables »..

Définition

En programmation impérative, le terme « code » fait référence à l'expression d'un algorithme dans un langage de programmation informatique spécifique.

Un processus de développement intégrant l'utilisation d'un pseudocode est une approche très utile lorsque l'informaticien souhaite structurer son projet par raffinements progressifs, en suivant une méthode de haut en bas (Top-Down). Dans cette approche, l'informaticien commence par exprimer les différents algorithmes spécifiques à son projet sous forme de descriptions pseudocodes. A ce stade, il peut modifier ces algorithmes pour assurer leur intégration correcte. Par la suite, il peut traduire ces descriptions dans un langage de programmation cible afin de les tester sur la machine.

L'objectif principal d'une approche basée sur le pseudocode est d'exprimer dans un premier temps les algorithmes dans un langage proche des langages naturels, tels que l'arabe, le français ou l'anglais. Le développement en pseudocode permet d'exprimer les spécifications et d'améliorer l'algorithme sans se préoccuper des contraintes syntaxiques propres au langage de programmation, ni des fonctions prédefinies spécifiques à ce langage. Le pseudocode vise ainsi de dégager le

concepteur d'un algorithme des détails techniques excessifs des langages de programmation informatique, tout en offrant la possibilité d'être traduit dans ces langages à une étape avancée du projet.

Définition

Le pseudocode désigne une description informelle et de haut niveau du fonctionnement d'un algorithme.

Le pseudocode est un langage de description d'algorithmes qui se situe à mi-chemin entre le langage naturel, compréhensible par les humains, et les langages de programmation, interprétables par les ordinateurs. Il suit les conventions de structuration du code tout en étant conçu pour être compris par les humains plutôt qu'interprété par les machines. L'écriture d'un algorithme en pseudocode vise à simplifier la transition vers un langage de programmation spécifique. Ainsi, avant de formuler une solution dans un langage de programmation particulier, il est souvent recommandé de l'exprimer dans un premier temps en pseudocode. Cette étape intermédiaire contribue à décomposer la complexité de la rédaction du programme. Lors de l'élaboration du pseudocode, l'accent est mis sur les modèles mathématiques essentiels utilisés pour résoudre le problème sous forme d'algorithme. A une étape ultérieure, le pseudocode est traduit dans la syntaxe impérative du langage de programmation désiré, en utilisant les fonctions prédefinies spécifiques à ce langage. Cette approche permet de se concentrer uniquement sur la modélisation mathématique de la solution lors de l'élaboration du pseudocode, évitant ainsi de se préoccuper simultanément de cette modélisation et des contraintes syntaxiques du langage utilisé. Le pseudocode vise à normaliser la phase de pré-écriture des programmes malgré la diversité des langages de programmation et de leurs syntaxes.

7.1 Pseudocode et opérations

En pseudocode, les opérations sont des instructions qui indiquent les calculs ou les actions à effectuer dans un algorithme. Ces opérations peuvent inclure des opérations mathématiques, des comparaisons, des affectations de valeurs aux variables, des lectures d'entrées, des affichages de résultats, et d'autres manipulations de données.

Postulat

- *L'affectation « ← » est une opération prédefinie 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 prédéfinis.*

Le pseudocode est un outil pour concevoir et confectionner des algorithmes de manière abstraite, avant de les implémenter dans un langage de programmation spécifique. Ainsi, les opérations en pseudocode peuvent être aussi simples ou complexes que nécessaire pour résoudre un problème donné. Elles décrivent de manière abstraite les étapes logiques que l'algorithme doit suivre, indépendamment des détails syntaxiques spécifiques à tout langage de programmation.

Exemples

L'affectation consiste à attribuer une valeur à un nom de variable. Elle se fait généralement à l'aide de l'opérateur d'attribution que nous notons ici « \leftarrow » :

```
x ← 10 ;
y ← x + 5 ;
```

Les opérations d'addition « + », soustraction « - », multiplication « * », division « / » et modulo « mod » sont supposées être prédéfinies pour le calcul arithmétique en pseudocode

```
somme ← a + b ;
différence ← a - b ;
produit ← a * b ;
quotient ← a / b ;
modulo ← a % b ;
```

La lecture et l'affichage sont des moyens d'interaction entre l'homme et la machine dans le contexte de l'informatique. La lecture décrit le processus par lequel un dispositif informatique récupère des données à partir d'une source externe. Cette source peut être le clavier, un fichier sur le disque dur, un capteur, une base de données, etc. La lecture permet ainsi de recevoir des informations de l'utilisateur ou de tout autre dispositif externe. L'affichage, d'autre part, est le processus par lequel un dispositif informatique présente des données à l'utilisateur ou à d'autres

systèmes. Cela peut se faire à travers l'écran (affichage visuel), l'imprimante (affichage papier), des haut-parleurs (affichage audio), ou d'autres dispositifs de sortie. L'affichage permet donc au dispositif informatique de communiquer des résultats, des messages ou des informations à l'utilisateur.

```
Lire(nombre) ;  
Afficher("Le nombre saisi est : " + nombre) ;
```

Ces deux opérations sont essentielles dans le développement des interfaces homme-machine, car elles permettent aux utilisateurs de communiquer avec les ordinateurs en fournissant des données (en lecture) et en recevant des résultats ou des informations (par affichage). Ces interactions sont à la base de nombreuses applications informatiques, des simples programmes en ligne de commande aux interfaces utilisateur graphiques sophistiquées.

L'union « \cup », l'intersection « \cap », la différence « \setminus » et la complémentation (ou négation) « \neg » ensemblistes sont supposées être prédéfinies pour l'algèbre ensembliste.

Les opérations booléennes sont supposées définie en pseudocode.

7.2 Structures de contrôle et pseudocode

Le pseudocode sert d'outil intermédiaire pour décrire de manière informelle et structurée les étapes d'un algorithme ou d'un processus, permettant aux programmeurs de planifier et de comprendre la logique d'un algorithme avant de le traduire dans un langage de programmation spécifique. Il a pour objectif de structurer les algorithmes.

Postulat

En écriture pseudocode, les structures de contrôle de base sont l'enchaînement, la répétition et le choix.

Ces trois structures de contrôle constituent le fondement de l'écriture pseudocode et sont utilisées pour décrire de manière abstraite les algorithmes préparant l'étape de programmation.

7.2.1 Enchaînement ou bloc d'instructions

L'enchaînement représente simplement l'exécution séquentielle des instructions. Les instructions sont exécutées dans l'ordre dans lequel elles sont écrites, de haut en bas.

Définition

Un enchaînement consiste en une succession d'instructions.

La syntaxe d'un enchainement est alors sous la forme générale suivante.

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

Exemple

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

7.2.2 Répétition

La structure de répétition permet de répéter un ensemble d'instructions tant qu'un nombre spécifique de fois n'est pas atteint ou qu'une condition donnée est vraie.

Définition

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

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

appelée boucle « pour » et

```
Tant_QUE Condition FAIRE
    Instruction1 ;
    Instruction2 ;
    ...
Fin_tant_QUE ;
```

appelée boucle « tant que ».

Le nombre de répétitions dans le premier cas est généralement défini pour une variable i qui parcourt les valeurs d'ensemble discret $n..m$ avec un pas de s . Cela peut être noté en pseudocode sous la forme « **i** \leftarrow **n:s:m** ». En revanche, dans le second cas, la répétition se poursuit indéfiniment tant que la clause spécifiée « **Condition** » reste vraie.

Exemples

La boucle suivante

```
...
Pour i  $\leftarrow$  5 à 17 FAIRE
    S[i]  $\leftarrow$  i2
Fin_pour ;
...
```

peut s'écrire alternativement

```
...
Pour i  $\leftarrow$  5:1:17 FAIRE
    S[i]  $\leftarrow$  i2
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 est fondamentale en programmation et dans de nombreux autres domaines. En programmation, les structures de contrôle de flux permettent d'exécuter différents blocs d'instructions en fonction de conditions spécifiques. Ces structures de contrôle de flux incluent généralement les instructions conditionnelles et les boucles.

Définition

Les instructions qui permettent d'opérer un choix parmi différents blocs 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 bloc d'instructions qui suit la première condition si celle-ci est vraie et on sort, sinon, on exécute le deuxième bloc si c'est la deuxième condition qui est vérifiée et ainsi de suite, on avance profondément 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**

$a \leftarrow i^3$;

 Afficher(a) ;

Sinon

$b \leftarrow i \bmod 12$;

 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. En pseudocode, il y a deux concepts pour déclarer des modules dans un algorithme global : les formes procédurales et les formes fonctionnelles.

Définition

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

Proc Nom_procédure(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 ». 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

Algorithm

```
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 composant d'un vecteur.

Algorithm

```
% 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 écriture 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  $\leftarrow$  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.

Algorithmme

```

Fonc fact  $\leftarrow$  Factorielle (n)
local i ;
i  $\leftarrow$  0 ;
fact  $\leftarrow$  1;
Tant_que i < n faire
    i  $\leftarrow$  i+1 ;
    fact  $\leftarrow$  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*.

Algorithmme

```

Proc Echanger(A,i,j)
Local n,x ;
n  $\leftarrow$  longueur(A) ;
x  $\leftarrow$  A[i];

```

```

A[i] ← A[j] ;
A[j] ← x
Fin_proc.

```

8.5 Tri par insertion

Dans cet exemple, nous proposons d'implémenter une procédure de tri par insertion d'un tableau A de n éléments.

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 Amélioration de lisibilité et optimisation des algorithmes

La lisibilité d'un algorithme est un facteur important pour la réutilisabilité du code et des méthodes. Son amélioration est essentielle pour la découverte d'éventuelles erreurs de conception et pour assurer des évolutions futures. Par ailleurs, il existe plusieurs méthodes pour améliorer la complexité et la lisibilité d'un algorithme. Ainsi, on peut par exemple : analyser la complexité de l'algorithme, évaluer la complexité temporelle et spatiale de l'algorithme, simplifier la syntaxe et utiliser des conventions de nommage claires.

L'analyse de la complexité de l'algorithme est une étape qui a pour but de comprendre les différentes étapes de l'algorithme et de leur impact sur sa complexité. Elle permet également d'identifier les étapes qui peuvent être optimisées.

9.1 Evaluation de la complexité temporelle et spatiale de l'algorithme

L'évaluation de la complexité temporelle et spatiale d'un algorithme permet de quantifier la complexité de l'algorithme en termes de temps et d'espace. Pour ce faire, on peut utiliser des outils de profilage pour identifier les parties de l'algorithme qui prennent le plus de temps. Une fois ces parties identifiées, on peut tester différentes approches d'optimisation et mesurer leurs performances pour choisir la meilleure. En effet, viser la complexité la plus faible possible permet de comparer différentes versions de l'algorithme pour la même tâche et de choisir la version la plus efficace.

9.2 Simplification de la syntaxe et commentaires

La simplification de la syntaxe rend l'algorithme plus lisible et compréhensible. Pour ce faire, il est important d'utiliser une syntaxe concise et claire, d'éviter les répétitions et les redondances, et d'insérer des commentaires pour expliquer le fonctionnement de l'algorithme ou de certaines de ses parties.

9.3 Amélioration des conventions de nommage

L'utilisation de conventions de nommage claires rend l'algorithme plus lisible et compréhensible. En effet, les noms des variables et des fonctions doivent être clairs et explicites, afin de correspondre aux données qu'ils représentent ou aux opérations qu'ils effectuent. Par exemple, dans une affectation, il faut éviter d'utiliser des noms de variables trop longs ou ambigus.

De plus, les structures de données représentent des types de données abstraites qui permettent d'évoquer et de manipuler des données de manière efficace. Elles sont utilisées dans de nombreux algorithmes pour stocker et organiser les données. En employant des structures de données adaptées, on peut même contribuer à réduire la complexité temporelle et spatiale de l'algorithme.

10 Exercices de récapitulation

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

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

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

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

E 1.5 Ecrire en pseudocode un 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 un algorithme de la fonction qui produit la variance d'un tableau x de n nombres réels.

Chapitre 2 : Simulation de séquences de nombres au hasard

1 Notions de simulation

L'évolution rapide des ordinateurs, devenus de plus en plus rapides et performants, a ouvert la voie à une diversité de calculs extrêmement complexes. Ce progrès technologique a permis aux ingénieurs, statisticiens et chercheurs en sciences des données d'émuler artificiellement le hasard, grâce au développement de l'informatique et des méthodes de calcul avancées. Parmi ces méthodes, les techniques Monte-Carlo se distinguent en tant qu'approches essentielles pour résoudre des problèmes de simulation et de prédiction. Elles fusionnent habilement l'usage de nombres aléatoires avec les lois de probabilité, permettant ainsi de reproduire des réalités souvent complexes, où la décision est confrontée à l'incertitude.



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

Les méthodes Monte-Carlo, fondamentales en statistique, sont intrinsèquement liées aux applications informatiques et en finance. Leur objectif est de générer artificiellement des valeurs aléatoires supposées être des réalisations de suites de variables aléatoires indépendantes et uniformément distribuées dans l'intervalle $[0,1]$, servant ainsi d'entrées sous-jacentes à ces méthodes. La simulation Monte Carlo constitue ainsi un outil largement employé en recherche opérationnelle et en sciences de gestion. Elle permet de créer des modèles détaillés de systèmes complexes présents dans divers domaines, allant de la conception de jeux sur ordinateur à

l'optimisation de portefeuille en finance, en passant par la météorologie, la biologie, l'ingénierie nucléaire, l'épidémiologie, le transport et les réseaux informatiques de communication. Ce type de méthodes consiste à répéter artificiellement des expériences fictives à grande échelle dans le but de générer, à la vitesse de la machine, des séquences de nombres reflétant l'aspect aléatoire des phénomènes étudiés (Metropolis et Ulam, 1949). A l'origine, dans l'appellation « Méthodes Monte Carlo », le terme « Monte Carlo » désigne la composante aléatoire ou incertaine, faisant ainsi allusion aux jeux de hasard dans les casinos de la rue Monte Carlo à Monaco (Metropolis et Ulam, 1949 ; Verma, 2020). Les résultats issus de ces méthodes ont par la suite été compilés et réutilisés dans divers contextes décisionnels, illustrant ainsi leur influence et leur pertinence dans différents domaines de la recherche scientifique et de l'ingénierie (Mordechai, 2011).

2 Techniques de génération de séquences de nombres au hasard

Lorsque nous avons besoin de générer des échantillons de variables aléatoires selon une distribution spécifique (par exemple, une distribution de Bernoulli, de Poisson, normale, exponentielle, etc.), nous verrons plus tard qu'il est possible de transformer les séquences de nombres pseudo-aléatoires uniformément distribués en séquences de nombres suivant la distribution souhaitée. Ainsi, les méthodes de génération de séquences de nombres pseudo-aléatoires intervient dans toute simulation Monte Carlo.

Définition

Par le terme de simulation (ou méthode Monte-Carlo), nous désignons l'ensemble des techniques qui permettent de créer ou de reproduire artificiellement des suites de nombres suivant des distributions de probabilité préalablement fixées.

2.1 Méthodes

Classiquement, il existe deux grandes classes de méthodes pour générer des séquences de nombres aléatoires : les tables de nombres au hasard et l'usage des outils informatiques.

2.1.1 Tables de Nombres au Hasard

Les tables de nombres au hasard sont des ensembles organisés de suites de chiffres qui facilitent la sélection de nombres aléatoires ou de séquences de nombres aléatoires de manière méthodique et contrôlée.

Définition

Une table de nombres au hasard est une table de séquences finies de chiffres qui est utilisée pour simuler la loi uniforme sur l'ensemble des chiffres.

Cependant, la signification précise du terme « nombre aléatoire » dans ce contexte fait l'objet de débats. Plusieurs tables de nombres aléatoires ont été publiées dans la littérature. Parmi les plus célèbres, on peut citer la table de Tippett (Tippett, 1927), la table de Kendall et B-B. Smith (Kendall et al., 1938), la table de Fisher et Yates (Fisher et al., 1943) et la table de la RAND Corporation (Rand Corp., 1955). Ces tables étaient utilisées à diverses fins, notamment la simulation et l'échantillonnage. Bien que leur utilité se soit estompée avec l'avènement des ordinateurs et des calculatrices, elles demeurent un outil pédagogique précieux pour comprendre les concepts fondamentaux de l'échantillonnage statistique.

Table de Tippett (1927)

C'est la première table de nombres aléatoires qui a été publiée. Elle a été éditée pour la première fois en 1927 par L. H. C. Tippett, est un recueil de 10 400 nombres à quatre chiffres extraits des archives des recensements britanniques du 19^e siècle (Tippett, 1927).

Table de Kendall et Smith (1938)

Cette table a été publiée dans un article intitulé « *Randomness and Random Sampling Numbers* » (Kendall et B-B. Smith, 1938). Cet article décrit une méthode qui produit des séquences de nombres pouvant être assimilés à des échantillons de nombres aléatoires. Les auteurs y ont également recommandé quatre tests pour évaluer si les ensembles de nombres sont appropriés pour un échantillonnage aléatoire, et ont souligné l'importance de disposer de tableaux supplémentaires en plus de ceux de Tippett.

Table de Fisher et Yates (1943)

La table de Fisher et Yates fournit 7500 nombres de à deux chiffres, extraits des décimales situées entre la 15^e et la 19^e positions des nombres présents dans les tables de logarithmes d'A. J. Johnson. Ces nombres ont ensuite été transformés afin de réduire la fréquence du chiffre 6 qui était prédominant dans la suite de nombres d'origine (Fisher et al., 1943).

Table de la RAND Corporation (1955)

La table de la RAND Corporation comprend 1 000 000 de nombres générés à l'aide d'une machine similaire à une « *roulette électronique* ». Comme certains nombres étaient plus fréquemment

présents que d'autres, la série de ces nombres a été rééquilibrée dans la version publiée (Rand Corp., 1955).

Exemple

Les tables de Kendall et B-B. Smith sont des outils statistiques utilisés pour générer des nombres au hasard.

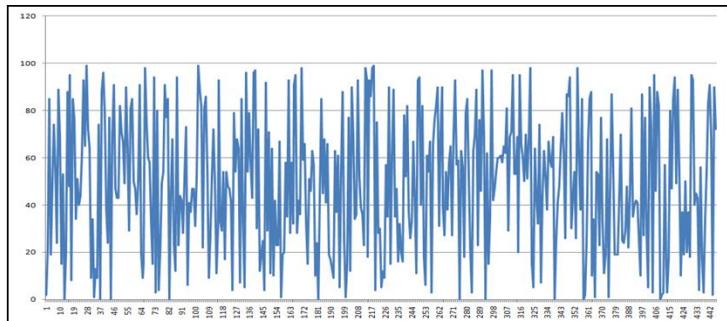
02 22 85 19 48 74 55 24 89 69 15 53 00 20 88 48 95 08
85 76 34 51 40 44 62 93 65 99 72 64 09 34 01 13 09 74
00 88 96 79 38 24 77 00 70 91 47 43 43 82 71 67 49 90
64 29 81 85 50 47 36 50 91 19 09 15 98 75 60 58 33 15
94 03 80 04 21 49 54 91 77 85 00 45 68 23 12 94 23 44
42 28 52 73 06 41 37 47 47 31 52 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 32 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 83 91 69 02 90 72

Pour lire les nombres à deux chiffres de cette table, il faut les parcourir un par un, ligne par ligne.

Figure 2.2 : Séquence de nombres extraits de la table de Kendall et B-B. Smith

Ensuite, cette séquence peut être représentée comme la trajectoire d'un processus stochastique. Les figures 2.3 représentent respectivement les données de la table présentée à la figure 2.2 et la table des nombres compris entre 0 et 1 obtenus en divisant l'ancienne séquence par 100.

Nombres d'après la table de Kendall & Babington Smith



Données uniformément distribuées sur l'intervalle $[0,1]$

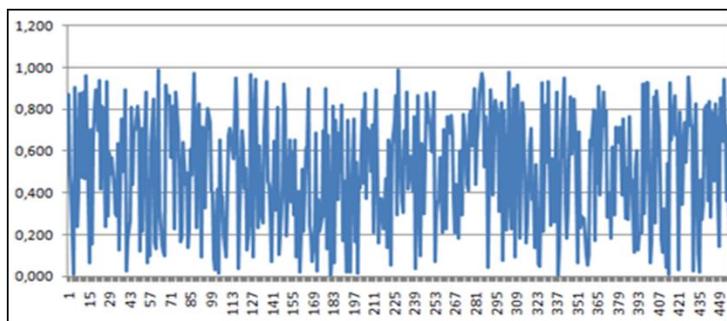


Fig. 2.3 : Comparaison des allures des données issues de tables et d'observation de la loi uniforme

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

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

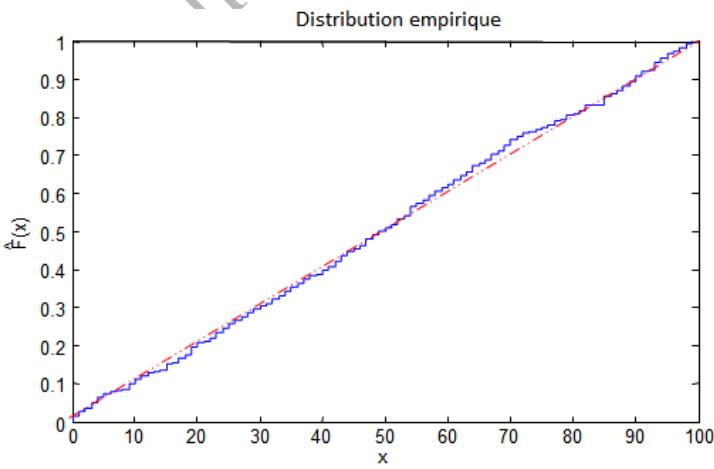


Fig. 2.4 : Comparaison de la distribution empirique des nombres tirés de la table de Kendall et B-B. Smith et de la distribution uniforme

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.

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 (nombre arbitraire);

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

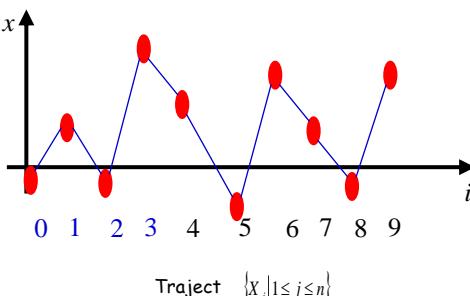


Fig. 2.5 : Trajectoire du générateur de nombres pseudo-aléatoires

Exemple

Dans cet exemple, dans l'idée de l'exploration des données simuler nous allons visualiser 500 nombres (Fig. 2.6) générés au hasard entre 0 et 1 par une méthode procédurale.

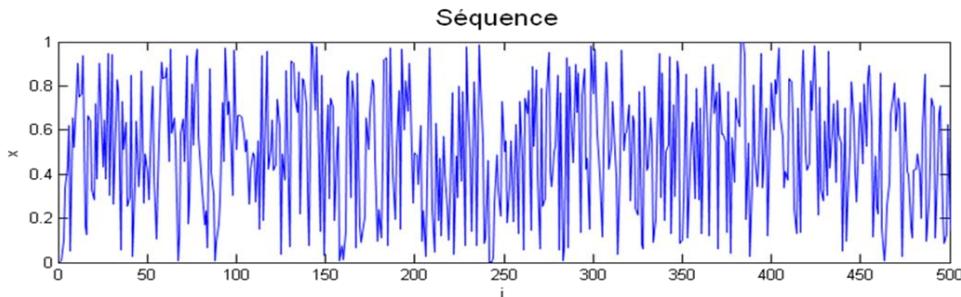


Fig. 2.6 : Suite uniforme de 500 nombres compris entre 0 et 1

Une fois que nous disposons des valeurs, nous pouvons les répartir en 100 catégories (par exemple) et créer un histogramme (voir Fig. 2.7) illustrant la répartition des valeurs par classe.

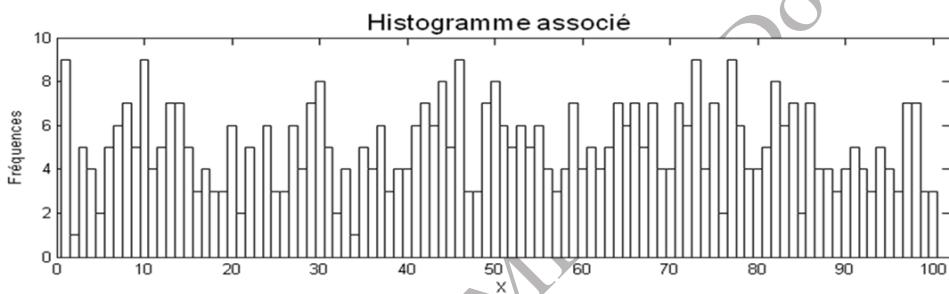


Fig. 2.7 : Histogramme associé des fréquences pour 100 classes de valeurs

L'histogramme associé des fréquences pour les classes de valeurs proposées offre une représentation visuelle détaillée de la répartition des données. En divisant ici l'ensemble de valeurs en 100 catégories distinctes, cet histogramme met en évidence la fréquence des occurrences dans chaque intervalle de valeurs. Chaque barre de l'histogramme représente une classe particulière, dont la hauteur correspond au nombre d'occurrences ou à la fréquence relative des valeurs tombant dans cette classe spécifique. Ce type de représentation graphique permet une analyse rapide et intuitive de la distribution des données, offrant ainsi des insights importants de l'apprentissage statistique.

2.2.2 Méthode du carré médian de Von-Neumann

La méthode du carré médian proposée par John von Neumann est l'une des premières méthodes conçues pour générer des nombres pseudo-aléatoires. L'algorithme suivant implémente cette méthode.

Algorithme 2.1

1. $i = 0$ et x_i un entier positif à quatre chiffres (Initialisations) ;
2. Elever au carré x_i^2 pour obtenir un entier comportant jusqu'à huit chiffres. Si le nombre de chiffres est inférieur à huit, ajoutez des zéros à gauche ;
3. Prenez les quatre chiffres du milieu de x_i^2 comme le nombre à quatre chiffres suivant, x_{i+1} ;
4. Insérez un point décimal à gauche de x_{i+1} pour en faire un nombre aléatoire u_{i+1} suivant $\mathcal{U}_{[0,1]}$;
5. Augmentez i de 1 ($i \leftarrow i + 1$) ;
6. Répétez le processus des étapes 2 à 5 jusqu'à la taille n de la séquence.

Exemple

- $x_0 = 2315 \Rightarrow x_0^2 = 05|3592|25$
- $x_1 = 3592 \Rightarrow x_1^2 = 12|9024|64$
- $x_2 = 9024 \Rightarrow x_2^2 = 81|4325|76$
- $x_3 = 4325 \Rightarrow x_3^2 = 18|7056|25$
- $x_4 = 7056 \Rightarrow x_4^2 = 49|7871|36$
- $x_5 = 7871, \dots$ etc.

Exercice

Ecrivez d'abord en pseudocode l'algorithme puis réaliser dans le langage Matlab (ou 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 générées par la méthode du carré médian pour différentes graines permet de mettre en lumière plusieurs observations significatives.

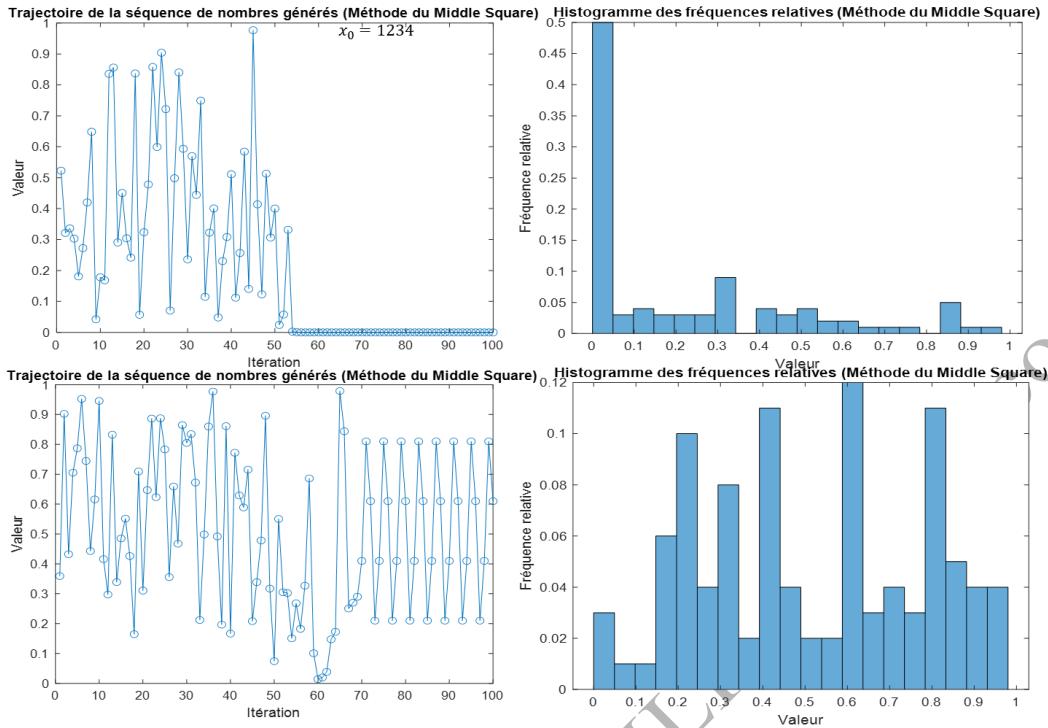


Fig. 2.8 : Visualisation de séquences de longueur 60 pour différents choix du germe.

Il est intéressant de noter que la méthode de von Neumann, malgré son utilité dans la génération de nombres pseudo-aléatoires, présente des caractéristiques spécifiques liées à sa périodicité et à son comportement stationnaire. En observant de plus près ses résultats, on constate que cette méthode peut devenir fortement périodique, voire même stationnaire.

2.2.3 Méthodes congruentielles

La méthode du carré médian de von Neumann a été remplacée en 1951 par des méthodes congruentielles 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 congruentiel, 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

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

Dans ce contexte, il est courant d'utiliser la terminologie suivante.

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 congruentiel 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 congruentiel considéré dans l'exemple 2 (précédent).
- Ecrivez le programme correspondant en Matlab (ou Octave) et produisez la visualisation des données générées pour différentes tailles N des séquences (par exemple : 50, 100, 500, 1000, ...).

- 3) Représentez enfin les histogrammes des différentes séquences chaque fois pour un nombre m différent des classes (10, 50, 100, 500, ...).
- 4) Que remarquez-vous lorsque la taille N de la séquence est « très grande ».
- 5) 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. $m/N >> 1/10$).

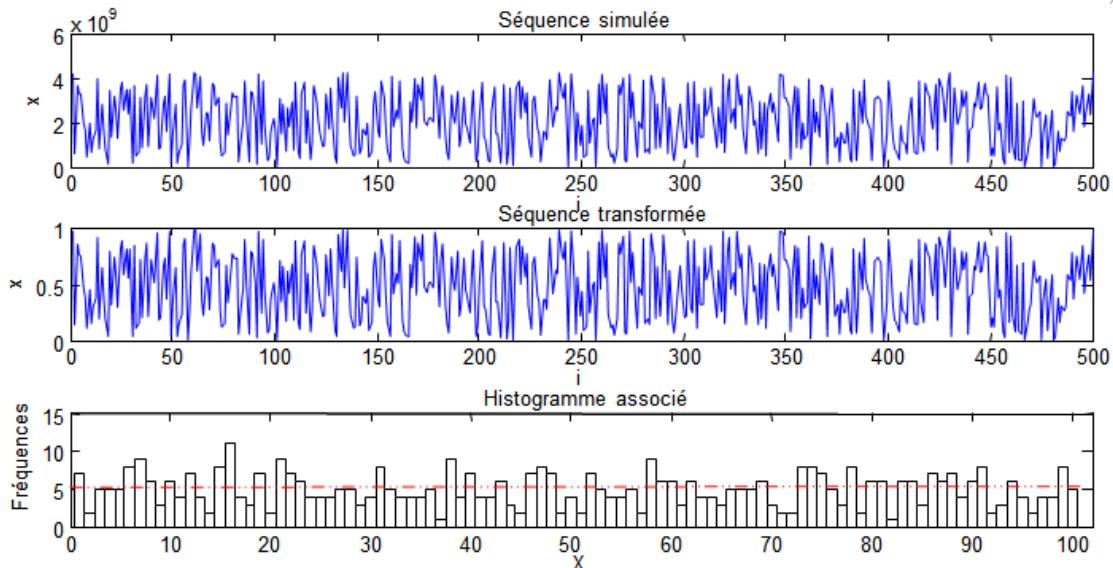


Fig. 2.9 : Exemples d'affichages pour l'exercice précédent

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 congruentielle, la période de la suite $(x_n)_{n \in \mathbb{N}}$ qu'elle définit.

2.2.4 Types de générateurs congruentiels

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

Définitions

On distingue trois types de générateurs congruentiels :

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 congruentielles 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 ont étudié les propriétés des générateurs de nombres pseudo-aléatoires basés sur des algorithmes congruents additifs et ont montré que, sous certaines conditions de régularité, la période des séquences générées peut atteindre la base modulo m (Hull et al., 1962).

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

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 $x_{n+1} = a x_n \pmod{m}$ (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 $a^{\left(\frac{m-1}{p}\right)} \neq 1 \pmod{m}$.
- 3) Si a est une racine primitive de $m - 1$ et si les entiers k et $m - 1$ sont premiers entre eux, alors, $a^k \pmod{m}$ 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 le nombre premier de Mersenne $m=2^{31}-1$ et tel que $a=7^5=16807$ est une racine primitive de $m-1$ possède, d'après le théorème 2, une période maximale, égale à $m-1$, c'est-à-dire,

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

Exemple 2 (Générateur de Lehmer)

Pour produire des nombres pseudo-aléatoires, le générateur de Lehmer utilise une méthode multiplicative basée sur la congruence modulo, avec les paramètres $m=2^{31}-1$ et $a=23$. On remarque que m est un nombre premier (nombre premier de Mersenne) et la période T de ce générateur

$$T = \frac{m-1}{2} = 1\,073\,741\,824$$

étant un diviseur de $m-1$, ce générateur est conforme à la proposition 1) du théorème 2.

3.3 Cas d'une base modulo une puissance de 2

Le théorème suivant énonce les conditions nécessaires et suffisantes pour maximiser la période d'un générateur multiplicatif lorsque la base modulo est une puissance de 2.

Théorème 3 (Admis)

Le générateur $x_i = a x_{i-1} \pmod{m}$ 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) $a \pmod{8} \in \{3, 5\}$

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

La discrépance est un concept utilisé en théorie des nombres et en analyse numérique pour mesurer à quel point un ensemble de points est uniformément distribué dans un espace donné. Lorsqu'on l'applique à une séquence générée par un générateur de nombres aléatoires, le terme discrépance a deux dimensions : une dimension philosophique révélant la divergence et le désaccord entre deux perceptions, et une dimension mathématique traduisant la capacité à remplir ou non l'espace dans toutes les directions.

4.1 Mesures de discrépance

Dans un premier lieu, on souhaite décomposer la séquence générée en des groupes de d termes $(u^{(1)}, u^{(2)}, \dots, u^{(d)})$ consécutifs de la série à n éléments. Ensuite, l'idée que nous souhaitons aborder ici concerne la notion de discrépance en dimension d , que nous allons définir ultérieurement comme la différence entre le nombre de points attendus et le nombre réel de vecteurs de dimension d parmi les points. $(u_i^{(1)}, u_i^{(2)}, \dots, u_i^{(d)}) \in]0,1[^d$, pour $i = 1..k$, tombant dans une région hyperrectangulaire, et que nous maximiserons sur toutes ces régions rectangulaires.

Dans cet objectif, commençons par établir la terminologie et les notations nécessaires à cette fin.

Notations et définitions

La terminologie suivante est d'usage en la matière :

- *On note $u \equiv (u^{(1)}, u^{(2)}, \dots, u^{(d)}) \in]0,1[^d$ (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 ce que n points soient équidistants les uns des autres dans l'intervalle $]0,1[$.

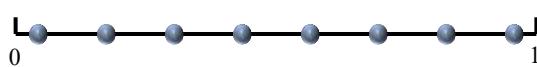


Fig. 2.10 : Points uniformément répartis en dimension $d = 1$.

En dimension deux, $d = 2$, on considère un ensemble P de k points de dimension d uniformément répartis (figure 2.11), tels que, on s'attend à ce que le nombre k_R de points de P situés dans l'hyperrectangle $R = [a_1, b_1] \times [a_2, b_2]$ soit approximativement $k_R = k \times \text{vol}(R)$.

Définition

Nous appelons :

- $D(P, R) = |k \times \text{vol}(R) - |P \cap R||$, écart de discrépance ;
- $D(P, \mathcal{R}) = \sup_{R \in \mathcal{R}} \{D(P, R)\}$, discrépance par rapport aux rectangles ;
- $\delta(n, \mathcal{R}) = \inf_{\substack{P \subset]0,1[^2 \\ |P|=k}} \{D(P, \mathcal{R})\}$, la plus petite discrépance à n points ;
- $\Delta(P)$, la finesse de recouvrement de discrépance (voir la figure 2.11).

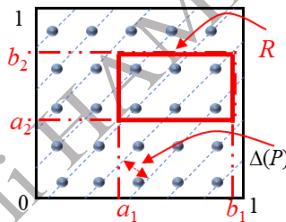


Fig. 2.11 : Représentation de la mesure de discrépance pour des points uniformément répartis en dimension $d = 2$.

Définition

Considérons un ensemble de points $u_i \equiv (u_i^{(1)}, u_i^{(2)}, \dots, u_i^{(d)})$, pour $i = 1..k$, de $]0,1[^d$, la discrépance mesure à quel point ces points sont répartis de manière uniforme dans l'hypercube $]0,1[^d$.

Dans ces conditions, un faible niveau de discrépance montre une répartition plus uniforme des points.

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 $\{u_i \in]0,1[\mid 1 \leq i \leq k\}$ est uniformément répartie sur $]0,1[$, si pour tout sous-intervalle $[a,b] \subset]0,1[$, on a,

$$\lim_{k \rightarrow +\infty} \left[\frac{1}{k} \left| \{u_1, u_2, \dots, u_k\} \cap [a, b] \right| \right] = b - a$$

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 $\{u_i \in]0,1[\mid 1 \leq i \leq k\}$ une séquence uniformément répartie sur $]0,1[$. Alors,

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

4.3 Discrépance étoile

La « discrépance étoile » est une mesure utilisée en statistique et en analyse de sensibilité pour évaluer la qualité d'un plan d'expériences ou de la répartition de l'échantillon. Elle permet de quantifier à quel point un ensemble de points est uniformément réparti dans un espace multidimensionnel. Ainsi, la discrépance étoile mesure à quel point les sous-ensembles de points dans l'espace multidimensionnel sont également répartis.

Définition

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

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

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 $\{u_k \mid k \in \mathbb{N}^\}$ est uniformément répartie sur $]0,1[^d$, si sa mesure de discrépance étoile est asymptotiquement nulle. i.e.*

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

4.4 Application

Idéalement une séquence obtenue à partir d'un générateur de nombres pseudo-aléatoires devrait être distribuée uniformément dans l'intervalle $]0,1[$. D'après l'analyse de discrépance en dimension deux, si cette propriété est effectivement vérifiée, les paires (u_i, u_{i+1}) de l'espace de dimension deux doivent 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  $\leftarrow$  Longueur(u);
Pour i  $\leftarrow$  1 à N - 1 Faire
    x[i]  $\leftarrow$  u[i];
    y[i]  $\leftarrow$  u[i+1];
Fin_pour ;
Dessiner((x,y))
Fin_proc.

```

Il est important de noter que, contrairement à l'intuition, les séquences de nombres pseudo-aléatoires générées à partir d'algorithmes ne sont pas toujours aussi aléatoires qu'on pourrait le penser. Au contraire, ces séquences peuvent souvent révéler des structures de discrépance, c'est-à-dire des schémas réguliers ou des non-uniformités dans leur répartition. Ces structures peuvent

provenir des propriétés intrinsèques de l'algorithme de génération, ce qui peut conduire à des résultats prévisibles et non aléatoires dans certaines situations. Cette observation met en lumière la complexité de la génération de nombres pseudo-aléatoires et souligne l'importance de choisir des méthodes de génération robustes et sophistiquées pour éviter de telles prévisibilités.

Le constat selon lequel les structures de discrépance sont fréquemment observées dans les séquences générées souligne un défaut majeur de la méthode de congruence simple. Cette méthode, bien qu'élémentaire et simple à mettre en œuvre, peut être sujette à des régularités dans les séquences générées, ce qui réduit son efficacité en tant que générateur de nombres aléatoires. La présence de ces structures compromet la qualité de l'aléa obtenu, ce qui peut être problématique dans des applications sensibles où l'aléatoire est essentiel, comme en cryptographie ou en simulation Monte-Carlo. Par conséquent, comme nous l'avons déjà signalé, il est indispensable d'employer des algorithmes plus subtiles pour générer des séquences pseudo-aléatoires présentant une meilleure qualité aléatoire et une distribution empirique plus conforme (très proche de la distribution uniforme).

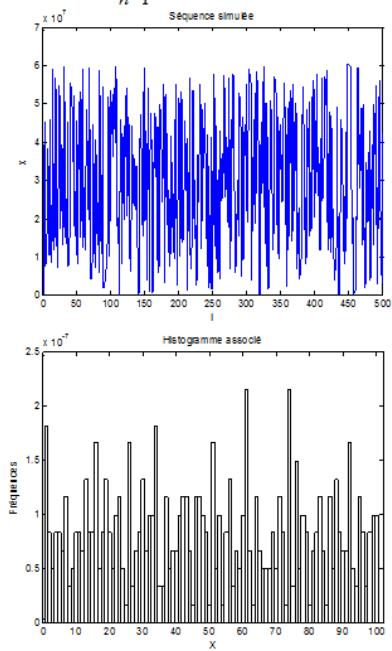
4.5 Etude d'exemples

Dans les exemples suivants, les indices statistiques de position et de variation sont calculés pour les séquences générées, et différentes analyses d'adéquation visuelle ainsi que de discrépance sont effectuées. L'objectif est d'analyser l'uniformité de la séquence produite par un générateur de nombres pseudo-aléatoires donné. Ainsi, l'histogramme de la séquence considérée sera représenté, permettant d'observer visuellement s'il y a des classes plus ou moins uniformément remplies que les autres, ainsi que la vérification visuelle de l'existence de structures de discrépance, en dimension deux, plus ou moins apparentes.

1) Générateur avec une forte structure de discrépance grossière

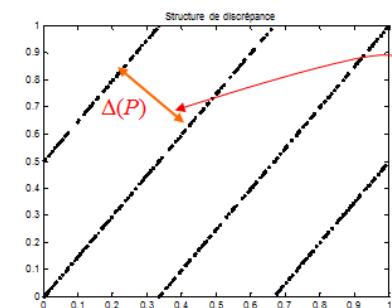
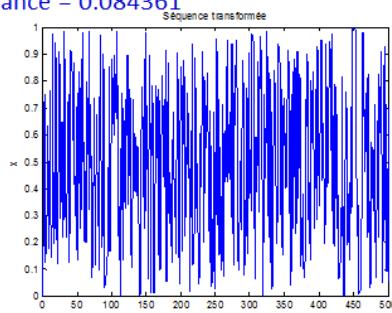
$$x_n = 30233086 x_{n-1} \bmod 60466169$$

Faible périodicité
Fort discrépance
Très faible finesse de recouvrement



Moyenne = 0.48933

Variance = 0.084361

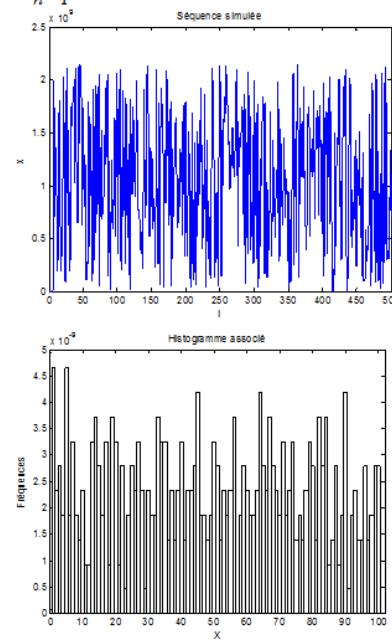


Finesse du
recouvrement de
discrépance

2) Générateur avec une forte structure de discrépance fine

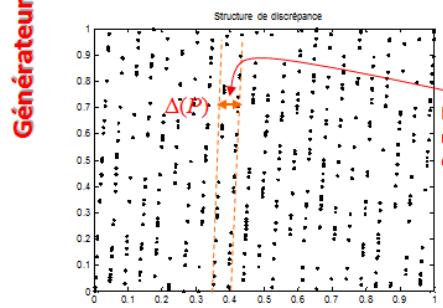
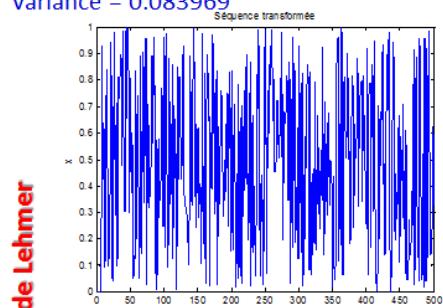
$$x_n = 23 x_{n-1} \bmod 2^{31}-1$$

Faible périodicité
Discrépance modérée
Assez bonne finesse de recouvrement



Moyenne = 0.4891

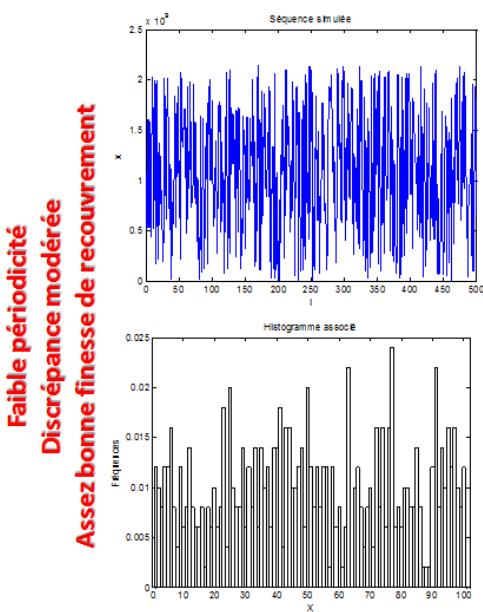
Variance = 0.083969



Finesse du
recouvrement de
discrépance

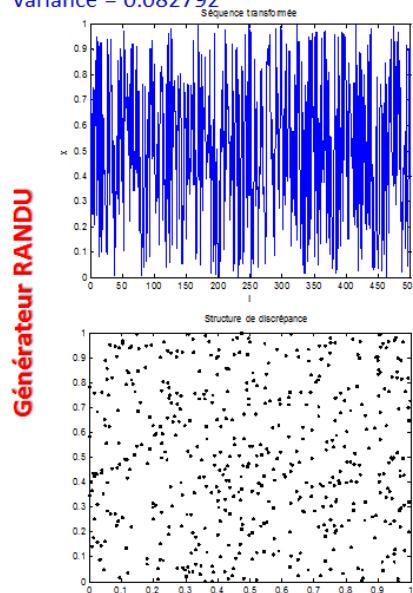
3) Générateur où la structure de discrépance n'est pas évidente (ou très apparente)

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



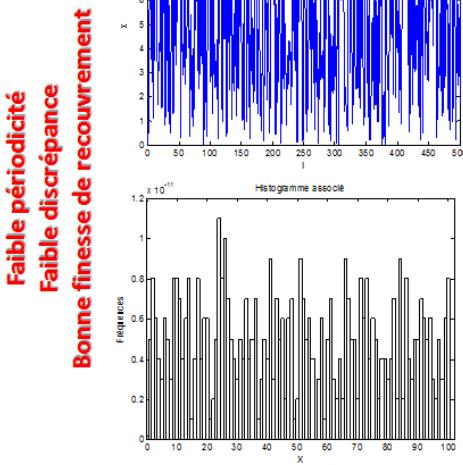
Moyenne = 0.5065

Variance = 0.082792



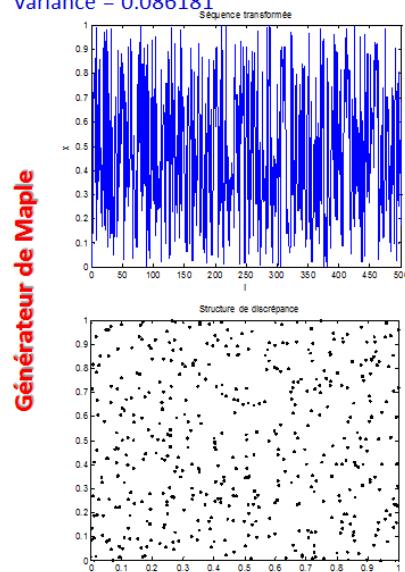
4) Générateur où la structure de discrépance n'est pas très évidente

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



Moyenne = 0.48621

Variance = 0.086181



5 Amélioration de la qualité des générateurs congruentiels

L'amélioration de la qualité des générateurs congruentiels est essentielle pour garantir des séquences de nombres pseudo-aléatoires de haute qualité. Nous, avons vu qu'il y a plusieurs méthodes qui peuvent être appliquées dans ce but sur le même type de générateur déployé. D'abord,

le choix judicieux des paramètres du générateur. Nous avons vu que les valeurs des paramètres dans l'algorithme congruentiel doivent être soigneusement sélectionnées. Des valeurs bien choisies peuvent contribuer à réduire les motifs périodiques et améliorer la qualité de la séquence générée. Ainsi, l'utilisation des nombres premiers comme base modulo peut augmenter la période du générateur et réduire les corrélations des séquences. De plus, incorporer des procédures de mélange ou des permutations dans le processus de génération peut augmenter la période des séquences, les rendant ainsi plus aléatoires. En combinant ces approches et en restant attentif aux avancées dans le domaine de la génération de nombres pseudo-aléatoires, il est possible d'améliorer significativement la qualité des générateurs congruentiels, les rendant ainsi plus fiables et mieux adaptés à diverses applications. L'objectif de la présente section est d'introduire plus sophistiquées pour obtenir des séquences de nombres avec une plus grande période et une amélioration de la discrépance en dimension deux.

5.1 Méthode du registre à décalage avec rétroaction linéaire

En dehors de la méthode de Lehmer, il existe d'autres méthodes de génération de nombres pseudo-aléatoires. Une de ces approches consiste à utiliser des formes alternatives de congruence, telles que l'équation

$$x_{n+1} = x_n (1 + x_n) \bmod 2^\alpha \text{ avec } x_0 \bmod 4 = 2$$

ou encore en employant des équations de la forme

$$x_n = (x_{n-j} + x_{n-k} + b) \bmod m \text{ avec } j \text{ et } k \text{ comme des valeurs fixes.}$$

En particulier, G.J. Mitchell et D.P. Moore sont connus pour leur contribution par le développement du générateurs de nombres pseudo-aléatoires

$$x_n = (x_{n-24} + x_{n-55}) \bmod m$$

en 1958 (Knuth, 1998). Cette méthode a été soumise à de nombreux tests statistiques qu'elle a passé avec succès. En outre, elle présente une très longue période pour $m = 2$, $T = 2^{55} - 1$ et pour $m = 2^\alpha$, $T = 2^{\alpha-1} (2^{55} - 1)$. Cependant, la principale lacune de ce générateur réside dans l'absence d'un fondement théorique solide. Dans ce cas encore, les valeurs de départ jouent un rôle essentiel.

Si ce générateur n'est initialisé qu'avec des valeurs nulles, la séquence générée sera constamment égale à zéro.

La méthode du registre à décalage avec rétroaction linéaire (*LFSR* pour *Linear Feedback Shift Register* en anglais) est en réalité une généralisation des méthodes congruentielles classiques. Elle s'écrit sous la forme d'une récurrence linéaire où chaque bit de la séquence pseudo-aléatoire est calculé en combinant linéairement un certain nombre de bits précédents du registre à décalage. Cette méthode permet de générer des séquences pseudo-aléatoires de manière efficace et rapide, en utilisant des opérations de décalage et des opérations de \oplus (XOR/OU-exclusif) entre les bits du registre. Le générateur s'énonce de la forme

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

avec $a_i \in \{0,1\}$, pour $i = 0..k$.

Pour déterminer la valeur à l'étape n suivant ce générateur, le processus nécessite la mémorisation des k étapes antérieures $(x_n; x_{n-1}, x_{n-2}, \dots, x_{n-k})$. De plus, toutes les séquences générées par ce modèle deviennent inévitablement périodiques au-delà d'un certain point, conformément à la nature des modèles congruentIELS. Ainsi, remarquons que si $k = 1$, on retrouve une méthode congruentIELLE additive classique.

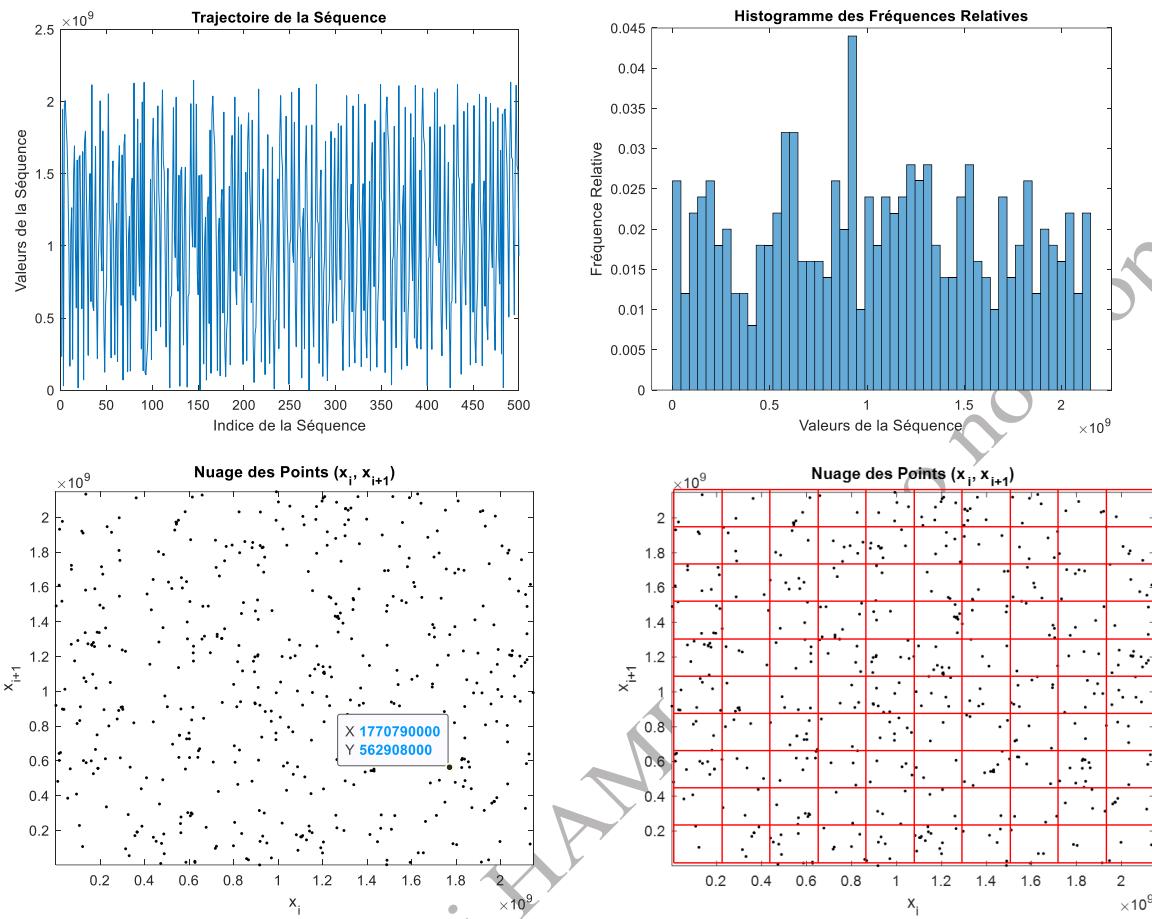
Dans le cas général, l'espace des états est $(\mathbf{Z}/m\mathbf{Z})^k$, avec $|(\mathbf{Z}/m\mathbf{Z})^k| = m^k$. Ces modèles sont utilisés en cryptographie pour concevoir des séquences de nombres pseudo-aléatoires. Les paramètres du modèle doivent être calibrés de manière à ce que l'allure de la séquence qui en découle exhibe la plus grande période possible tout en maintenant des coûts de calcul et de stockage raisonnables.

Exemples

1) Le générateur de Fibonacci est basé sur la suite de Fibonacci modulo la valeur maximale m désirée. Il secrét

$$x_n = (x_{n-1} \oplus x_{n-2}) \bmod m \text{ avec les valeurs initiales } x_0 \text{ et } x_1 \text{ en entrée}$$

Ainsi, dans le cas où $m = 2^{31} - 1$, nous pouvons tracer les performances de ce générateur



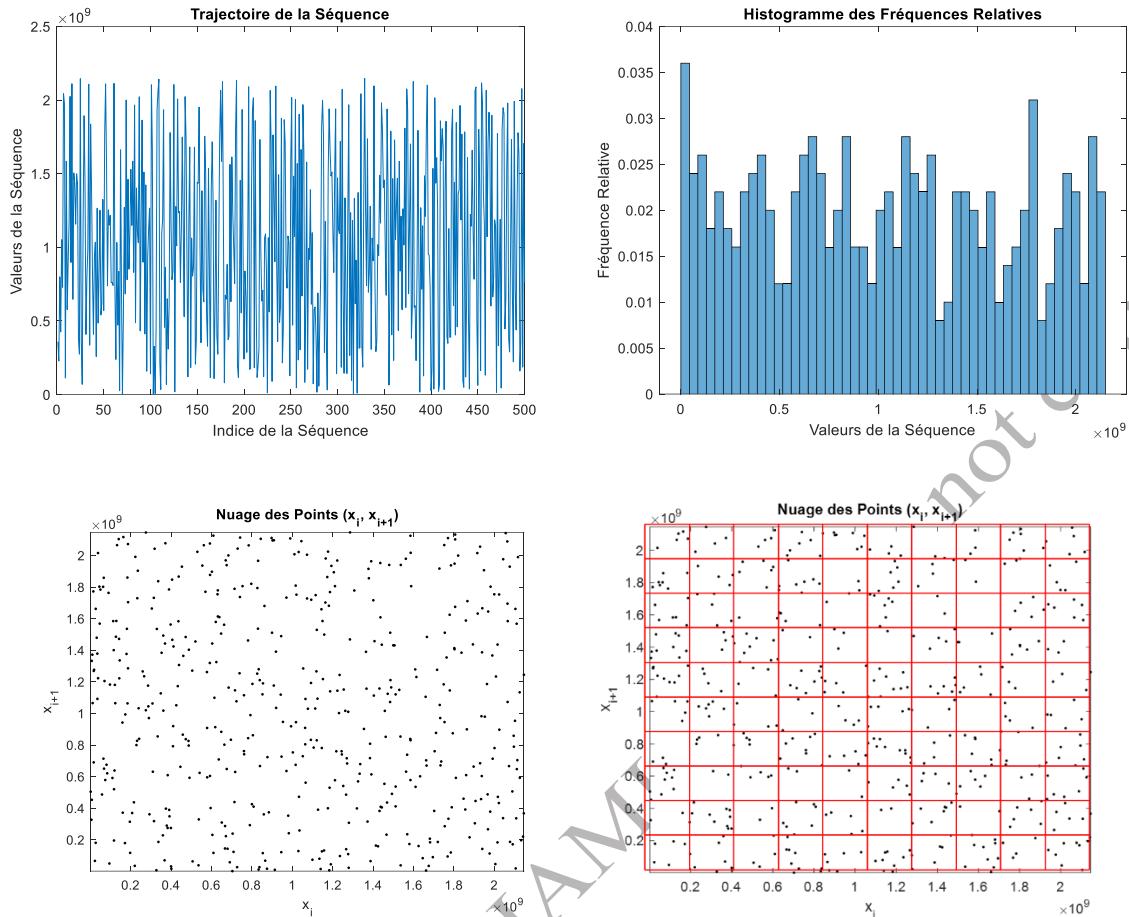
En général, on peut également employer une variante plus générale

$$x_n = (x_{n-1} + x_{n-k}) \bmod m \text{ avec les valeurs initiales } x_0, x_1, \dots \text{ et } x_{k-1} \text{ en entrée}$$

Ainsi, la qualité du générateur dépend de k et des nombres x_0, x_1, \dots et x_{k-1} utilisés pour initialiser la suite.

2) Revenons au modèle de G.J. Mitchell et D.P. Moore

$$x_n = (x_{n-24} \oplus x_{n-55}) \bmod 2^{31}$$



Tous ces générateurs sont faciles à implémenter et demandent peu de ressources pour les mettre en marche. Les générateurs de type registre à décalage avec rétroaction linéaire sont particulièrement intéressants en raison de leur efficacité exceptionnelle du point de vue du temps de calcul. Leur rapidité d'exécution en fait des outils privilégiés dans de nombreuses applications informatiques, notamment dans les domaines qui exigent la génération rapide et continue de séquences pseudo-aléatoires. Cette rapidité de traitement est déterminante pour accomplir des tâches telles que la simulation Monte-Carlo, l'analyse de systèmes complexes, la cryptographie et d'autres applications nécessitant la génération en temps réel d'un grand nombre de nombres pseudo-aléatoires.

5.2 Méthodes congruentielles vectorielles

L'amélioration des générateurs de nombres pseudo-aléatoires est souvent entreprise dans le but de prolonger leurs périodes, permettant ainsi de générer des séquences « indépendantes » de plus en plus longues.

L'approche que nous considérons ici consiste à coupler plusieurs générateurs congruentiels 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 et sont ensuite combinés. Ce type de méthodes donne lieu à une vaste famille de générateurs pouvant être utilisés pour produire des vecteurs uniformes de dimensions variées en combinant différents générateurs. En effet, l'espace des états de ces 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, en choisissant judicieusement les paramètres du modèle et en utilisant des bases modulo appropriées, m_i (pour $i = 1..k$) très élevées, nous pouvons espérer obtenir des séquences de très longues périodes.

Les générateurs vectoriels reposent sur le calcul matriciel linéaire, les rendant ainsi particulièrement adaptés au calcul graphique (GPU) ainsi qu'aux ordinateurs vectoriels et multiprocesseurs.

Exemple

Le générateur vectoriel proposé par L'Écuyer est un exemple remarquable dans ce contexte.

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

Ce générateur présente de bonnes propriétés en termes de qualité et d'efficacité. Dans ce cas,

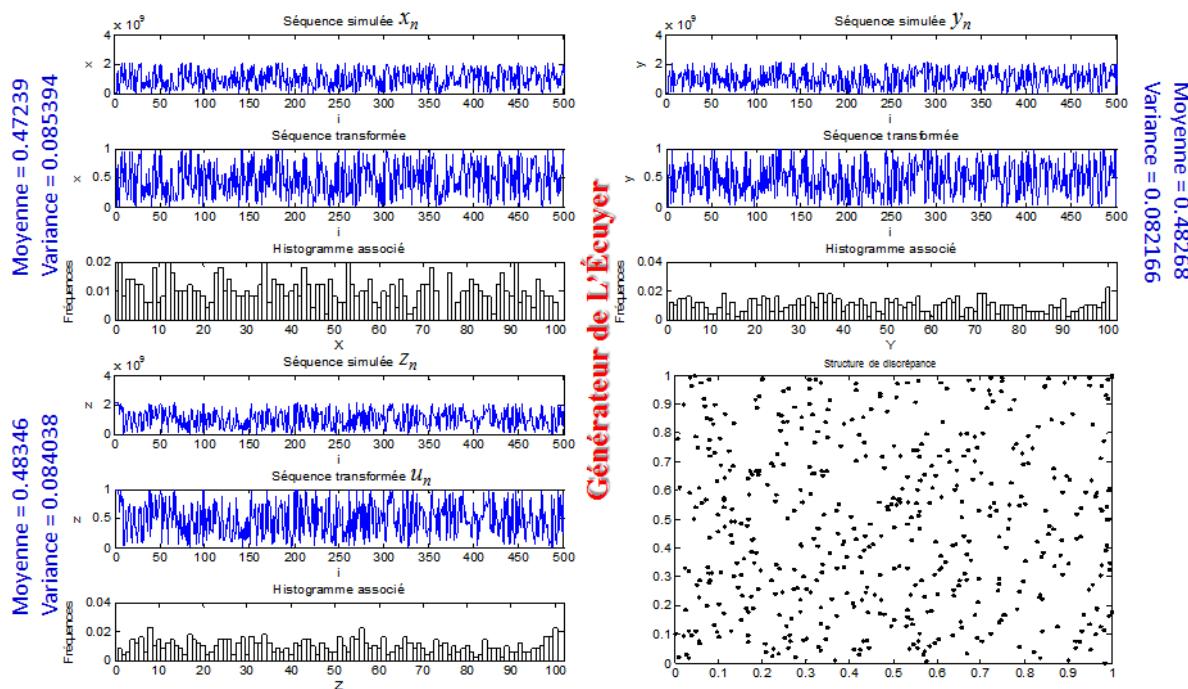
$$\mathbf{X}_n = \begin{pmatrix} X_n \\ Y_n \end{pmatrix}, \quad \mathbf{m} = \begin{pmatrix} 2^{31} - 85 \\ 2^{31} - 249 \end{pmatrix} \text{ et } \mathbf{C} = \frac{1}{2^{31} - 86} \times (1, -1).$$

En utilisant des techniques avancées basées sur le calcul matriciel, ce générateur peut générer des séquences pseudo-aléatoires de haute qualité, adaptées à un large éventail d'applications. Les méthodes vectorielles permettent de générer simultanément plusieurs nombres aléatoires, améliorant ainsi l'efficacité du processus de génération.

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 est ainsi

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

de l'ordre de 2 milliards de milliards.



Pour ce type de générateurs, à mesure que la taille des groupes mélangés augmente, la structure de discrépance tend à disparaître, ce qui signifie que les paires successives suivent une distribution uniforme. Cependant, cela s'accompagne d'un temps de calcul beaucoup plus important par rapport à l'amélioration obtenue par la méthode du registre à décalage avec rétroaction linéaire.

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

Lorsque les méthodes numériques traditionnelles pour le calcul d'intégrales se révèlent difficiles, l'utilisation de méthodes impliquant la génération de nombres aléatoires offre une alternative pratique. Ces applications sont généralement désignées sous le nom de « *méthodes d'intégration Monte-Carlo* ». Elles jouent un rôle essentiel dans le calcul bayésien des probabilités, notamment lorsqu'il s'agit d'évaluer des intégrales multidimensionnelles complexes. L'une des raisons de leur utilité réside dans leur capacité à gérer des espaces de grande dimension, où d'autres méthodes numériques peuvent rencontrer des difficultés. En utilisant des techniques de génération de nombres aléatoires, ces méthodes échantillonnent l'espace d'intégration de manière aléatoire et fournissent des estimations numériques des intégrales. Cette approche probabiliste permet d'obtenir des résultats fiables même dans des espaces de grande complexité. L'intégration Monte Carlo est appréciée non seulement par les statisticiens et les chercheurs en probabilités, mais aussi dans une gamme variée de domaines. Elle s'applique avec succès à des secteurs aussi divers que l'assurance qualité et la fiabilité des logiciels, l'évaluation des performances et le dimensionnement des réseaux informatiques, la prévision des marchés financiers, ainsi que dans les domaines avancés de l'apprentissage automatique et de l'intelligence artificielle.

Pour simplifier la présentation des méthodes d'intégration Monte-Carlo, nous proposons de commencer par le calcul d'intégrales unidimensionnelles. Cela permettra d'établir des bases solides et compréhensibles des concepts fondamentaux de la méthode. En se limitant à une seule dimension, nous pourrons exposer les principes essentiels de manière claire et concise, offrant ainsi une compréhension approfondie des méthodes employées. Une fois ces concepts maîtrisés, nous pourrons considérer la méthode dans le cadre multidimensionnel, assurant ainsi une transition fluide vers des espaces de dimensions supérieures.

6.1 Forme canonique

Considérons une fonction g définie sur les nombres réels dans l'intervalle $[0,1]$, supposons que cette fonction soit intégrable sur cet ensemble. Nous proposons de calculer numériquement l'intégrale de cette fonction

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

L'application de la méthode Monte-Carlo consiste à considérer une variable aléatoire U uniformément distribuée sur $[0,1]$ et de remarquer 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 et identiquement distribuées, avec une moyenne commune égale à I , ainsi qu'une variance égale $\sigma^2 = \text{var}[g(U)]$. En effet, selon la loi faible des grands nombres, nous pouvons écrire

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

Autrement dit, lorsque k est suffisamment grand, \hat{I}_k constitue une estimation (ou approximation) de I en termes de convergence en probabilité de la suite des $(\hat{I}_k)_{k \in \mathbb{N}^*}$ vers I . i.e.

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

Ainsi, en générant un assez grand nombre k de nombres pseudo-aléatoires u_1, \dots, u_k dans l'intervalle $[0,1]$, nous sommes pratiquement sûrs que I est la moyenne des $g(u_1), \dots, g(u_k)$. En effet, sauf si l'échantillon est extrêmement large, il est souvent ardu de savoir si la précision de la théorie asymptotique en probabilité est suffisante pour interpréter nos résultats en toute confiance.

Algorithme

Proc Integrale_simple(k)

local i, sum, u ;

sum \leftarrow 0 ;

Pour i \leftarrow 1 à k **Faire**

 u \leftarrow RND() ;

 sum \leftarrow sum + g(u)

```

Fin_pour ;

Retourner(sum/k) % Estimation de l'intégrale comme moyenne des évaluations de g(u)

Fin_proc.

```

Nous supposons que la fonction « RND() », préalablement définie dans le contexte de ce pseudocode, génère des nombres aléatoires uniformément distribués dans l'intervalle [0,1]. Cette fonction est essentielle pour l'implémentation de la méthode de Monte-Carlo, car elle permet de créer des valeurs aléatoires nécessaires à l'évaluation de la fonction à intégrer à différents points de l'intervalle. L'utilisation de cette fonction garantit l'aléatoire et l'imprévisibilité des échantillons, fondamentaux pour l'exactitude de l'approximation numérique de l'intégrale.

Question d'implémentation

Réaliser dans Matlab (ou Octave) un programme qui implémente cette méthode d'une fonction g intégrable sur [0,1].

Exemple

Soit à calculer

$$I = \int_0^1 x^3 \times \log(1 + \cos x) dx$$

La fonction $g : x \mapsto x^3 \times \log(1 + \cos x)$ étant composée de produits de fonctions bornées sur [0,1], elle est également bornée sur [0,1]; ainsi, g est intégrable sur [0,1]. On peut ainsi écrire, en Matlab, conformément au pseudocode de l'algorithme du calcul de l'intégrale Monte Carlo précédemment présenté.

```

function estimation = MonteCarloIntegration()
% Nombre d'échantillons
N = 100000;
% Initialisation de la somme
sum = 0;
% Génération des échantillons et calcul de la somme
for i = 1:N
    % Génération d'un nombre aléatoire dans l'intervalle [0,1]
    x = rand();
    % Évaluation de la fonction g(x)
    g = x^3 * log(1 + cos(x));
    % Ajout à la somme partielle
    sum = sum + g;

```

```

end
% Estimation de l'intégrale en utilisant la méthode de Monte-Carlo
sum = (1/N) * sum;
% Affichage du résultat
disp(['Estimation de l''intégrale: ', num2str(estimation)]);
end

```

Ainsi, à l'exécution de ce script, nous avons

```
>> MonteCarloIntegration
```

```
Estimation de l'intégrale de g: 0.13022
```

Le script MATLAB précédent, implémentant la fonction « MonteCarloIntegration », peut être amélioré en utilisant la syntaxe spécifique et les fonctions prédéfinies de ce langage de programmation. Ces optimisations permettent d'exécuter le code de manière plus efficiente. Ci-après, une version optimisée.

```

function estimation = MonteCarloIntegration_opt()
% Nombre d'échantillons
N = 100000;
% Génération des nombres aléatoires dans l'intervalle [0,1]
x = rand(1, N);
% Évaluation de la fonction g(x) pour tous les échantillons en une seule opération
g = x.^3 .* log(1 + cos(x));
% Estimation de l'intégrale en utilisant la méthode de Monte-Carlo
estimation = mean(g);
% Affichage du résultat
disp(['Estimation de l''intégrale de g: ', num2str(estimation)]);
end

```

6.2 Cas général des deux bornes finies

Dans le contexte général où les bornes sont limitées, pour évaluer l'intégrale,

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

nous devrons recourir à une transformation de variables

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

soit $du = dx / (b - a)$. Ainsi, nous obtenons la forme canonique

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

où $h(u) = (b-a) g[a + (b-a) u]$.

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

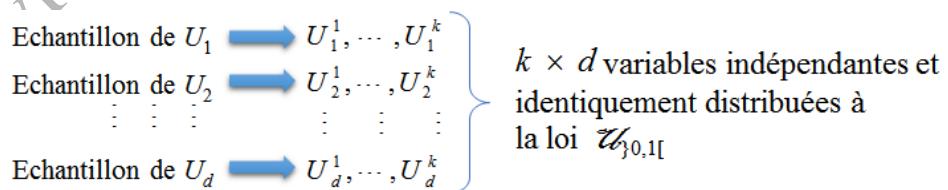
Des changements de variables adéquats permettent de ramener les calculs au cas canonique du paragraphe 5.1 lorsque l'une ou même les deux bornes de l'intégrale sont infinies. Cette question fait l'objet de l'exercice **E 2.7** ci-après.

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 reste la même, égale à $1/\sqrt{n}$, quelle 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 déterministes classiques, telles que la méthode des trapèzes par exemple.

En revanche, dans le cas multidimensionnel ($d > 1$), la vitesse de convergence de l'intégration Monte-Carlo reste de l'ordre de $1/\sqrt{n}$, tandis 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 (c'est-à-dire que les dérivées d'ordre inférieur ou égal à s sont toutes bornées). Ainsi, la méthode Monte-Carlo devient compétitive par rapport aux méthodes d'intégration déterministes dès que la dimension d'intégration dépasse deux fois le seuil s

$$d > 2s$$

Dans ces conditions, l'usage de la méthode Monte-Carlo utilisant une fonction d'importance pour le calcul d'intégrales multiples de haut ordre devient pertinent.

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 $X_1 = x_1, X_2 = x_2, \dots, X_{i-1} = x_{i-1}$:

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

où $f_{\langle i \rangle}$ 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,

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

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 $\langle g_1, g_2 \rangle = \int_{\mathbb{R}^k} g_1(x) g_2(x) dx$ 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 I . 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 de Matlab (ou 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 congruentiel défini par $u_{n+1} = 23u_n + 17 \bmod 256$.

- Ecrire en Octave (ou Matlab) une fonction qui implémente ce générateur.
- Que se passe-t-il dans chacun des cas lorsque $u_0 = 10$, $u_0 = 11$, $u_0 = 20$, $u_0 = 113$?
- 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 congruentiels 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 de ce que vous avez vu dans le cours, sur des séquences de longueur 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 congruentiel défini par le terme général $u_{n+1} = 31415821 u_n + 1 \bmod 108$ et la valeur initiale $u_0 = 1$.

- Ecrivez en Octave (ou Matlab) la fonction qui implémente ce générateur.
- Générer la séquence des 20 premiers termes de ce générateur.
- Que remarquez-vous à propos du dernier chiffre des nombres générés ?
- Expliquez ce qui se passe.

E 2.5 On définit les trois générateurs congruentiels multiplicatifs suivant

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

et la suite croisée de terme général

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

- Identifiez la matrice \mathbf{A} , le vecteur \mathbf{C} et l'expression de la suite \mathbf{X}_n qui formalisent ce modèle vectoriel sous la forme

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

en considérant

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

- Implémentez en Matlab (ou Octave) 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 π)

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.

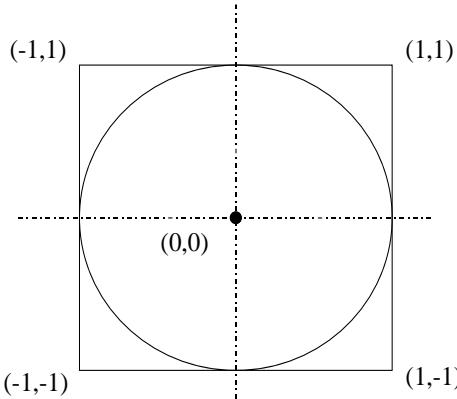


Figure 2.10 : Cercle inscrit dans un carré

Sur la base de ces éléments de modélisation, donnez un algorithme permettant de calculer une valeur approximative de π .

E 2.7 Trouvez, dans chacun des cas, la fonction h permettant de ramener l'intégrale considérée sous une forme canonique pour le calcul Monte-Carlo. Puis, écrivez ensuite en pseudocode l'algorithme correspondant à chacun des cas avant d'écrire en Matlab permettant de calculer numériquement les intégrales suivantes :

a) $I_1 = \int_7^8 \cosh(x^5 + 3) dx$

b) $I_2 = \int_0^{+\infty} \log(e^{-x} + 10) dx$

c) $I_3 = \int_0^4 \int_{-1}^1 \int_5^{10} \log(x_1 + x_2^2 x_3) dx_1 dx_2 dx_3$

E 2.8 Dans chacun des trois cas suivants, 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) $I = \int_0^{+\infty} \int_0^x \int_0^y \exp[-(x + y^2 + 2z)] dz dy dx .$

E 2.9 Montrez comment pourrait-on calculer, pour des valeurs adéquates de a et b , à l'aide de la méthode Monte-Carlo l'intégrale

$$I(a,b) = \int_a^b \log(1+x^2) \exp(-x^2) dx$$

- a) En appliquant la méthode Monte-Carlo, proposez en pseudocode un algorithme pour calculer $I(0,1)$;
- b) Produisez un script en Matlab (ou Octave) pour évaluer la valeur de cette dernière intégrale.

Chapitre 3 : Simulation de variables aléatoires réelles discrètes

1 Notions de simulation de variables aléatoires réelles discrètes

Dans l'étude des variables aléatoires réelles, leur caractérisation repose sur divers critères de discrimination. Une méthode couramment employée consiste à les différencier en fonction de l'ensemble de leurs valeurs possibles, c'est-à-dire les valeurs pour lesquelles la fonction de probabilité associée à la variable aléatoire n'est pas nulle. Cet ensemble est désigné comme le support de la variable aléatoire. Ainsi, les variables aléatoires peuvent être classées en deux catégories distinctes : les variables aléatoires discrètes, caractérisées par un support discret, et les variables aléatoires continues, dont le support est continu.

En se fondant sur ce critère de discrimination, il est possible d'effectuer une subdivision supplémentaire parmi les variables aléatoires discrètes réelles. Celles dont le support est fini se distinguent des variables aléatoires discrètes réelles dont le support est infini. Cette distinction revêt une importance particulière du point de vue de l'implémentation informatique. En effet, dans le cas des variables aléatoires discrètes à support fini, la gestion du stockage des valeurs du support est relativement directe et efficace.

Cependant, lorsque le support d'une variable aléatoire discrète réelle est infini, la situation se complexifie sur le plan informatique. Cependant, des études récentes ont montré que moyennant une certaine tolérance en termes d'erreur de probabilité, il est possible de ramener l'implémentation informatique de ces variables aléatoires discrètes réelles à support discret infini à un cadre comparable à celui des variables aléatoires à support fini. Ces travaux ouvrent ainsi des perspectives intéressantes pour la gestion informatique des variables aléatoires discrètes à support infini, permettant ainsi d'étendre leur utilité pratique dans divers domaines d'application.

2 Méthode de transformation inverse

En statistiques et en théorie des probabilités, une variable aléatoire discrète est une variable aléatoire qui peut prendre un nombre fini ou dénombrable de valeurs distinctes. Le support d'une variable aléatoire discrète est l'ensemble des valeurs pour lesquelles la variable aléatoire a une probabilité non nulle.

Définition

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

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

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

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

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 $U \sim U_{]0,1[}$ 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)  $\wedge$  (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 ← 1 à N Faire
    seq[i] ← 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 ← Rnd();
Si u < p Alors x ← 1
    Sinon x ← 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 ← 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.

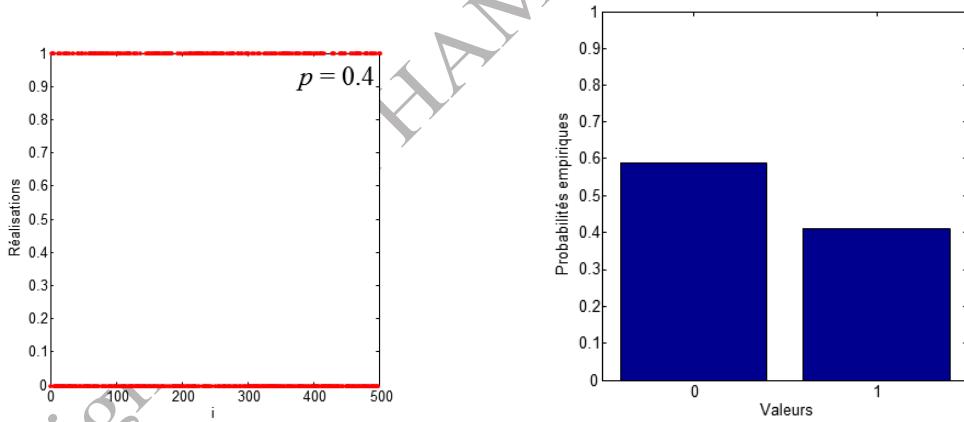


Fig. 3.1 : Visualisation des résultats de simulation d'une variable aléatoire de Bernoulli

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_ε 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_ε , 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 N0

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

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

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

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

Algorithme

Proc Simul_Poisson(lam,eps)

Local i,F,pr,x,p:

pr \leftarrow exp(lam); i \leftarrow 0; p[1] \leftarrow pr; F \leftarrow pr;

Tant que F $<$ 1-eps **Faire**

i \leftarrow i+1; pr \leftarrow pr*lam/i; p[i] \leftarrow 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,eps)
```

```
local i , seq ;
```

```
Pour i ← 1 à N Faire
```

```
seq[i] ← Simul_Poisson(lambda,eps)
```

```
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.

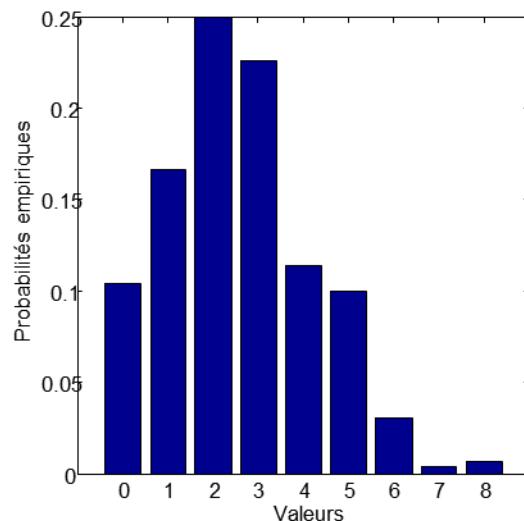
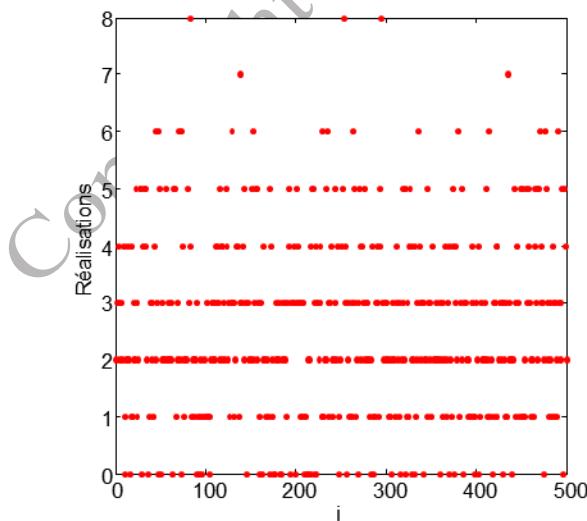


Fig. 3.2 : Visualisation des résultats de simulation d'une variable aléatoire de Poisson

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).

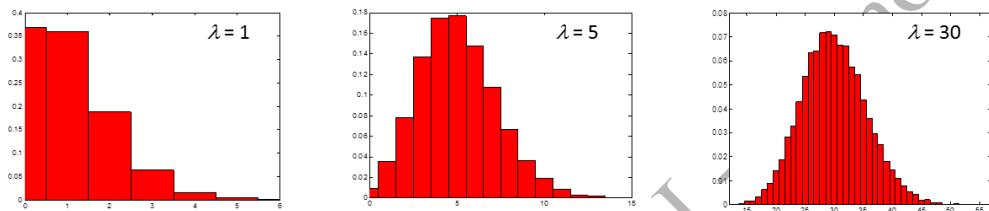


Fig. 3.2 : Evolution de la distribution d'une variable aléatoire de Poisson en fonction du paramètre λ

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 $U \sim U_{[0,1]}$ 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.

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

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

$$= \frac{\Pr(Y = k) \Pr\left(U \leq \frac{p_k}{c q_k}\right)}{\Pr(\text{Acceptation})} = \frac{q_k p_k}{\Pr(\text{Acceptation}) c q_k} = \frac{p_k}{c \Pr(\text{Acceptation})}$$

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

Cela nous permet d'établir le pseudocode final de l'algorithme de la méthode de rejet/acceptation dans le contexte discret.

Algorithme

```

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

Il suffit d'écrire la procédure suivante pour générer une séquence de réalisations de la loi de distribution de X :

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 complétement 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.

Il faut noter 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 . Cependant, dans la solution proposée actuelle, k_n'a pas été généré à partir d'un générateur de Y . Ce choix a pour conséquence de ralentir davantage la convergence de l'algorithme de simulation de X à partir de Y .

C'est pourquoi en informatique, il est parfois acceptable de faire des choix qui ne correspondent pas exactement à l'idéal mathématique, à condition qu'il existe un intérêt justifiant ces choix et que nous puissions les juger acceptables ou les maîtriser (ou les contourner) en trouvant des solutions alternatives pour atténuer leurs impacts négatifs.

Contrairement à ce que pourrait penser un novice, un bon informaticien doit aussi être un bon mathématicien. Cela lui permet de comprendre d'un côté les conséquences négatives résultant de choix éloignés des idéaux théoriques, et de l'autre côté, de savoir comment surmonter ces conséquences s'il existe des avantages pratiques justifiant ces choix. Cette affirmation est ce que nous cherchons à défendre dans cet exercice, et nous tenterons de la soutenir davantage dans l'exercice E 3.4. Toutefois, il est important de noter que cette idée ne devrait en aucun cas justifier les erreurs que certains algorithmiciens et programmeurs pourraient commettre !

4 Méthode de composition

La méthode de composition ou de mélange en simulation est une technique utilisée pour modéliser un système complexe en combinant des éléments plus simples. Ce sont des techniques qui utilisent une combinaison de différentes distributions de probabilité pour modéliser des systèmes stochastiques complexes. Ils peuvent être utiles pour capturer des fonctionnalités telles que la multimodalité, l'hétérogénéité, la non-linéarité et l'incertitude dans les données. Cette approche est

couramment utilisée en simulation stochastique, en particulier dans les domaines tels que la simulation de systèmes physiques, la simulation de processus industriels, la simulation de jeux vidéo, etc. La méthode de composition permet de diviser un système en parties modélisables individuellement, de leur construire des simulateurs, puis de combiner les simulations pour obtenir le comportement global du système.

Dans le contexte des variables aléatoires discrètes, considérons deux variables aléatoires réelles discrètes données, notées X_1 et X_2 . Lorsque la variable aléatoire X résulte du mélange des variables X_1 et X_2 , cela peut être exprimé par

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

où $\alpha \in]0,1[$.

De cette façon, la simulation de la variable aléatoire X en utilisant la méthode de composition peut être réalisée en se basant sur les simulateurs \tilde{X}_1 et \tilde{X}_2 des variables aléatoires X_1 et X_2 respectivement. Ainsi,

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

En d'autres termes, le processus de simulation peut être divisé en quatre étapes distinctes en suivant le schéma détaillé dans l'algorithme ci-après.

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 .

En utilisant la notation pseudocode, le processus peut être représenté comme suit :

```
Algorithm
Proc Simul_comp(alpha)
Local u,x:
u ← Rnd ();
Si (u < alpha) Alors
    x ← Simul_X1()
Sinon
    x ← Simul_X2()
Fin_si;
Retourner(x)
Fin_proc.
```

Pour générer une séquence, nous pouvons écrire :

```
Algorithm
Proc Comp_sequence(N,alpha)
local i , seq ;
Pour i ← 1 à N Faire
    seq[i] ← Simul_comp(alpha)
Fin_pour ;
Afficher(seq)
Fin_proc.
```

La méthode de mixture simplifie la modélisation du comportement des systèmes complexes en les décomposant en comportements plus gérables pour la simulation. Elle offre ainsi la possibilité de réutiliser les modèles de simulation existants des divers comportements partiels afin de construire des systèmes ayant des comportements plus complexes. Cette approche évite la création de modèles excessivement complexes et permet d'économiser du temps et des ressources lors du développement de comportements composés.

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 Matlab (ou 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

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

- Produisez en pseudocode un simulateur pour une séquence de longueur m suivant la loi binomiale négative de paramètres n et p .

- b) Ecrivez ce programme en Matlab (ou 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 Matlab (ou 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 Matlab (ou 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

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

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

Chapitre 4 : Simulation de variables aléatoires continues

1 Introduction

La simulation d'une variable aléatoire implique la génération délibérée d'observations provenant de cette variable. Jusqu'à présent, notre exploration s'est concentrée sur la génération artificielle d'événements et de séquences de valeurs pour des variables aléatoires discrètes. Il est important de noter que pour chaque méthode utilisée dans la génération de variables aléatoires discrètes, il existe une approche similaire pour les variables aléatoires continues. Le chapitre actuel se propose d'approfondir cette question, en se concentrant spécifiquement sur les variables aléatoires continues.

La simulation des variables aléatoires continues revêt une importance particulière dans de nombreux domaines scientifiques et techniques, notamment en ingénierie, en finance et en sciences naturelles. La génération précise de séquences de ces variables est essentielle pour la compréhension des comportements de phénomènes continus. Ainsi, ce chapitre vise à offrir une vue d'ensemble exhaustive des différentes méthodes utilisées pour générer des séquences de variables aléatoires continues, explorant à la fois les techniques classiques et les développements récents dans ce domaine en constante évolution.

2 Méthode de transformation inverse

2.1 Principe de construction

La méthode de transformation inverse est en effet une technique essentielle utilisée pour générer des échantillons aléatoires à partir d'une distribution de probabilité donnée. Dans le contexte d'une

variable aléatoire continue X , la distribution de X peut être caractérisée par sa fonction de densité de probabilité $f(x)$ ou sa fonction de répartition $F(x)$. L'objectif principal de la méthode de transformation inverse est de générer des valeurs aléatoires qui suivent la même distribution que la variable aléatoire X . Pour ce faire, on utilise la fonction de répartition $F(x)$ pour calculer l'inverse de cette fonction, noté $F^{-1}(u)$, où u est une variable aléatoire uniforme sur l'intervalle $[0,1]$. En générant des échantillons de u et en les transformant à l'aide de $F(u)$, on obtient des échantillons aléatoires de la variable X souhaitée.

Définition

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

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

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.

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

Dans le cadre des variables aléatoires continues, la méthode de transformation inverse est une technique utilisée pour générer des échantillons aléatoires à partir d'une variable aléatoire continue avec une fonction de répartition (CDF) connue. Cette méthode repose sur l'inversion de la fonction de répartition, permettant ainsi de transformer des échantillons de variables aléatoires uniformes en échantillons de la variable aléatoire souhaitée. Ainsi, la méthode de transformation inverse pour la simulation de variables aléatoires continues se base sur la connaissance de la forme analytique de l'inverse F^{-1} de la fonction de répartition F .

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,

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

Ce qui montre le résultat.

Cette méthode est utiliser par excellence lorsque la fonction de répartition F possède un inverse dans la forme analytique est connue.

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.
- Implémentez en Matlab (ou 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.

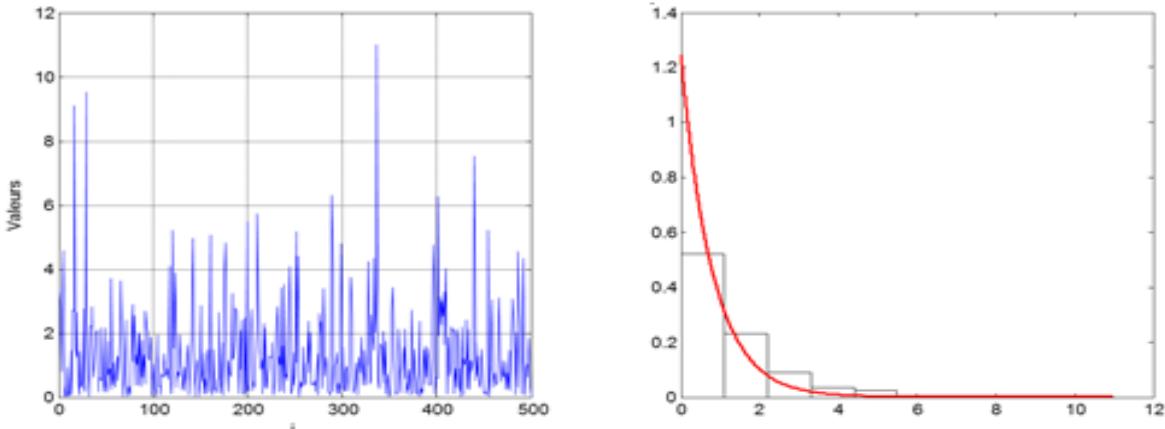


Fig. 3.1 : Séquence de 500 nombres selon une loi exponentielle de paramètre $\lambda = 2$ comparaison des distributions empirique et théorique de cette loi

3 Méthode de rejet/acceptation (variable de contrôle)

La méthode de rejet/acceptation est une technique utilisée pour générer des échantillons aléatoires à partir d'une distribution de probabilité donnée $f(x)$ en utilisant une distribution auxiliaire $g(x)$ que l'on sait reproduire facilement de façon artificielle si

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

Cette propriété signifie que la variable aléatoire X est contrôlée par la variable aléatoire Y . Dans ces conditions, la méthode repose sur le principe de rejet/acceptation, dans le sens où les éléments de l'échantillon généré à partir de la distribution $g(x)$ sont acceptés ou rejettés en fonction de leur position par rapport à la valeur pour la distribution $f(x)$. Si un élément de l'échantillon est accepté, il est retenu ; sinon, il est rejeté et un nouvel élément est généré. La simulation de X à partir de Y se fait alors selon l'algorithme ci-après.

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.

La proposition suivante nous renseigne sur le comportement et la complexité aléatoire du simulateur \tilde{X} ainsi obtenu.

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 $y \in \text{Supp}(X)$

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

Par ailleurs, d’après le théorème des probabilités totales

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

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

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

4 Méthode de composition

Comme dans le cas des variables discrètes, la simulation d'un comportement composé de deux variables aléatoires continues peut être obtenue à partir des échantillons simulés des comportements partiels. Ainsi, la simulation d'un mélange X de deux caractères aléatoires X_1 et X_2 indépendants dans des proportions α 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

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

Cela signifie que pour générer la valeur de la variable aléatoire X , nous pouvons commencer par générer un nombre aléatoire u , qui suit une distribution uniforme, $U \sim \mathcal{U}_{[0,1]}$, et ensuite utiliser ce nombre pour définir la valeur de X comme étant la valeur x_1 du simulateur \tilde{X}_1 si $u < \alpha$ et la valeur x_2 du simulateur \tilde{X}_2 si $u > \alpha$.

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 Matlab (ou 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 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

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

- 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 Matlab (ou Octave) un programme qui visualise les principales propriétés des séquences de variables aléatoires 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.

- Déterminez le support de X .
- Déterminez la constante C .
- En utilisant comme variable de contrôle une v.a. uniforme adéquate, écrivez en pseudocode un algorithme pour la simulation la v.a. X .
- Ecrivez en Matlab (ou 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

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

- Déterminez le support de X .
- Déterminez la constante C .
- Montrez que la méthode de rejet/acceptation est applicable.
- Ecrivez en pseudocode une procédure pour générer une séquence de taille N de la loi de X .
- Comment utiliseriez-vous cet algorithme pour simuler une loi normale $\mathcal{N}(m, \sigma)$ avec $m \neq 0$ et $\sigma > 0$.
- Ecrivez en Matlab (ou 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.

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

Références bibliographiques

- Acín, A., Masanes, L. (2016). Certified randomness in quantum physics. *Nature* 540, 213–219. <https://doi.org/10.1038/nature20119>.
- Verma, S. P. (2020). Monte Carlo Simulation. Chap. 8, in Verma, S. P. (Ed.), *Road from Geochemistry to Geochemometrics*. Springer. https://doi.org/10.1007/978-981-13-9278-8_8
- E. Knuth. (1981). *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 2nd edition.
- Mordechai, S. (Ed.). (2011). *Applications of Monte Carlo Method in Science and Engineering*. InTech, DOI: 10.5772/1954.
- Rae, A. L. and Alvarez, R.M. (2018). Introduction to Polling and Survey Methods, in Lonna Rae Atkeson, and R. Michael Alvarez (eds), *The Oxford Handbook of Polling and Survey Methods*, Oxford Handbooks. <https://doi.org/10.1093/oxfordhb/9780190213299.013.34>
- Metropolis, N. and S. Ulam (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247), 335-341. <https://doi.org/10.2307/2280232>
- Tippet, L. H. C. (1927). Random number tables. *Tracts of computer*, vol. 15, Cambridge University Press.

Fisher, R. A., Yates F. (1938). Statistical table for biological, agricultural and medical research, 6th Edition (1982), Longman Group Limited, England, pp. 37-38 & 134-139.

Kendall, M.G. and Smith, B-B. (1938). Randomness and Random sampling numbers, Jour. Roy. Sat. Soc., 103, pp. 147-166.

Rand Corp. (1955). A Million Random Digits with 100,000 Normal Deviates. MR-1418.
https://www.rand.org/pubs/monograph_reports/MR1418.html#download

Xin, L. and Xin, H. (2023). Decision-making under uncertainty – a quantum value operator approach. Int J Theor Phys 62, 48. <https://doi.org/10.1007/s10773-023-05308-w>

Hull, T.E. and Dobell, A.R. (1962). Random number generators. Siam Review, Vol. 4, N°3, pp 230-254.